FreeFEM Documentation

Release 4.0

Frederic Hecht

Mar 25, 2019

CONTENTS

1 Introduction **1**
1.1 Download FreeFEM . . . 2
1.2 Installation guide . . . 2
1.3 Contributing . . . 9
1.4 Citation . . . 10
1.5 Authors . . . 10

2 Tutorials **13**
2.1 Getting started . . . 15
2.2 Classification of partial differential equations . . . 20
2.3 Membrane . . . 23
2.4 Heat Exchanger . . . 28
2.5 Acoustics . . . 31
2.6 Thermal Conduction . . . 33
2.7 Irrotational Fan Blade Flow and Thermal effects . . . 37
2.8 Pure Convection : The Rotating Hill . . . 40
2.9 The System of elasticity . . . 45
2.10 The System of Stokes for Fluids . . . 47
2.11 A projection algorithm for the Navier-Stokes equations . . . 48
2.12 Newton Method for the Steady Navier-Stokes equations . . . 53
2.13 A Large Fluid Problem . . . 56
2.14 An Example with Complex Numbers . . . 62
2.15 Optimal Control . . . 63
2.16 A Flow with Shocks . . . 67
2.17 Time dependent schema optimization for heat equations . . . 69
2.18 Tutorial to write a transient Stokes solver in matrix form . . . 72
2.19 Wifi Propagation . . . 74
2.20 Plotting in Matlab and Octave . . . 77

3 Documentation **85**
3.1 Notations . . . 86
3.2 Mesh Generation . . . 89
3.3 Finite element . . . 151
3.4 Visualization . . . 188
3.5 Algorithms & Optimization . . . 196
3.6 Parallelization . . . 224
3.7 Plugins . . . 264
3.8 Developers . . . 270
3.9 ffddm . . . 286

4 Language references **315**
4.1 Types . . . 316
4.2 Global variables . . . 330
4.3 Quadrature formulae . . . 335
4.4 Operators . . . 339
4.5 Loops . . . 342
4.6 I/O . . . 344
4.7 Functions . . . 347
4.8 External libraries . . . 385

5 Mathematical Models **461**
5.1 Static problems . . . 461
5.2 Elasticity . . . 482
5.3 Non-linear static problems . . . 491
5.4 Eigen value problems . . . 494
5.5 Evolution problems . . . 498
5.6 Navier-Stokes equations . . . 506
5.7 Variational Inequality . . . 516
5.8 Domain decomposition . . . 519
5.9 Fluid-structure coupled problem . . . 526
5.10 Transmission problem . . . 530
5.11 Free boundary problems . . . 533
5.12 Non-linear elasticity . . . 536
5.13 Compressible Neo-Hookean materials . . . 540
5.14 Whispering gallery modes . . . 549

6 Examples **553**
6.1 Misc . . . 553
6.2 Mesh Generation . . . 559
6.3 Finite Element . . . 571
6.4 Visualization . . . 574
6.5 Algorithms & Optimizations . . . 578
6.6 Parallelization . . . 592
6.7 Developers . . . 605

Bibliography **633**

CHAPTER

ONE

INTRODUCTION

FreeFEM is a partial differential equation solver for non-linear multi-physics systems in 2D and 3D.

Problems involving partial differential equations from several branches of physics, such as fluid-structure interactions, require interpolations of data on several meshes and their manipulation within one program. **FreeFEM** includes a fast interpolation algorithm and a language for the manipulation of data on multiple meshes.

FreeFEM is written in C++ and its language is a C++ idiom.

1.1 Download FreeFEM

1.1.1 Latest binary packages

Operating System	FreeFEM Version	Size	Date
MacOS 10.13	`3.61`	445.3 Mb	May 09, 2018
MacOS 10.12	`3.59`	507.1 Mb	Mar 16, 2018
MacOS 10.11	`3.61`	376.5 Mb	Jul 12, 2018
MacOS 10.10	`3.59`	507.1 Mb	Mar 16, 2018
Ubuntu	`3.62`	27.6 Mb	Jan 14, 2019
Windows 64bits	`3.61`	87.6 Mb	Jul 12, 2018
Windows 32bits	`3.46`	66.4 Mb	May 09, 2017
Source_4.0	`4.0-beta`	3.2 Mb	Jan 18, 2019
Source	`latest`	–	Feb 22, 2018
Older versions	–	–	–

The source code is available on the FreeFEM GitHub Repository.

1.1.2 Syntax highlighters

Lexer type	Version	Description
Emacs	`0.3`	freefem-mode.el
Textmate 2	`1.0`	FreeFem.tmbundle
Gedit	`1.0`	ffpp.lang
Atom	`0.3`	language-freefem or via the Atom package manager
Pygments	`1.0`	freefem.py
Vim	`0.1`	edp.vim

1.2 Installation guide

1.2.1 Easy installation

First, go to the *download page* and choose your platform: Linux, MacOS or Windows.

Note: Binary packages are available for Microsoft Windows, MacOS and some Linux distributions.

Install **FreeFEM** by double-clicking on the appropriate file. Under Linux and MacOS the install directory is one of the following `/usr/local/bin`, `/usr/local/share/freefem++`, `/usr/local/lib/ff++`

Windows binary installation

First download the windows installation executable, then double click to install **FreeFEM**.

In most cases just answer yes (or type return) to all questions.

Otherwise in the Additional Task windows, check the box "Add application directory to your system path." This is required otherwise the program `ffglut.exe` will not be found.

By now you should have two new icons on your desktop:

- `FreeFem++ (VERSION).exe`, the `freefem++` application.
- `FreeFem++ (VERSION) Examples`, a link to the `freefem++` examples folder.

where `(VERSION)` is the version of the files (for example 3.59).

By default, the installed files are in `C:\Programs Files\FreeFem++`. In this directory, you have all the `.dll` files and other applications: `FreeFem++-nw.exe`, `ffglut.exe`, ... The syntax for the command-line tools are the same as those of `FreeFem.exe`.

MacOS X binary installation

Download the MacOS X binary version file, extract all the files by double clicking on the icon of the file, go the the directory and put the `FreeFem+.app` application in the `/Applications` directory.

If you want terminal access to **FreeFEM** just copy the file `FreeFem++` in a directory of your `$PATH` shell environment variable.

Arch AUR package

An up-to-date package of **FreeFEM** for Arch is available on the Archlinux user repository.

To install it:

```
git clone https://aur.archlinux.org/freefem++-git.git
cd freefem++-git
makepkg -si
```

Note: Thanks to Stephan Husmann

1.2.2 Text-editor

Atom

In order to get the syntax highlighting in Atom, you have to install the FreeFEM language support.

You can do it directly in Atom: Edit -> Preferences -> Install, and search for `language-freefem-offical`.

To launch scripts directly from Atom, you have to install the `atom-runner` package. Once installed, modify the Atom configuration file (Edit -> Config...) to have something like that:

```
"*":
   ...

   runner:
      extensions:
         edp: "FreeFem++"
      scopes:
         "Freefem++": "FreeFem++"
```

Reboot Atom, and use Alt+R to run a FreeFem++ script.

Gedit

In order to get the syntax highlighting in Gedit, you have to downlaod the Gedit parser and copy it in `/usr/share/gtksourceview-3.0/language-specs/`.

1.2.3 Compilation

Compilation on OSX (>=10.13)

Remark: Blocks of code are shell commands in terminal.

1. Install Xcode, Xcode Command Line tools and Xcode Additional Tools from the Apple website
2. Install gcc from http://hpc.sourceforge.net

```
curl -O http://prdownloads.sourceforge.net/hpc/gfortran-8.1-bin.tar.gz?download
sudo tar zxvf gfortran-8.1-bin.tar.gz -C /
```

3. Install autoconf and automake from macport or with Homebrew

```
sudo port install autoconf
sudo port install automake
```

```
brew install autoconf
brew install automake
```

4. Install mactex from ctan
5. Install the openmpi source code

```
./configure CC=/usr/local/bin/gcc CXX=/usr/local/bin/g++ F77=/usr/local/bin/
↪gfortran FC=/usr/local/bin/gfortran
make
sudo make install
```

6. Install gsl

```
curl -O https://fr.mirror.babylon.network/gnu/gsl/gsl-2.4.tar.gz
tar zxvf gsl-2.4.tar.gz
cd gsl-2.4
./configure CC=/usr/local/bin/gcc
make
sudo make install
```

7. Install git
8. Download the **FreeFEM** source from the repository

```
git clone https://github.com/FreeFem/FreeFem-sources.git
```

9) Compile **FreeFEM**. Don't forget to update the MacOS SDK version with your own in the command below:

```
cd FreeFem-sources
autoreconf -i
./configure '-with-suffix=macos-10.13' '-without-fltk' '--enable-download' '--
↪enable-optim' 'MPIRUN=/usr/local/bin/mpirun' '--enable-m64' '--without-x'
↪'CC=clang -isysroot /Applications/Xcode.app//Contents/Developer/Platforms/
↪MacOSX.platform/Developer/SDKs/MacOSX10.13.sdk' 'CFLAGS=-mmacosx-version-min=10.
↪13' 'CXXFLAGS=-mmacosx-version-min=10.13 -std=c++11' 'CXX=clang++ -isysroot /
↪Applications/Xcode.app//Contents/Developer/Platforms/MacOSX.platform/Developer/
↪SDKs/MacOSX10.13.sdk' 'F77=/usr/local/bin/gfortran' 'FC=/usr/local/bin/gfortran
↪' 'MPICXX=/usr/local/bin/mpic++' 'MPICC=/usr/local/bin/mpicc' 'MPIFC=/usr/local/
↪bin/mpif90' 'MPIF77=/usr/local/bin/mpif90' '--enable-maintainer-mode'
```

(continues on next page)

(continued from previous page)

```
make
make check
sudo make install
```

Compilation on Ubuntu

1. Install the following dependencies

```
sudo apt-get update && sudo apt-get upgrade
sudo apt-get install cpp freeglut3-dev g++ gcc gfortran \
    ghostscript m4 make patch pkg-config wget python unzip \
    libopenblas-dev liblapack-dev libhdf5-dev libgsl-dev \
    libscotch-dev libfftw3-dev libarpack2-dev libsuitesparse-dev \
    libmumps-seq-dev libnlopt-dev coinor-libipopt-dev libgmm++-dev libtet1.5-dev \
    gnuplot-qt autoconf automake autotools-dev bison flex gdb valgrind git cmake

# mpich is required for the FreeFem parallel computing version
sudo apt-get install mpich
```

> **Warning:** In the oldest distribution of Ubuntu, `libgsl-dev` does not exists, use `libgsl2-dev` instead

2. Download **FreeFEM** source from the repository

```
git clone https://github.com/FreeFem/FreeFem-sources.git
```

3. Autoconf

```
cd FreeFem-sources
autoreconf -i
```

> **Note:** if your autoreconf version is too old, do `tar zxvf AutoGeneratedFile.tar.gz`

4. Configure

```
./configure --enable-download --enable-optim --disable-pastix
```

> **Note:** To see all the options, type `./configure --help`

5. Download the packages

```
./download/getall -a
```

> **Note:** All the third party packages have their own licence

6. Download and compile petsc & slepc

```
cd download/ff-petsc
make petsc-slepc SUDO=sudo
cd -
```

7. Reconfigure with petsc and slepc

```
./reconfigure
```

8. Build

```
make
```

Note: If your computer has many threads, you can run `make` in parallel using `make -j16` for 16 threads, for example.

Note: Optionnally, check the compilation with `make check`

9. Install

```
sudo make install
```

Compilation on Arch Linux

Warning: As Arch is in rolling release, the following information can be quickly outdated !

Warning: **FreeFEM** fails to compile using the newest version of gcc 8.1.0, use an older one instead.

1. Install the following dependencies:

```
pacman -Syu
pacman -S git openmpi gcc-fortran wget python
    freeglut ghostscript m4 make patch gmm
    blas lapack hdf5 gsl fftw arpack suitesparse
    gnuplot autoconf automake bison flex gdb
    valgrind cmake texlive-most
```

2. Download the **FreeFEM** source from the repository

```
git clone https://github.com/FreeFem/FreeFem-sources.git
```

3. Autoconf

```
cd FreeFem-sources
autoreconf -i
```

4. Configure

```
./configure --enable-download --enable-optim --disable-pastix
```

Note: To see all the options, type `./configure --help`

5. Download the packages

```
./download/getall -a
```

Note: All the third party packages have their own licence

6. Download and compile petsc & slepc

```
cd download/ff-petsc
make petsc-slepc SUDO=sudo
cd -
```

7. Reconfigure with petsc and slepc

```
./reconfigure
```

8. Build

```
make
```

Note: If your computer has many threads, you can run `make` in parallel using `make -j16` for 16 threads, for example.

Note: Optionnally, check the compilation with `make check`

9. Install

```
sudo make install
```

Compilation on Linux with Intel software tools

Follow the guide

Compilation on Windows

1. Install MS MPI v7 (msmpisdk.msi and MSMpiSetup.exe)
2. Install Msys2 (x86_64 version)
3. Start MSYS2 MSYS
4. Open `MSYS2 MSYS terminal` to install dependancies
   - for 64bits system:

```
pacman -Syu
pacman -S autoconf automake-wrapper bash bash-completion \
    bison bsdcpio bsdtar bzip2 coreutils curl dash file filesystem \
    findutils flex gawk gcc gcc-fortran gcc-libs grep gzip inetutils info less␣
↪lndir \
    make man-db git mingw-w64-x86_64-freeglut mingw-w64-x86_64-gcc \
    mingw-w64-x86_64-gcc-fortran mingw-w64-x86_64-gsl mingw-w64-x86_64-hdf5 \
    mingw-w64-x86_64-openblas mintty msys2-keyring msys2-launcher-git \
    msys2-runtime ncurses pacman pacman-mirrors pactoys-git patch pax-git \
    perl pkg-config pkgfile rebase sed tar tftp-hpa time tzcode unzip util-linux␣
↪which
```

- for 32bits system:

```
pacman -Syu
pacman -S autoconf automake-wrapper bash bash-completion \
    bison bsdcpio bsdtar bzip2 coreutils curl dash file filesystem \
    findutils flex gawk gcc gcc-fortran gcc-libs grep gzip inetutils info less␣
↪lndir \
    make man-db git mingw-w64-i686-freeglut mingw-w64-i686-gcc \
    mingw-w64-i686-gcc-fortran mingw-w64-i686-gsl mingw-w64-i686-hdf5 \
    mingw-w64-i686-openblas mintty msys2-keyring msys2-launcher-git \
    msys2-runtime ncurses pacman pacman-mirrors pactoys-git patch pax-git \
perl pkg-config pkgfile rebase sed tar tftp-hpa time tzcode unzip util-linux which
```

5. Open `MingW64 terminal` (or `MingW32`) to compile **FreeFEM**

```
git clone https://github.com/FreeFem/FreeFem-sources
cd FreeFem-sources
autoreconf -i
./configure --enable-download --disable-pastix --disable-hips
./download/getall -a
make -j4
make check
make install
```

The **FreeFEM** executable (and some other like `ffmedit`, ...) are in `C:\msys64\mingw64\bin` (or `C:\msys32\mingw32\bin`).

1.2.4 Environment variables and init file

FreeFEM reads a user's init file named `freefem++.pref` to initialize global variables: `verbosity`, `includepath`, `loadpath`.

Note: The variable `verbosity` changes the level of internal printing (0: nothing unless there are syntax errors, 1: few, 10: lots, etc. ...), the default value is 2.

The included files are found in the `includepath` list and the load files are found in the `loadpath` list.

The syntax of the file is:

```
verbosity = 5
loadpath += "/Library/FreeFem++/lib"
loadpath += "/Users/hecht/Library/FreeFem++/lib"
includepath += "/Library/FreeFem++/edp"
```

(continues on next page)

(continued from previous page)

```
includepath += "/Users/hecht/Library/FreeFem++/edp"
# This is a comment
load += "funcTemplate"
load += "myfunction"
load += "MUMPS_seq"
```

The possible paths for this file are

- under Unix and MacOs

```
/etc/freefem++.pref
$(HOME)/.freefem++.pref
freefem++.pref
```

- under windows

```
freefem++.pref
```

We can also use shell environment variables to change verbosity and the search rule before the init files.

```
export FF_VERBOSITY=50
export FF_INCLUDEPATH="dir;;dir2"
export FF_LOADPATH="dir;;dir3"
```

Note: The separator between directories must be ";" and not ":" because ":" is used under Windows.

Note: To show the list of init of **FreeFEM** , do

```
export FF_VERBOSITY=100;
./FreeFem++-nw
```

1.3 Contributing

1.3.1 Bug report

Concerning the FreeFEM documentation

Open an Issue on **FreeFem-doc** repository.

Concerning the FreeFEM compilation or usage

Open an Issue on **FreeFem-sources** repository.

1.3.2 Improve content

Ask one of the contributors for Collaborator Access or make a Pull Request.

1.4 Citation

1.4.1 If you use FreeFEM, please cite the following reference in your work:

APA

```
Hecht, F. (2012). New development in FreeFem++. Journal of numerical mathematics, 20(3-4), 251-266.
```

ISO690

```
HECHT, Frédéric. New development in FreeFem++. Journal of numerical mathematics, 2012, vol. 20, no 3-4, p. 251-266.
```

MLA

```
Hecht, Frédéric. "New development in FreeFem++." Journal of numerical mathematics 20. 3-4 (2012): 251-266.
```

BibTeX

```bibtex
@article{MR3043640,
  AUTHOR = {Hecht, F.},
  TITLE = {New development in FreeFem++},
  JOURNAL = {J. Numer. Math.},
  FJOURNAL = {Journal of Numerical Mathematics},
  VOLUME = {20}, YEAR = {2012},
  NUMBER = {3-4}, PAGES = {251--265},
  ISSN = {1570-2820},
  MRCLASS = {65Y15},
  MRNUMBER = {3043640}
}
```

1.5 Authors

Frédéric Hecht
Professor at Laboratoire Jacques Louis Lions (LJLL), Pierre and Marie Curie University, Paris
frederic.hecht@sorbonne-universite.fr
https://www.ljll.math.upmc.fr/hecht/

Sylvain Auliac
Former PhD student at LJLL, optimization interface with nlopt, ipopt, cmaes, ...
https://www.ljll.math.upmc.fr/auliac/

Olivier Pironneau
Professor of numerical analysis at the Paris VI university and at LJLL, numerical methods in fluid
Member of the Institut Universitaire de France and Academie des Sciences
https://www.ljll.math.upmc.fr/pironneau/

Jacques Morice
Former Post-Doc at LJLL, three dimensions mesh generation and coupling with medit

Antoine Le Hyaric
Research engineer from CNRS, expert in software engineering for scientific applications, electromagnetics simulations, parallel computing and three-dimensionsal visualization
https://www.ljll.math.upmc.fr/lehyaric/

Kohji Ohtsuka
Professor at Hiroshima Kokusai Gakuin University, Japan and chairman of the World Scientific and Engineering Academy and Society, Japan. Fracture dynamic, modeling and computing
https://sites.google.com/a/comfos.org/comfos/

Pierre Jolivet
CNRS researcher, MPI interface with PETSc, HPDDM, . . .
http://jolivet.perso.enseeiht.fr/

And all the dedicated Github contributors

CHAPTER

TWO

TUTORIALS

The **FreeFEM** language is *typed*, polymorphic and reentrant with *macro generation*.

Every variable must be typed and declared in a statement. Each statement is separated from the next by a semicolon `;`.

The **FreeFEM** language is a C++ idiom with something that is more akin to LaTeX.

For the specialist, one key guideline is that **FreeFEM** rarely generates an internal finite element array, this was adopted for speed and consequently **FreeFEM** could be hard to beat in terms of execution speed, except for the time lost in the interpretation of the language (which can be reduced by a systematic usage of `varf` and `matrix` instead of `problem`).

The Development Cycle: Edit–Run/Visualize–Revise

Many examples and tutorials are given there after and in the *examples section*. It is better to study them and learn by example.

If you are a beginner in the finite element method, you may also have to read a book on variational formulations.

The development cycle includes the following steps:

Modeling: From strong forms of PDE to weak forms, one must know the variational formulation to use **FreeFEM**; one should also have an eye on the reusability of the variational formulation so as to keep the same internal matrices; a typical example is the time dependent heat equation with an implicit time scheme: the internal matrix can be factorized only once and **FreeFEM** can be taught to do so.

Programming: Write the code in **FreeFEM** language using a text editor such as the one provided in your integrated environment.

Run: Run the code (here written in file `mycode.edp`). That can also be done in terminal mode by :

```
FreeFem++ mycode.edp
```

Visualization: Use the keyword `plot` directly in `mycode.edp` to display functions while **FreeFEM** is running. Use the plot-parameter `wait=1` to stop the program at each plot.

Debugging: A global variable `debug` (for example) can help as in `wait=true` to `wait=false`.

```
bool debug = true;

border a(t=0, 2.*pi){x=cos(t); y=sin(t); label=1;};
border b(t=0, 2.*pi){x=0.8+0.3*cos(t); y=0.3*sin(t); label=2;};

plot(a(50) + b(-30), wait=debug); //plot the borders to see the intersection
//so change (0.8 in 0.3 in b)
//if debug == true, press Enter to continue

```

(continues on next page)

(continued from previous page)

```
mesh Th = buildmesh(a(50) + b(-30));
plot(Th, wait=debug); //plot Th then press Enter

fespace Vh(Th,P2);
Vh f = sin(pi*x)*cos(pi*y);
Vh g = sin(pi*x + cos(pi*y));

plot(f, wait=debug); //plot the function f
plot(g, wait=debug); //plot the function g
```

Changing debug to false will make the plots flow continuously. Watching the flow of graphs on the screen (while drinking coffee) can then become a pleasant experience.

Error management

Error messages are displayed in the console window. They are not always very explicit because of the template structure of the C++ code (we did our best!). Nevertheless they are displayed at the right place. For example, if you forget parenthesis as in:

```
bool debug = true;
mesh Th = square(10,10;
plot(Th);
```

then you will get the following message from **FreeFEM**:

```
    2 : mesh Th = square(10,10;
 Error line number 2, in file bb.edp, before  token ;
parse error
  current line = 2
Compile error : parse error
        line number :2, ;
error Compile error : parse error
        line number :2, ;
 code = 1
```

If you use the same symbol twice as in:

```
real aaa = 1;
real aaa;
```

then you will get the message:

```
2 : real aaa; The identifier aaa exists
      the existing type is <Pd>
      the new  type is <Pd>
```

If you find that the program isn't doing what you want you may also use `cout` to display in text format on the console window the value of variables, just as you would do in C++.

The following example works:

```
...
fespace Vh(Th, P1);
Vh u;
cout << u;
matrix A = a(Vh, Vh);
cout << A;
```

Another trick is to *comment in and out* by using `//` as in C++. For example:

```
real aaa =1;
// real aaa;
```

2.1 Getting started

For a given function $f(x, y)$, find a function $u(x, y)$ satisfying :

$$\begin{aligned} -\Delta u(x,y) &= f(x,y) && \text{for all } (x,y) \text{ in } \Omega \\ u(x,y) &= 0 && \text{for all } (x,y) \text{ on } \partial\Omega \end{aligned} \tag{2.1}$$

Here $\partial\Omega$ is the boundary of the bounded open set $\Omega \subset \mathbb{R}^2$ and $\Delta u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}$.

We will compute u with $f(x, y) = xy$ and Ω the unit disk. The boundary $C = \partial\Omega$ is defined as:

$$C = \{(x, y)|\ x = \cos(t),\ y = \sin(t),\ 0 \le t \le 2\pi\}$$

Note: In **FreeFEM**, the domain Ω is assumed to be described by the left side of its boundary.

The following is the **FreeFEM** program which computes u:

```
// Define mesh boundary
border C(t=0, 2*pi){x=cos(t); y=sin(t);}

// The triangulated domain Th is on the left side of its boundary
mesh Th = buildmesh(C(50));

// The finite element space defined over Th is called here Vh
fespace Vh(Th, P1);
Vh u, v;// Define u and v as piecewise-P1 continuous functions

// Define a function f
func f= x*y;

// Get the clock in second
real cpu=clock();

// Define the PDE
solve Poisson(u, v, solver=LU)
    = int2d(Th)(    // The bilinear part
          dx(u)*dx(v)
        + dy(u)*dy(v)
    )
    - int2d(Th)(    // The right hand side
          f*v
    )
    + on(C, u=0);   // The Dirichlet boundary condition

// Plot the result
plot(u);

// Display the total computational time
cout << "CPU time = " << (clock()-cpu) << endl;
```

(a) Mesh Th by `buildmesh(C(50))`

(b) Isovalue by `plot(u)`

Fig. 2.1: Poisson's equation

As illustrated in Fig. 2.1b, we can see the isovalue of u by using **FreeFEM** `plot` command (see line 29 above).

Note: The qualifier `solver=LU` (line 18) is not required and by default a multi-frontal `LU` is used.

The lines containing `clock` are equally not required.

Tip: Note how close to the mathematics **FreeFEM** language is.

Lines 19 to 24 correspond to the mathematical variational equation:

$$\int_{T_h} (\frac{\partial u}{\partial x}\frac{\partial v}{\partial x} + \frac{\partial u}{\partial y}\frac{\partial v}{\partial y})\mathrm{d}x\mathrm{d}y = \int_{T_h} fv\mathrm{d}x\mathrm{d}y$$

for all v which are in the finite element space V_h and zero on the boundary C.

Tip: Change `P1` into `P2` and run the program.

This first example shows how **FreeFEM** executes with no effort all the usual steps required by the finite element method (FEM). Let's go through them one by one.

On the line 2:

The boundary Γ is described analytically by a parametric equation for x and for y. When $\Gamma = \sum_{j=0}^{J} \Gamma_j$ then each curve Γ_j must be specified and crossings of Γ_j are not allowed except at end points.

The keyword `label` can be added to define a group of boundaries for later use (boundary conditions for instance). Hence the circle could also have been described as two half circle with the same label:

```
border Gamma1(t=0, pi){x=cos(t); y=sin(t); label=C};
border Gamma2(t=pi, 2.*pi){x=cos(t); y=sin(t); label=C};
```

Boundaries can be referred to either by name (`Gamma1` for example) or by label (`C` here) or even by its internal number here 1 for the first half circle and 2 for the second (more examples are in *Meshing Examples*).

On the line 5

The triangulation $\mathcal{T}_h$ of Ω is automatically generated by `buildmesh(C(50))` using 50 points on `C` as in Fig. 2.1a.

The domain is assumed to be on the left side of the boundary which is implicitly oriented by the parametrization. So an elliptic hole can be added by typing:

```
border C(t=2.*pi, 0){x=0.1+0.3*cos(t); y=0.5*sin(t);};
```

If by mistake one had written:

```
border C(t=0, 2.*pi){x=0.1+0.3*cos(t); y=0.5*sin(t);};
```

then the inside of the ellipse would be triangulated as well as the outside.

Note: Automatic mesh generation is based on the Delaunay-Voronoi algorithm. Refinement of the mesh are done by increasing the number of points on Γ, for example `buildmesh(C(100))`, because inner vertices are determined by the density of points on the boundary.

Mesh adaptation can be performed also against a given function f by calling `adaptmesh(Th,f)`.

Now the name $\mathcal{T}_h$ (`Th` in **FreeFEM**) refers to the family $\{T_k\}_{k=1,\cdots,n_t}$ of triangles shown in Fig. 2.1a.

Traditionally h refers to the mesh size, n_t to the number of triangles in $\mathcal{T}_h$ and n_v to the number of vertices, but it is seldom that we will have to use them explicitly.

If Ω is not a polygonal domain, a "skin" remains between the exact domain Ω and its approximation $\Omega_h = \cup_{k=1}^{n_t} T_k$. However, we notice that all corners of $\Gamma_h = \partial\Omega_h$ are on Γ.

On line 8:

A finite element space is, usually, a space of polynomial functions on elements, triangles here only, with certain matching properties at edges, vertices etc. Here `fespace Vh(Th, P1)` defines V_h to be the space of continuous functions which are affine in x, y on each triangle of T_h.

As it is a linear vector space of finite dimension, basis can be found. The canonical basis is made of functions, called the *hat function* ϕ_k which are continuous piecewise affine and are equal to 1 on one vertex and 0 on all others. A typical hat function is shown on Fig. 2.2b.

Note: The easiest way to define ϕ_k is by making use of the *barycentric coordinates* $\lambda_i(x,y),\ i = 1,2,3$ of a point $q = (x,y) \in T$, defined by $\sum_i \lambda_i = 1,\quad \sum_i \lambda_i \vec{q^i} = \vec{q}$ where $q^i,\ i = 1,2,3$ are the 3 vertices of T. Then it is easy to see that the restriction of ϕ_k on T is precisely λ_k.

Then:

$$V_h(\mathcal{T}_h, P_1) = \left\{ w(x,y) \,\middle|\, w(x,y) = \sum_{k=1}^{M} w_k \phi_k(x,y),\ w_k \text{ are real numbers} \right\} \tag{2.2}$$

where M is the dimension of V_h, i.e. the number of vertices. The w_k are called the *degree of freedom* of w and M the number of degree of freedom.

It is said also that the *nodes* of this finite element method are the vertices.

Setting the problem

(a) `mesh Th`

(b) Graph of ϕ_1 (left) and ϕ_6 (right)

Fig. 2.2: Hat functions

On line 9, `Vh u, v` declares that u and v are approximated as above, namely:

$$u(x,y) \simeq u_h(x,y) = \sum_{k=0}^{M-1} u_k \phi_k(x,y) \tag{2.3}$$

On the line 12, the right hand side `f` is defined analytically using the keyword `func`.

Line 18 to 26 define the bilinear form of equation (2.1) and its Dirichlet boundary conditions.

This *variational formulation* is derived by multiplying (2.1) by $v(x,y)$ and integrating the result over Ω:

$$-\int_\Omega v\Delta u \,\mathrm{d}x\mathrm{d}y = \int_\Omega v f \,\mathrm{d}x\mathrm{d}y$$

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

$$a(u,v) - \ell(f,v) = 0 \quad \forall v \text{ satisfying } v = 0 \text{ on } \partial\Omega.$$

with:

$$\begin{array}{rcl} a(u,v) & = & \int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x\mathrm{d}y \\ \ell(f,v) & = & \int_\Omega f v \,\mathrm{d}x\mathrm{d}y \end{array} \tag{2.4}$$

In **FreeFEM** the **Poisson** problem can be declared only as in:

```
Vh u,v; problem Poisson(u,v) = ...
```

and solved later as in:

```
Poisson; //the problem is solved here
```

or declared and solved at the same time as in:

```
Vh u,v; solve Poisson(u,v) = ...
```

and (2.4) is written with `dx(u)` $= \partial u/\partial x$, `dy(u)` $= \partial u/\partial y$ and:

$\int_\Omega \nabla u \cdot \nabla v \, \mathrm{d}x\mathrm{d}y \longrightarrow$ `int2d(Th)( dx(u)*dx(v) + dy(u)*dy(v) )`

$\int_\Omega f v \, \mathrm{d}x\mathrm{d}y \longrightarrow$ `int2d(Th)( f*v )` (Notice here, u is unused)

> **Warning:** In **FreeFEM bilinear terms and linear terms should not be under the same integral** indeed to construct the linear systems **FreeFEM** finds out which integral contributes to the bilinear form by checking if both terms, the unknown (here `u`) and test functions (here `v`) are present.

Solution and visualization

On line 15, the current time in seconds is stored into the real-valued variable `cpu`.

Line 18, the problem is solved.

Line 29, the visualization is done as illustrated in Fig. 2.1b.

(see *Plot* for zoom, postscript and other commands).

Line 32, the computing time (not counting graphics) is written on the console. Notice the C++-like syntax; the user needs not study C++ for using **FreeFEM**, but it helps to guess what is allowed in the language.

Access to matrices and vectors

Internally **FreeFEM** will solve a linear system of the type

$$\sum_{j=0}^{M-1} A_{ij} u_j - F_i = 0, \quad i = 0, \cdots, M-1; \qquad F_i = \int_\Omega f \phi_i \, \mathrm{d}x\mathrm{d}y \tag{2.5}$$

which is found by using (2.3) and replacing v by ϕ_i in (2.4). The Dirichlet conditions are implemented by penalty, namely by setting $A_{ii} = 10^{30}$ and $F_i = 10^{30} * 0$ if i is a boundary degree of freedom.

Note: The number 10^{30} is called `tgv` (*très grande valeur* or *very high value* in english) and it is generally possible to change this value, see the item :freefem‘solve, tgv=‘

The matrix $A = (A_{ij})$ is called *stiffness matrix*. If the user wants to access A directly he can do so by using (see section *Variational form, Sparse matrix, PDE data vector* for details).

```
varf a(u,v)
    = int2d(Th)(
          dx(u)*dx(v)
        + dy(u)*dy(v)
    )
    + on(C, u=0)
    ;
matrix A = a(Vh, Vh); //stiffness matrix
```

The vector F in (2.5) can also be constructed manually:

```
varf l(unused,v)
    = int2d(Th)(
          f*v
    )
    + on(C, unused=0)
    ;
Vh F;
F[] = l(0,Vh); //F[] is the vector associated to the function F
```

The problem can then be solved by:

```
u[] = A^-1*F[]; //u[] is the vector associated to the function u
```

Note: Here `u` and `F` are finite element function, and `u[]` and `F[]` give the array of value associated (`u[]` $\equiv (u_i)_{i=0,\dots,M-1}$ and `F[]` $\equiv (F_i)_{i=0,\dots,M-1}$).

So we have:

$$\mathtt{u}(x,y) = \sum_{i=0}^{M-1} \mathtt{u}[][i]\phi_i(x,y), \qquad \mathtt{F}(x,y) = \sum_{i=0}^{M-1} \mathtt{F}[][i]\phi_i(x,y)$$

where $\phi_i, i = 0\dots, , M-1$ are the basis functions of *Vh* like in equation (ref{equation3}), and $M =$ `Vh.ndof` is the number of degree of freedom (i.e. the dimension of the space *Vh*).

The linear system (2.5) is solved by `UMFPACK` unless another option is mentioned specifically as in:

```
Vh u, v;
problem Poisson(u, v, solver=CG) = int2d(...
```

meaning that `Poisson` is declared only here and when it is called (by simply writing `Poisson;`) then (2.5) will be solved by the Conjugate Gradient method.

2.2 Classification of partial differential equations

Summary : *It is usually not easy to determine the type of a system. Yet the approximations and algorithms suited to the problem depend on its type:*

- *Finite Elements compatible (LBB conditions) for elliptic systems*
- *Finite difference on the parabolic variable and a time loop on each elliptic subsystem of parabolic systems; better stability diagrams when the schemes are implicit in time.*
- *Upwinding, Petrov-Galerkin, Characteristics-Galerkin, Discontinuous-Galerkin, Finite Volumes for hyperbolic systems plus, possibly, a time loop.*

When the system changes type, then expect difficulties (like shock discontinuities) !

Elliptic, parabolic and hyperbolic equations

A partial differential equation (PDE) is a relation between a function of several variables and its derivatives.

$$F\left(\varphi(x), \frac{\partial\varphi}{\partial x_1}(x), \cdots, \frac{\partial\varphi}{\partial x_d}(x), \frac{\partial^2\varphi}{\partial x_1^2}(x), \cdots, \frac{\partial^m\varphi}{\partial x_d^m}(x)\right) = 0, \quad \forall x \in \Omega \subset \mathbb{R}^d$$

The range of x over which the equation is taken, here Ω, is called the *domain* of the PDE. The highest derivation index, here m, is called the *order*. If F and φ are vector valued functions, then the PDE is actually a *system* of PDEs.

Unless indicated otherwise, here by convention *one* PDE corresponds to one scalar valued F and φ. If F is linear with respect to its arguments, then the PDE is said to be *linear*.

The general form of a second order, linear scalar PDE is $\frac{\partial^2\varphi}{\partial x_i \partial x_j}$ and $A : B$ means $\sum_{i,j=1}^{d} a_{ij} b_{ij}$.

$$\alpha\varphi + a \cdot \nabla\varphi + B : \nabla(\nabla\varphi) = f \quad \text{in} \quad \Omega \subset \mathbb{R}^d,$$

where $f(x), \alpha(x) \in \mathbb{R}$, $a(x) \in \mathbb{R}^d$, $B(x) \in \mathbb{R}^{d\times d}$ are the PDE *coefficients*. If the coefficients are independent of x, the PDE is said to have *constant coefficients*.

To a PDE we associate a quadratic form, by replacing φ by 1, $\partial\varphi/\partial x_i$ by z_i and $\partial^2\varphi/\partial x_i \partial x_j$ by $z_i z_j$, where z is a vector in $\mathbb{R}^d$:

$$\alpha + A \cdot z + z^T B z = f.$$

If it is the equation of an ellipse (ellipsoid if $d \geq 2$), the PDE is said to be *elliptic*; if it is the equation of a parabola or a hyperbola, the PDE is said to be *parabolic* or *hyperbolic*.

If $A \equiv 0$, the degree is no longer 2 but 1, and for reasons that will appear more clearly later, the PDE is still said to be hyperbolic.

These concepts can be generalized to systems, by studying whether or not the polynomial system $P(z)$ associated with the PDE system has branches at infinity (ellipsoids have no branches at infinity, paraboloids have one, and hyperboloids have several).

If the PDE is not linear, it is said to be *non-linear*. Those are said to be locally elliptic, parabolic, or hyperbolic according to the type of the linearized equation.

For example, for the non-linear equation

$$\frac{\partial^2\varphi}{\partial t^2} - \frac{\partial\varphi}{\partial x}\frac{\partial^2\varphi}{\partial x^2} = 1$$

we have $d = 2$, $x_1 = t$, $x_2 = x$ and its linearized form is:

$$\frac{\partial^2 u}{\partial t^2} - \frac{\partial u}{\partial x}\frac{\partial^2\varphi}{\partial x^2} - \frac{\partial\varphi}{\partial x}\frac{\partial^2 u}{\partial x^2} = 0$$

which for the unknown u is locally elliptic if $\frac{\partial\varphi}{\partial x} < 0$ and locally hyperbolic if $\frac{\partial\varphi}{\partial x} > 0$.

Tip: Laplace's equation is elliptic:

$$\Delta\varphi \equiv \frac{\partial^2\varphi}{\partial x_1^2} + \frac{\partial^2\varphi}{\partial x_2^2} + \cdots + \frac{\partial^2\varphi}{\partial x_d^2} = f, \ \forall x \in \Omega \subset \mathbb{R}^d$$

Tip: The *heat* equation is parabolic in $Q = \Omega\times]0, T[\subset \mathbb{R}^{d+1}$:

$$\frac{\partial\varphi}{\partial t} - \mu\Delta\varphi = f \ \forall x \in \Omega \subset \mathbb{R}^d, \ \forall t \in]0, T[$$

Tip: If $\mu > 0$, the *wave* equation is hyperbolic:

$$\frac{\partial^2\varphi}{\partial t^2} - \mu\Delta\varphi = f \ \text{ in } \ Q.$$

Tip: The *convection diffusion* equation is parabolic if $\mu \neq 0$ and hyperbolic otherwise:

$$\frac{\partial \varphi}{\partial t} + a\nabla\varphi - \mu\Delta\varphi = f$$

Tip: The *biharmonic* equation is elliptic:

$$\Delta(\Delta\varphi) = f \text{ in } \Omega.$$

Boundary conditions

A relation between a function and its derivatives is not sufficient to define the function. Additional information on the boundary $\Gamma = \partial\Omega$ of Ω, or on part of Γ is necessary. Such information is called a *boundary condition*.

For example:

$$\varphi(x) \text{ given, } \forall x \in \Gamma,$$

is called a *Dirichlet boundary condition*. The *Neumann* condition is

$$\frac{\partial \varphi}{\partial \mathbf{n}}(x) \text{ given on } \Gamma \text{ (or } \mathbf{n} \cdot B\nabla\varphi, \text{given on } \Gamma \text{ for a general second order PDE)}$$

where $\mathbf{n}$ is the normal at $x \in \Gamma$ directed towards the exterior of Ω (by definition $\frac{\partial \varphi}{\partial \mathbf{n}} = \nabla\varphi \cdot \mathbf{n}$).

Another classical condition, called a *Robin* (or *Fourier*) condition is written as:

$$\varphi(x) + \beta(x)\frac{\partial \varphi}{\partial n}(x) \text{ given on } \Gamma.$$

Finding a set of boundary conditions that defines a unique φ is a difficult art.

In general, an elliptic equation is well posed (*i.e.* φ is unique) with one Dirichlet, Neumann or Robin condition on the whole boundary.

Thus, Laplace's equation is well posed with a Dirichlet or Neumann condition but also with :

$$\varphi \text{ given on } \Gamma_1,\ \frac{\partial \varphi}{\partial n} \text{ given on } \Gamma_2,\ \Gamma_1 \cup \Gamma_2 = \Gamma,\ \dot{\Gamma_1} \cap \dot{\Gamma_2} = \emptyset.$$

Parabolic and hyperbolic equations rarely require boundary conditions on all of $\Gamma \times]0, T[$. For instance, the heat equation is well posed with :

$$\varphi \text{ given at } t = 0 \text{ and Dirichlet or Neumann or mixed conditions on } \partial\Omega.$$

Here t is time so the first condition is called an initial condition. The whole set of conditions is also called Cauchy condition.

The wave equation is well posed with :

$$\varphi \text{ and } \frac{\partial \varphi}{\partial t} \text{ given at } t = 0 \text{ and Dirichlet or Neumann or mixed conditions on } \partial\Omega.$$

2.3 Membrane

Summary : *Here we shall learn how to solve a Dirichlet and/or mixed Dirichlet Neumann problem for the Laplace operator with application to the equilibrium of a membrane under load. We shall also check the accuracy of the method and interface with other graphics packages*

An elastic membrane Ω is attached to a planar rigid support Γ, and a force $f(x)dx$ is exerted on each surface element $\mathrm{d}x = \mathrm{d}x_1\mathrm{d}x_2$. The vertical membrane displacement, $\varphi(x)$, is obtained by solving Laplace's equation:

$$-\Delta\varphi = f \text{ in } \Omega$$

As the membrane is fixed to its planar support, one has:

$$\varphi|_\Gamma = 0$$

If the support wasn't planar but had an elevation $z(x_1, x_2)$ then the boundary conditions would be of non-homogeneous Dirichlet type.

$$\varphi|_\Gamma = z$$

If a part Γ_2 of the membrane border Γ is not fixed to the support but is left hanging, then due to the membrane's rigidity the angle with the normal vector n is zero; thus the boundary conditions are:

$$\varphi|_{\Gamma_1} = z, \quad \frac{\partial\varphi}{\partial n}|_{\Gamma_2} = 0$$

where $\Gamma_1 = \Gamma - \Gamma_2$; recall that $\frac{\partial\varphi}{\partial n} = \nabla\varphi \cdot n$ Let us recall also that the Laplace operator Δ is defined by:

$$\Delta\varphi = \frac{\partial^2\varphi}{\partial x_1^2} + \frac{\partial^2\varphi}{\partial x_2^2}$$

Todo: Check references

With such *"mixed boundary conditions"* the problem has a unique solution (see Dautray-Lions (1988), Strang (1986) and Raviart-Thomas (1983)). The easiest proof is to notice that φ is the state of least energy, i.e.

$$E(\phi) = \min_{\varphi - z \in V} E(v), \quad \text{with} \quad E(v) = \int_\Omega (\frac{1}{2}|\nabla v|^2 - fv)$$

and where V is the subspace of the Sobolev space $H^1(\Omega)$ of functions which have zero trace on Γ_1. Recall that ($x \in \mathbb{R}^d,\ d = 2$ here):

$$H^1(\Omega) = \{u \in L^2(\Omega)\ :\ \nabla u \in (L^2(\Omega))^d\}$$

Calculus of variation shows that the minimum must satisfy, what is known as the weak form of the PDE or its variational formulation (also known here as the theorem of virtual work)

$$\int_\Omega \nabla\varphi \cdot \nabla w = \int_\Omega fw \quad \forall w \in V$$

Next an integration by parts (Green's formula) will show that this is equivalent to the PDE when second derivatives exist.

Warning: Unlike Freefem+ which had both weak and strong forms, **FreeFEM** implements only weak formulations. It is not possible to go further in using this software if you don't know the weak form (i.e. variational formulation) of your problem: either you read a book, or ask help form a colleague or drop the matter. Now if you want to solve a system of PDE like $A(u,v) = 0,\ B(u,v) = 0$ don't close this manual, because in weak form it is

$$\int_\Omega (A(u,v)w_1 + B(u,v)w_2) = 0 \ \ \forall w_1, w_2 ...$$

Example

Let an ellipse have the length of the semimajor axis $a = 2$, and unitary the semiminor axis. Let the surface force be $f = 1$. Programming this case with **FreeFEM** gives:

```
// Parameters
real theta = 4.*pi/3.;
real a = 2.; //The length of the semimajor axis
real b = 1.; //The length of the semiminor axis
func z = x;

// Mesh
border Gamma1(t=0., theta){x=a*cos(t); y=b*sin(t);}
border Gamma2(t=theta, 2.*pi){x=a*cos(t); y=b*sin(t);}
mesh Th = buildmesh(Gamma1(100) + Gamma2(50));

// Fespace
fespace Vh(Th, P2); //P2 conforming triangular FEM
Vh phi, w, f=1;

// Solve
solve Laplace(phi, w)
    = int2d(Th)(
          dx(phi)*dx(w)
        + dy(phi)*dy(w)
    )
    - int2d(Th)(
          f*w
    )
    + on(Gamma1, phi=z)
    ;

// Plot
plot(phi, wait=true, ps="membrane.eps"); //Plot phi
plot(Th, wait=true, ps="membraneTh.eps"); //Plot Th

// Save mesh
savemesh(Th,"Th.msh");
```

A triangulation is built by the keyword `buildmesh`. This keyword calls a triangulation subroutine based on the Delaunay test, which first triangulates with only the boundary points, then adds internal points by subdividing the edges. How fine the triangulation becomes is controlled by the size of the closest boundary edges.

The PDE is then discretized using the triangular second order finite element method on the triangulation; as was briefly indicated in the previous chapter, a linear system is derived from the discrete formulation whose size is the number of vertices plus the number of mid-edges in the triangulation.

The system is solved by a multi-frontal Gauss LU factorization implemented in the package `UMFPACK`.

(a) Mesh of the ellipse **(b)** Level lines of the membrane deformation

Fig. 2.3: Membrane

The keyword `plot` will display both $\mathbb{T}_h$ and φ (remove `Th` if φ only is desired) and the qualifier `fill=true` replaces the default option (colored level lines) by a full color display.

```
plot(phi,wait=true,fill=true); //Plot phi with full color display
```

Results are on Fig. 2.3a and Fig. 2.3b.

Next we would like to check the results !

One simple way is to adjust the parameters so as to know the solutions. For instance on the unit circle `a=1`, $\varphi_e = \sin(x^2 + y^2 - 1)$ solves the problem when:

$$z = 0, f = -4(\cos(x^2 + y^2 - 1) - (x^2 + y^2)\sin(x^2 + y^2 - 1))$$

except that on Γ_2 $\partial_n \varphi = 2$ instead of zero. So we will consider a non-homogeneous Neumann condition and solve:

$$\int_\Omega \nabla\varphi \cdot \nabla w = \int_\Omega f w + \int_{\Gamma_2} 2w \quad \forall w \in V$$

We will do that with two triangulations, compute the L^2 error:

$$\epsilon = \int_\Omega |\varphi - \varphi_e|^2$$

and print the error in both cases as well as the log of their ratio an indication of the rate of convergence.

```
//  Parameters
verbosity = 0; //to remove all default output
real theta = 4.*pi/3.;
real a=1.; //the length of the semimajor axis
real b=1.; //the length of the semiminor axis
func f = -4*(cos(x^2+y^2-1) - (x^2+y^2)*sin(x^2+y^2-1));
func phiexact = sin(x^2 + y^2 - 1);

// Mesh
```

(continues on next page)

(continued from previous page)

```
border Gamma1(t=0., theta){x=a*cos(t); y=b*sin(t);}
border Gamma2(t=theta, 2.*pi){x=a*cos(t); y=b*sin(t);}

// Error loop
real[int] L2error(2); //an array of two values
for(int n = 0; n < 2; n++){
    // Mesh
    mesh Th = buildmesh(Gamma1(20*(n+1)) + Gamma2(10*(n+1)));

    // Fespace
    fespace Vh(Th, P2);
    Vh phi, w;

    // Solve
    solve Laplace(phi, w)
        = int2d(Th)(
              dx(phi)*dx(w)
            + dy(phi)*dy(w)
        )
        - int2d(Th)(
              f*w
        )
        - int1d(Th, Gamma2)(
              2*w
        )
        + on(Gamma1,phi=0)
        ;

    // Plot
    plot(Th, phi, wait=true, ps="membrane.eps");

    // Error
    L2error[n] = sqrt(int2d(Th)((phi-phiexact)^2));
}

// Display loop
for(int n = 0; n < 2; n++)
    cout << "L2error " << n << " = " << L2error[n] << endl;

// Convergence rate
cout << "convergence rate = "<< log(L2error[0]/L2error[1])/log(2.) << endl;
```

The output is:

```
L2error 0 = 0.00462991
L2error 1 = 0.00117128
convergence rate = 1.9829
times: compile 0.02s, execution 6.94s
```

We find a rate of 1.93591, which is not close enough to the 3 predicted by the theory.

The Geometry is always a polygon so we lose one order due to the geometry approximation in $O(h^2)$.

Now if you are not satisfied with the `.eps` plot generated by **FreeFEM** and you want to use other graphic facilities, then you must store the solution in a file very much like in `C++`. It will be useless if you don't save the triangulation as well, consequently you must do

```
{
    ofstream ff("phi.txt");
    ff << phi[];
}
savemesh(Th,"Th.msh");
```

For the triangulation the name is important: **the extension determines the format**.

Fig. 2.4: The 3D version drawn by `gnuplot` from a file generated by **FreeFEM**

Still that may not take you where you want. Here is an interface with gnuplot to produce the Fig. 2.4.

```
//to build a gnuplot data file
{
    ofstream ff("graph.txt");
    for (int i = 0; i < Th.nt; i++)
    {
        for (int j = 0; j < 3; j++)
            ff << Th[i][j].x << " "<< Th[i][j].y << " " << phi[][Vh(i,j)] << endl;

        ff << Th[i][0].x << " " << Th[i][0].y << " " << phi[][Vh(i,0)] << "\n\n\n"
    }
}
```

We use the finite element numbering, where `Wh(i,j)` is the global index of j^{Th} degrees of freedom of triangle number i.

Then open `gnuplot` and do:

```
set palette rgbformulae 30,31,32
splot "graph.txt" w l pal
```

This works with `P2` and `P1`, but not with `P1nc` because the 3 first degrees of freedom of `P2` or `P2` are on vertices and not with `P1nc`.

2.4 Heat Exchanger

Summary: *Here we shall learn more about geometry input and triangulation files, as well as read and write operations.*

The problem Let $\{C_i\}_{1,2}$, be 2 thermal conductors within an enclosure C_0 (see Fig. 2.5).

Fig. 2.5: Heat exchanger geometry

The first one is held at a constant temperature u_1 the other one has a given thermal conductivity κ_2 3 times larger than the one of C_0.

We assume that the border of enclosure C_0 is held at temperature $20°C$ and that we have waited long enough for thermal equilibrium.

In order to know $u(x)$ at any point x of the domain Ω, we must solve:

$$\nabla \cdot (\kappa \nabla u) = 0 \text{ in } \Omega, \quad u_{|\Gamma} = g$$

where Ω is the interior of C_0 minus the conductor C_1 and Γ is the boundary of Ω, that is $C_0 \cup C_1$.

Here g is any function of x equal to u_i on C_i.

The second equation is a reduced form for:

$$u = u_i \text{ on } C_i, \quad i = 0, 1.$$

The variational formulation for this problem is in the subspace $H_0^1(\Omega) \subset H^1(\Omega)$ of functions which have zero traces on Γ.

$$u - g \in H_0^1(\Omega) : \int_\Omega \nabla u \nabla v = 0 \forall v \in H_0^1(\Omega)$$

Let us assume that C_0 is a circle of radius 5 centered at the origin, C_i are rectangles, C_1 being at the constant temperature $u_1 = 60°C$ (so we can only consider its boundary).

```
// Parameters
int C1=99;
int C2=98; //could be anything such that !=0 and C1!=C2

// Mesh
border C0(t=0., 2.*pi){x=5.*cos(t); y=5.*sin(t);}

border C11(t=0., 1.){x=1.+t; y=3.; label=C1;}
border C12(t=0., 1.){x=2.; y=3.-6.*t; label=C1;}
border C13(t=0., 1.){x=2.-t; y=-3.; label=C1;}
border C14(t=0., 1.){x=1.; y=-3.+6.*t; label=C1;}

border C21(t=0., 1.){x=-2.+t; y=3.; label=C2;}
border C22(t=0., 1.){x=-1.; y=3.-6.*t; label=C2;}
border C23(t=0., 1.){x=-1.-t; y=-3.; label=C2;}
border C24(t=0., 1.){x=-2.; y=-3.+6.*t; label=C2;}

plot(   C0(50) //to see the border of the domain
    + C11(5)+C12(20)+C13(5)+C14(20)
    + C21(-5)+C22(-20)+C23(-5)+C24(-20),
    wait=true, ps="heatexb.eps");

mesh Th=buildmesh(C0(50)
    + C11(5)+C12(20)+C13(5)+C14(20)
    + C21(-5)+C22(-20)+C23(-5)+C24(-20));

plot(Th,wait=1);

// Fespace
fespace Vh(Th, P1);
Vh u, v;
Vh kappa=1 + 2*(x<-1)*(x>-2)*(y<3)*(y>-3);

// Solve
solve a(u, v)
    = int2d(Th)(
          kappa*(
              dx(u)*dx(v)
            + dy(u)*dy(v)
        )
    )
    +on(C0, u=20)
    +on(C1, u=60)
    ;

// Plot
plot(u, wait=true, value=true, fill=true, ps="HeatExchanger.eps");
```

Note the following:

- `C0` is oriented counterclockwise by t, while `C1` is oriented clockwise and `C2` is oriented counterclockwise. This is why `C1` is viewed as a hole by `buildmesh`.
- `C1` and `C2` are built by joining pieces of straight lines. To group them in the same logical unit to input the boundary conditions in a readable way we assigned a label on the boundaries. As said earlier, borders have an internal number corresponding to their order in the program (check it by adding a `cout << C22;` above). This is essential to understand how a mesh can be output to a file and re-read (see below).
- As usual the mesh density is controlled by the number of vertices assigned to each boundary. It is not possible

(a) Heat exchanger mesh

(b) Heat exchanger solution

Fig. 2.6: Heat exchanger

to change the (uniform) distribution of vertices but a piece of boundary can always be cut in two or more parts, for instance `C12` could be replaced by `C121+C122`:

```
// border C12(t=0.,1.){x=2.; y=3.-6.*t; label=C1;}
border C121(t=0.,0.7){x=2.; y=3.-6.*t; label=C1;}
border C122(t=0.7,1.){x=2.; y=3.-6.*t; label=C1;}
...
buildmesh(.../*+ C12(20) */ + C121(12) + C122(8) + ...);
```

Tip: Exercise :

Use the symmetry of the problem with respect to the axes.

Triangulate only one half of the domain, and set Dirichlet conditions on the vertical axis, and Neumann conditions on the horizontal axis.

Writing and reading triangulation files Suppose that at the end of the previous program we added the line

```
savemesh(Th, "condensor.msh");
```

and then later on we write a similar program but we wish to read the mesh from that file. Then this is how the condenser should be computed:

```
// Mesh
mesh Sh = readmesh("condensor.msh");

// Fespace
fespace Wh(Sh, P1);
Wh us, vs;

// Solve
solve b(us, vs)
```

(continues on next page)

(continued from previous page)

```
    = int2d(Sh)(
          dx(us)*dx(vs)
        + dy(us)*dy(vs)
    )
    +on(1, us=0)
    +on(99, us=1)
    +on(98, us=-1)
    ;

// Plot
plot(us);
```

Note that the names of the boundaries are lost but either their internal number (in the case of `C0`) or their label number (for `C1` and `C2`) are kept.

2.5 Acoustics

Summary : *Here we go to grip with ill posed problems and eigenvalue problems*

Pressure variations in air at rest are governed by the wave equation:

$$\frac{\partial^2 u}{\partial t^2} - c^2 \Delta u = 0$$

When the solution wave is monochromatic (and that depends on the boundary and initial conditions), u is of the form $u(x,t) = Re(v(x)e^{ikt})$ where v is a solution of Helmholtz's equation:

$$\begin{aligned} k^2 v + c^2 \Delta v &= 0 \quad \text{in } \Omega \\ \tfrac{\partial v}{\partial n}|_\Gamma &= g \end{aligned}$$

where g is the source.

Note the "+" sign in front of the Laplace operator and that $k > 0$ is real. This sign may make the problem ill posed for some values of $\frac{c}{k}$, a phenomenon called "resonance".

At resonance there are non-zero solutions even when $g = 0$. So the following program may or may not work:

```
// Parameters
real kc2 = 1.;
func g = y*(1.-y);

// Mesh
border a0(t=0., 1.){x=5.; y=1.+2.*t;}
border a1(t=0., 1.){x=5.-2.*t; y=3.;}
border a2(t=0., 1.){x=3.-2.*t; y=3.-2.*t;}
border a3(t=0., 1.){x=1.-t; y=1.;}
border a4(t=0., 1.){x=0.; y=1.-t;}
border a5(t=0., 1.){x=t; y=0.;}
border a6(t=0., 1.){x=1.+4.*t; y=t;}

mesh Th = buildmesh(a0(20) + a1(20) + a2(20)
    + a3(20) + a4(20) + a5(20) + a6(20));

// Fespace
fespace Vh(Th, P1);
Vh u, v;
```

(continues on next page)

(continued from previous page)

```

// Solve
solve sound(u, v)
   = int2d(Th)(
        u*v * kc2
      - dx(u)*dx(v)
      - dy(u)*dy(v)
   )
   - int1d(Th, a4)(
        g * v
   )
   ;

// Plot
plot(u, wait=1, ps="Sound.eps");
```

Results are on Fig. 2.7a. But when $kc2$ is an eigenvalue of the problem, then the solution is not unique:

- if $u_e \neq 0$ is an eigen state, then for any given solution $u + u_e$ is **another** solution.

To find all the u_e one can do the following :

```
// Parameters
real sigma = 20; //value of the shift

// Problem
// OP = A - sigma B ; // The shifted matrix
varf op(u1, u2)
   = int2d(Th)(
        dx(u1)*dx(u2)
      + dy(u1)*dy(u2)
      - sigma* u1*u2
   )
   ;

varf b([u1], [u2])
   = int2d(Th)(
        u1*u2
   )
   ; // No Boundary condition see note \ref{note BC EV}

matrix OP = op(Vh, Vh, solver=Crout, factorize=1);
matrix B = b(Vh, Vh, solver=CG, eps=1e-20);

// Eigen values
int nev=2; // Number of requested eigenvalues near sigma

real[int] ev(nev);  // To store the nev eigenvalue
Vh[int] eV(nev);    // To store the nev eigenvector

int k=EigenValue(OP, B, sym=true, sigma=sigma, value=ev, vector=eV,
   tol=1e-10, maxit=0, ncv=0);

cout << ev(0) << " 2 eigen values " << ev(1) << endl;
v = eV[0];
plot(v, wait=true, ps="eigen.eps");
```

(a) Amplitude of an acoustic signal coming from the left vertical wall.

(b) First eigen state ($\lambda = (k/c)^2 = 19.4256$) close to 20 of eigenvalue problem: $-\Delta\varphi = \lambda\varphi$ and $\frac{\partial\varphi}{\partial n} = 0$ on Γ}

Fig. 2.7: Acoustics

2.6 Thermal Conduction

Summary : *Here we shall learn how to deal with a time dependent parabolic problem. We shall also show how to treat an axisymmetric problem and show also how to deal with a nonlinear problem*

How air cools a plate

We seek the temperature distribution in a plate $(0, Lx) \times (0, Ly) \times (0, Lz)$ of rectangular cross section $\Omega = (0, 6) \times (0, 1)$; the plate is surrounded by air at temperature u_e and initially at temperature $u = u_0 + \frac{x}{L}u_1$. In the plane perpendicular to the plate at $z = Lz/2$, the temperature varies little with the coordinate z; as a first approximation the problem is 2D.

We must solve the temperature equation in Ω in a time interval (0,T).

$$\begin{array}{rcl} \partial_t u - \nabla \cdot (\kappa \nabla u) & = 0 & \text{in } \Omega \times (0, T) \\ u(x, y, 0) & = u_0 + x u_1 & \\ \kappa \frac{\partial u}{\partial n} + \alpha(u - u_e) & = 0 & \text{on } \Gamma \times (0, T) \end{array}$$

Here the diffusion κ will take two values, one below the middle horizontal line and ten times less above, so as to simulate a thermostat.

The term $\alpha(u - u_e)$ accounts for the loss of temperature by convection in air. Mathematically this boundary condition is of Fourier (or Robin, or mixed) type.

The variational formulation is in $L^2(0, T; H^1(\Omega))$; in loose terms and after applying an implicit Euler finite difference approximation in time; we shall seek $u^n(x, y)$ satisfying for all $w \in H^1(\Omega)$:

$$\int_\Omega (\frac{u^n - u^{n-1}}{\delta t} w + \kappa \nabla u^n \nabla w) + \int_\Gamma \alpha(u^n - u_u e) w = 0$$

```
// Parameters
func u0 = 10. + 90.*x/6.;
func k = 1.8*(y<0.5) + 0.2;
```

(continues on next page)

(continued from previous page)

```
real ue = 25.;
real alpha=0.25;
real T=5.;
real dt=0.1 ;

// Mesh
mesh Th = square(30, 5, [6.*x,y]);

// Fespace
fespace Vh(Th, P1);
Vh u=u0, v, uold;

// Problem
problem thermic(u, v)
    = int2d(Th)(
          u*v/dt
        + k*(
              dx(u) * dx(v)
            + dy(u) * dy(v)
        )
    )
    + int1d(Th, 1, 3)(
          alpha*u*v
    )
    - int1d(Th, 1, 3)(
          alpha*ue*v
    )
    - int2d(Th)(
          uold*v/dt
    )
    + on(2, 4, u=u0)
    ;

// Time iterations
ofstream ff("thermic.dat");
for(real t = 0; t < T; t += dt){
    uold = u; //equivalent to u^{n-1} = u^n
    thermic; //here the thermic problem is solved
    ff << u(3., 0.5) << endl;
    plot(u);
}
```

Note: We must separate by hand the bilinear part from the linear one.

Note: The way we store the temperature at point (3, 0.5) for all times in file `thermic.dat`. Should a one dimensional plot be required, the same procedure can be used. For instance to print $x \mapsto \frac{\partial u}{\partial y}(x, 0.9)$ one would do:

```
for(int i = 0; i < 20; i++)
   cout << dy(u)(6.0*i/20.0,0.9) << endl;
```

Results are shown on Fig. 2.8a and Fig. 2.8b.

(a) Temperature at $t = 4.9$.

(b) Decay of temperature versus time at $x = 3, y = 0.5$

Fig. 2.8: Thermal conduction

2.6.1 Axisymmetry: 3D Rod with circular section

Let us now deal with a cylindrical rod instead of a flat plate. For simplicity we take $\kappa = 1$.

In cylindrical coordinates, the Laplace operator becomes (r is the distance to the axis, z is the distance along the axis, θ polar angle in a fixed plane perpendicular to the axis):

$$\Delta u = \frac{1}{r}\partial_r(r\partial_r u) + \frac{1}{r^2}\partial^2_{\theta\theta}u + \partial^2_{zz}.$$

Symmetry implies that we loose the dependence with respect to θ; so the domain Ω is again a rectangle $]0, R[\times]0, |[$. We take the convention of numbering of the edges as in `square()` (1 for the bottom horizontal ...); the problem is now:

$$\begin{aligned}
r\partial_t u - \partial_r(r\partial_r u) - \partial_z(r\partial_z u) &= 0 && \text{in } \Omega \\
u(t=0) &= u_0 + \tfrac{z}{L_z}(u_1 - u) \\
u|_{\Gamma_4} &= u_0 \\
u|_{\Gamma_2} &= u_1 \\
\alpha(u - u_e) + \tfrac{\partial u}{\partial n}|_{\Gamma_1\cup\Gamma_3} &= 0
\end{aligned}$$

Note that the PDE has been multiplied by r.

After discretization in time with an implicit scheme, with time steps `dt`, in the **FreeFEM** syntax r becomes x and z becomes y and the problem is:

```
problem thermaxi(u, v)
    = int2d(Th)(
          (u*v/dt + dx(u)*dx(v) + dy(u)*dy(v))*x
    )
    + int1d(Th, 3)(
          alpha*x*u*v
    )
    - int1d(Th, 3)(
          alpha*x*ue*v
    )
```

(continues on next page)

(continued from previous page)

```
    - int2d(Th)(
          uold*v*x/dt
    )
    + on(2, 4, u=u0);
```

Note: The bilinear form degenerates at $x = 0$. Still one can prove existence and uniqueness for u and because of this degeneracy no boundary conditions need to be imposed on Γ_1.

2.6.2 A Nonlinear Problem : Radiation

Heat loss through radiation is a loss proportional to the absolute temperature to the fourth power (Stefan's Law). This adds to the loss by convection and gives the following boundary condition:

$$\kappa \frac{\partial u}{\partial n} + \alpha(u - u_e) + c[(u + 273)^4 - (u_e + 273)^4] = 0$$

The problem is nonlinear, and must be solved iteratively. If m denotes the iteration index, a semi-linearization of the radiation condition gives

$$\frac{\partial u^{m+1}}{\partial n} + \alpha(u^{m+1} - u_e) + c(u^{m+1} - u_e)(u^m + u_e + 546)((u^m + 273)^2 + (u_e + 273)^2) = 0,$$

because we have the identity $a^4 - b^4 = (a - b)(a + b)(a^2 + b^2)$.

The iterative process will work with $v = u - u_e$.

```
...
// Parameters
real rad=1e-8;
real uek=ue+273;

// Mesh
fespace Vh(Th, P1);
Vh vold, w, v=u0-ue, b;

// Problem
problem thermradia(v, w)
    = int2d(Th)(
          v*w/dt
        + k*(dx(v) * dx(w) + dy(v) * dy(w))
    )
    + int1d(Th, 1, 3)(
          b*v*w
    )
    - int2d(Th)(
          vold*w/dt
    )
    + on(2, 4, v=u0-ue)
    ;

for (real t=0;t<T;t+=dt){
    vold = v;
    for (int m = 0; m < 5; m++){
        b = alpha + rad * (v + 2*uek) * ((v+uek)^2 + uek^2);
```

(continues on next page)

(continued from previous page)

```
        thermradia;
    }
}
vold = v+ue;

// Plot
plot(vold);
```

2.7 Irrotational Fan Blade Flow and Thermal effects

Summary : *Here we will learn how to deal with a multi-physics system of PDEs on a complex geometry, with multiple meshes within one problem. We also learn how to manipulate the region indicator and see how smooth is the projection operator from one mesh to another.*

Incompressible flow

Without viscosity and vorticity incompressible flows have a velocity given by:

$$u = \begin{pmatrix} \frac{\partial \psi}{\partial x_2} \\ -\frac{\partial \psi}{\partial x_1} \end{pmatrix}, \qquad \text{where } \psi \text{ is solution of} \qquad \Delta\psi = 0$$

This equation expresses both incompressibility ($\nabla \cdot u = 0$) and absence of vortex ($\nabla \times u = 0$).

As the fluid slips along the walls, normal velocity is zero, which means that ψ satisfies:

$$\psi \text{ constant on the walls.}$$

One can also prescribe the normal velocity at an artificial boundary, and this translates into non constant Dirichlet data for ψ.

Airfoil

Let us consider a wing profile S in a uniform flow. Infinity will be represented by a large circle C where the flow is assumed to be of uniform velocity; one way to model this problem is to write:

$$\Delta\psi = 0 \text{ in } \Omega, \qquad \psi|_S = 0, \quad \psi|_C = u_\infty y$$

where $\partial\Omega = C \cup S$

The NACA0012 Airfoil

An equation for the upper surface of a NACA0012 (this is a classical wing profile in aerodynamics) is:

$$y = 0.17735\sqrt{x} - 0.075597x - 0.212836x^2 + 0.17363x^3 - 0.06254x^4.$$

```
// Parameters
real S = 99;

// Mesh
border C(t=0., 2.*pi){x=5.*cos(t); y=5.*sin(t);}
border Splus(t=0., 1.){x=t; y=0.17735*sqrt(t) - 0.075597*t
   - 0.212836*(t^2) + 0.17363*(t^3) - 0.06254*(t^4); label=S;}
border Sminus(t=1., 0.){x=t; y=-(0.17735*sqrt(t) - 0.075597*t
   -0.212836*(t^2) + 0.17363*(t^3) - 0.06254*(t^4)); label=S;}
```

(continues on next page)

(a) Zoom around the NACA0012 airfoil showing the streamlines (curve ψ = constant). To obtain such a plot use the interactive graphic command: "+" and p.

(b) Temperature distribution at time T=25 (now the maximum is at 90 instead of 120). Note that an incidence angle has been added here.

Fig. 2.9: The NACA0012 Airfoil

(continued from previous page)

```
mesh Th = buildmesh(C(50) + Splus(70) + Sminus(70));

// Fespace
fespace Vh(Th, P2);
Vh psi, w;

// Solve
solve potential(psi, w)
   = int2d(Th)(
        dx(psi)*dx(w)
      +dy(psi)*dy(w)
   )
   + on(C, psi = y)
   + on(S, psi=0)
   ;

plot(psi, wait=1);
```

A zoom of the streamlines are shown on Fig. 2.9a.

2.7.1 Heat Convection around the airfoil

Now let us assume that the airfoil is hot and that air is there to cool it. Much like in the previous section the heat equation for the temperature v is

$$\partial_t v - \nabla \cdot (\kappa \nabla v) + u \cdot \nabla v = 0, \;\; v(t=0) = v_0, \;\; \frac{\partial v}{\partial n}|_C = 0$$

But now the domain is outside AND inside S and κ takes a different value in air and in steel. Furthermore there is convection of heat by the flow, hence the term $u \cdot \nabla v$ above.

Consider the following, to be plugged at the end of the previous program:

```
// Parameters
real S = 99;
real dt=0.05;
real nbT=50;

// Mesh
border C(t=0., 2.*pi){x=5.*cos(t); y=5.*sin(t);}
border Splus(t=0., 1.){x=t; y=0.17735*sqrt(t) - 0.075597*t
   - 0.212836*(t^2) + 0.17363*(t^3) - 0.06254*(t^4); label=S;}
border Sminus(t=1., 0.){x=t; y=-(0.17735*sqrt(t) - 0.075597*t
   -0.212836*(t^2) + 0.17363*(t^3) - 0.06254*(t^4)); label=S;}
border D(t=0., 2.){x=1.+t; y=0.;} // Added to have a fine mesh at trail
mesh Sh = buildmesh(C(25) + Splus(-90) + Sminus(-90) + D(200));
int steel=Sh(0.5,0).region, air=Sh(-1,0).region;

// Fespaces
fespace Vh(Sh, P2);
Vh psi, w;

fespace Wh(Sh, P1);
Wh v, vv;

fespace W0(Sh,P0);
W0 k=0.01*(region==air)+0.1*(region==steel);
W0 u1=dy(psi)*(region==air), u2=-dx(psi)*(region==air);
Wh vold = 120*(region==steel);

// Problem
int i;
problem thermic(v, vv, init=i, solver=LU)
   = int2d(Sh)(
        v*vv/dt
      + k*(dx(v) * dx(vv) + dy(v) * dy(vv))
      + 10*(u1*dx(v)+u2*dy(v))*vv
   )
   - int2d(Sh)(
        vold*vv/dt
   )
   ;

for(i = 0; i < nbT; i++){
   v = vold;
   thermic;
   plot(v);
}
```

Note: How steel and air are identified by the mesh parameter region which is defined when buildmesh is called and takes an integer value corresponding to each connected component of Ω;

How the convection terms are added without upwinding. Upwinding is necessary when the Pecley number $|u|L/\kappa$ is large (here is a typical length scale), The factor 10 in front of the convection terms is a quick way of multiplying the velocity by 10 (else it is too slow to see something).

The solver is Gauss' LU factorization and when `init` $\neq 0$ the LU decomposition is reused so it is much faster after the first iteration.

2.8 Pure Convection : The Rotating Hill

Summary: *Here we will present two methods for upwinding for the simplest convection problem. We will learn about Characteristics-Galerkin and Discontinuous-Galerkin Finite Element Methods.*

Let Ω be the unit disk centered at $(0,0)$; consider the rotation vector field

$$\mathbf{u} = [u1, u2], \qquad u_1 = y, \quad u_2 = -x$$

Pure convection by $\mathbf{u}$ is

$$\begin{aligned} \partial_t c + \mathbf{u}.\nabla c &= 0 && \text{in } \Omega \times (0,T) \\ c(t=0) &= c^0 && \text{in } \Omega. \end{aligned}$$

The exact solution $c(x_t, t)$ at time t en point x_t is given by:

$$c(x_t, t) = c^0(x, 0)$$

where x_t is the particle path in the flow starting at point x at time 0. So x_t are solutions of

$$\dot{x_t} = u(x_t), \qquad x_{t=0} = x, \quad \text{where} \quad \dot{x_t} = \frac{\mathrm{d}(t \mapsto x_t)}{\mathrm{d}t}$$

The ODE are reversible and we want the solution at point x at time t (not at point x_t) the initial point is x_{-t}, and we have

$$c(x, t) = c^0(x_{-t}, 0)$$

The game consists in solving the equation until $T = 2\pi$, that is for a full revolution and to compare the final solution with the initial one; they should be equal.

2.8.1 Solution by a Characteristics-Galerkin Method

In **FreeFEM** there is an operator called `convect([u1,u2], dt, c)` which compute $c \circ X$ with X is the convect field defined by $X(x) = x_{dt}$ and where x_τ is particule path in the steady state velocity field $\mathbf{u} = [u1, u2]$ starting at point x at time $\tau = 0$, so x_τ is solution of the following ODE:

$$\dot{x}_\tau = u(x_\tau), \mathbf{x}_{\tau=0} = x.$$

When $\mathbf{u}$ is piecewise constant; this is possible because x_τ is then a polygonal curve which can be computed exactly and the solution exists always when $\mathbf{u}$ is divergence free; convect returns $c(x_{df}) = C \circ X$.

```
// Parameters
real dt = 0.17;

// Mesh
border C(t=0., 2.*pi) {x=cos(t); y=sin(t);};
mesh Th = buildmesh(C(100));

// Fespace
fespace Uh(Th, P1);
Uh cold, c = exp(-10*((x-0.3)^2 +(y-0.3)^2));
Uh u1 = y, u2 = -x;

// Time loop
```

(continues on next page)

(continued from previous page)

```
real t = 0;
for (int m = 0; m < 2.*pi/dt; m++){
    t += dt;
    cold = c;
    c = convect([u1, u2], -dt, cold);
    plot(c, cmm=" t="+t +", min="+c[].min+", max="+c[].max);
}
```

Note: 3D plots can be done by adding the qualifyer `dim=3` to the plot instruction.

The method is very powerful but has two limitations:

- it is not conservative
- it may diverge in rare cases when $|\mathbf{u}|$ is too small due to quadrature error.

2.8.2 Solution by Discontinuous-Galerkin FEM

Discontinuous Galerkin methods take advantage of the discontinuities of c at the edges to build upwinding. There are may formulations possible. We shall implement here the so-called dual-P_1^{DC} formulation (see [ERN2006]):

$$\int_\Omega (\frac{c^{n+1}-c^n}{\delta t} + u\cdot\nabla c)w + \int_E (\alpha|n\cdot u| - \frac{1}{2}n\cdot u)[c]w = \int_{E_\Gamma^-} |n\cdot u|cw \quad \forall w$$

where E is the set of inner edges and E_Γ^- is the set of boundary edges where $u\cdot n < 0$ (in our case there is no such edges). Finally $[c]$ is the jump of c across an edge with the convention that c^+ refers to the value on the right of the oriented edge.

```
// Parameters
real al=0.5;
real dt = 0.05;

// Mesh
border C(t=0., 2.*pi) {x=cos(t); y=sin(t);};
mesh Th = buildmesh(C(100));

// Fespace
fespace Vh(Th,P1dc);
Vh w, ccold, v1 = y, v2 = -x, cc = exp(-10*((x-0.3)^2 +(y-0.3)^2));

// Macro
macro n() (N.x*v1 + N.y*v2) // Macro without parameter

// Problem
problem Adual(cc, w)
    = int2d(Th)(
          (cc/dt+(v1*dx(cc)+v2*dy(cc)))*w
    )
    + intalledges(Th)(
          (1-nTonEdge)*w*(al*abs(n)-n/2)*jump(cc)
    )
    - int2d(Th)(
          ccold*w/dt
    )
```

(continues on next page)

(a) The rotating hill after one revolution with Characteristics-Galerkin

(b) The rotating hill after one revolution with Discontinuous P_1 Galerkin

Fig. 2.10: Rotating hill

(continued from previous page)

```
    ;

// Time iterations
for (real t = 0.; t < 2.*pi; t += dt){
    ccold = cc;
    Adual;
    plot(cc, fill=1, cmm="t="+t+", min="+cc[].min+", max="+ cc[].max);
}

// Plot
real [int] viso = [-0.2, -0.1, 0., 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1., 1.
↪1];
plot(cc, wait=1, fill=1, ps="ConvectCG.eps", viso=viso);
plot(cc, wait=1, fill=1, ps="ConvectDG.eps", viso=viso);
```

Note: New keywords: `intalledges` to integrate on all edges of all triangles

$$\texttt{intalledges(Th)} \equiv \sum_{T\in\texttt{Th}} \int_{\partial T}$$

(so all internal edges are see two times), nTonEdge which is one if the triangle has a boundary edge and two otherwise, `jump` to implement $[c]$.

Results of both methods are shown on Fig. 2.10a nad Fig. 2.10b with identical levels for the level line; this is done with the plot-modifier viso.

Notice also the macro where the parameter **u** is not used (but the syntax needs one) and which ends with a `//`; it simply replaces the name `n` by `(N.x*v1+N.y*v2)`. As easily guessed `N.x,N.y` is the normal to the edge.

Now if you think that DG is too slow try this:

```
// Parameters
real al=0.5;
real dt = 0.05;

// Mesh
border C(t=0., 2.*pi) {x=cos(t); y=sin(t);};
mesh Th = buildmesh(C(100));

// Fespace
fespace Vh(Th,P1dc);
Vh w, ccold, v1 = y, v2 = -x, cc = exp(-10*((x-0.3)^2 +(y-0.3)^2));
Vh rhs=0;

// Macro
macro n() (N.x*v1 + N.y*v2) // Macro without parameter

// Problem
real t = 0.;

varf vAdual (cc, w)
    = int2d(Th)(
          (cc/dt+(v1*dx(cc)+v2*dy(cc)))*w
    )
    + intalledges(Th)(
          (1-nTonEdge)*w*(al*abs(n)-n/2)*jump(cc)
    )
    ;

varf vBdual (cc, w)
    = - int2d(Th)(
          ccold*w/dt
    )
    ;

matrix AA = vAdual(Vh, Vh);
matrix BB = vBdual(Vh, Vh);
set (AA, init=t, solver=sparsesolver);

// Time iterations
for (t = 0.; t < 2.*pi; t += dt){
    ccold = cc;
    rhs[] = BB * ccold[];
    cc[] = AA^-1 * rhs[];
    plot(cc, fill=1, cmm="t="+t+", min="+cc[].min+", max="+ cc[].max);
}
```

Notice the new keyword `set` to specify a solver in this framework; the modifier `init` is used to tell the solver that the matrix has not changed (`init=true`), and the name parameter are the same that in problem definition (see *Problem*)

2.8.3 Finite Volume Methods can also be handled with FreeFEM but it requires programming.

For instance the $P_0 - P_1$ Finite Volume Method of Dervieux *et al* associates to each P_0 function c^1 a P_0 function c^0 with constant value around each vertex q^i equal to $c^1(q^i)$ on the cell σ_i made by all the medians of all triangles having q^i as vertex.

Then upwinding is done by taking left or right values at the median:

$$\int_{\sigma_i} \frac{1}{\delta t}(c^{1^{n+1}} - c^{1^n}) + \int_{\partial\sigma_i} u \cdot nc^- = 0, \forall i$$

It can be programmed as :

```
load "mat_dervieux"; //External module in C++ must be loaded

// Parameters
real dt = 0.025;

// Mesh
border a(t=0., 2.*pi){x=cos(t); y=sin(t);}
mesh th = buildmesh(a(100));

// Fespace
fespace Vh(th,P1);
Vh vh, vold, u1=y, u2=-x;
Vh v=exp(-10*((x-0.3)^2 +(y-0.3)^2)), vWall=0, rhs=0;

// Problem
//qf1pTlump means mass lumping is used
problem FVM(v,vh) = int2d(th,qft=qf1pTlump)(v*vh/dt)
    - int2d(th,qft=qf1pTlump)(vold*vh/dt)
    + int1d(th,a)(((u1*N.x+u2*N.y)<0)*(u1*N.x+u2*N.y)*vWall*vh)
+ rhs[] ;

matrix A;
MatUpWind0(A, th, vold, [u1, u2]);

// Time loop
for (int t = 0; t < 2.*pi ; t += dt){
    vold = v;
    rhs[] = A * vold[];
    FVM;
    plot(v, wait=0);
}
```

the "mass lumping" parameter forces a quadrature formula with Gauss points at the vertices so as to make the mass matrix diagonal; the linear system solved by a conjugate gradient method for instance will then converge in one or two iterations.

The right hand side `rhs` is computed by an external C++ function `MatUpWind0(...)` which is programmed as :

```
// Computes matrix a on a triangle for the Dervieux FVM
int fvmP1P0(double q[3][2], // the 3 vertices of a triangle T
    double u[2], // convection velocity on T
    double c[3], // the P1 function on T
    double a[3][3],// output matrix
    double where[3]) // where>0 means we're on the boundary
{
    for (int i = 0; i < 3; i++)
        for(int j = 0; j < 3; j++) a[i][j] = 0;

    for(int i = 0; i < 3; i++){
        int ip = (i+1)%3, ipp = (ip+1)%3;
        double unL = -((q[ip][1] + q[i][1] - 2*q[ipp][1])*u[0]
            - (q[ip][0] + q[i][0] - 2*q[ipp][0])*u[1])/6.;
```

(continues on next page)

(continued from previous page)

```
        if (unL > 0){
            a[i][i] += unL;
            a[ip][i] -=unL;
        }
        else{
            a[i][ip] += unL;
            a[ip][ip] -=unL;
        }
        if (where[i] && where[ip]){ // this is a boundary edge
            unL = ((q[ip][1] - q[i][1])*u[0] - (q[ip][0] - q[i][0])*u[1])/2;
            if (unL > 0){
                a[i][i] += unL;
                a[ip][ip] += unL;
            }
        }
    }
    return 1;
}
```

It must be inserted into a larger .cpp file, shown in Appendix A, which is the load module linked to **FreeFEM**.

2.9 The System of elasticity

Elasticity

Solid objects deform under the action of applied forces:

a point in the solid, originally at (x, y, z) will come to (X, Y, Z) after some time; the vector $\mathbf{u} = (u_1, u_2, u_3) = (X - x, Y - y, Z - z)$ is called the displacement. When the displacement is small and the solid is elastic, Hooke's law gives a relationship between the stress tensor $\sigma(u) = (\sigma_{ij}(u))$ and the strain tensor $\epsilon(u) = \epsilon_{ij}(u)$

$$\sigma_{ij}(u) = \lambda\delta_{ij}\nabla.\mathbf{u} + 2\mu\epsilon_{ij}(u),$$

where the Kronecker symbol $\delta_{ij} = 1$ if $i = j$, 0 otherwise, with

$$\epsilon_{ij}(u) = \frac{1}{2}(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}),$$

and where λ, μ are two constants that describe the mechanical properties of the solid, and are themselves related to the better known constants E, Young's modulus, and ν, Poisson's ratio:

$$\mu = \frac{E}{2(1+\nu)}, \quad \lambda = \frac{E\nu}{(1+\nu)(1-2\nu)}.$$

Lamé's system

Let us consider a beam with axis Oz and with perpendicular section Ω. The components along x and y of the strain $\mathbf{u}(x)$ in a section Ω subject to forces $\mathbf{f}$ perpendicular to the axis are governed by:

$$-\mu\Delta\mathbf{u} - (\mu + \lambda)\nabla(\nabla.\mathbf{u}) = \mathbf{f} \text{ in } \Omega,$$

where λ ,μ are the Lamé coefficients introduced above.

Remark, we do not use this equation because the associated variational form does not give the right boundary condition, we simply use:

$$-div(\sigma) = \mathbf{f} \quad \text{in } \Omega$$

where the corresponding variational form is:

$$\int_\Omega \sigma(u) : \epsilon(\mathbf{v})\ dx - \int_\Omega \mathbf{v} f\ dx = 0;$$

where : denotes the tensor scalar product, i.e. $a : b = \sum_{i,j} a_{ij} b_{ij}$.

So the variational form can be written as :

$$\int_\Omega \lambda \nabla.u \nabla.v + 2\mu\epsilon(\mathbf{u}) : \epsilon(\mathbf{v})\ dx - \int_\Omega \mathbf{v} f\ dx = 0;$$

Tip: Consider an elastic plate with the undeformed rectangle shape $[0, 20] \times [-1, 1]$.

The body force is the gravity force $\mathbf{f}$ and the boundary force $\mathbf{g}$ is zero on lower, upper and right sides. The left vertical side of the beam is fixed. The boundary conditions are:

$$\begin{array}{rll} \sigma.\mathbf{n} & = \mathbf{g} \quad = 0 & \text{on } \Gamma_1, \Gamma_4, \Gamma_3, \\ \mathbf{u} & = \mathbf{0} & \text{on } \Gamma_2 \end{array}$$

Here $\mathbf{u} = (u, v)$ has two components.

The above two equations are strongly coupled by their mixed derivatives, and thus any iterative solution on each of the components is risky. One should rather use **FreeFEM**'s system approach and write:

```
// Parameters
real E = 21e5;
real nu = 0.28;

real f = -1;

// Mesh
mesh Th = square(10, 10, [20*x,2*y-1]);

// Fespace
fespace Vh(Th, P2);
Vh u, v;
Vh uu, vv;

// Macro
real sqrt2=sqrt(2.);
macro epsilon(u1,u2) [dx(u1),dy(u2),(dy(u1)+dx(u2))/sqrt2] //
// The sqrt2 is because we want: epsilon(u1,u2)'* epsilon(v1,v2) = epsilon(u):␣
↪epsilon(v)
macro div(u,v) ( dx(u)+dy(v) ) //

// Problem
real mu= E/(2*(1+nu));
real lambda = E*nu/((1+nu)*(1-2*nu));

solve lame([u, v], [uu, vv])
   = int2d(Th)(
        lambda * div(u, v) * div(uu, vv)
      + 2.*mu * ( epsilon(u,v)' * epsilon(uu,vv) )
   )
   - int2d(Th)(
        f*vv
```

(continues on next page)

(a) Vector **(b)** Deformation

Fig. 2.11: Elasticity

(continued from previous page)

```
   )
   + on(4, u=0, v=0)
   ;

// Plot
real coef=100;
plot([u, v], wait=1, ps="lamevect.eps", coef=coef);

// Move mesh
mesh th1 = movemesh(Th, [x+u*coef, y+v*coef]);
plot(th1,wait=1,ps="lamedeform.eps");

// Output
real dxmin = u[].min;
real dymin = v[].min;

cout << " - dep. max x = "<< dxmin << " y=" << dymin << endl;
cout << "   dep. (20, 0) = " << u(20, 0) << " " << v(20, 0) << endl;
```

The output is:

```
-- square mesh : nb vertices  =121 ,  nb triangles = 200 ,  nb boundary edges 40
-- Solve :           min -0.00174137  max 0.00174105
         min -0.0263154  max 1.47016e-29
- dep.  max   x = -0.00174137 y=-0.0263154
   dep.  (20,0)  = -1.8096e-07 -0.0263154
times: compile 0.010219s, execution 1.5827s
```

Solution of Lamé's equations for elasticity for a 2D beam deflected by its own weight and clamped by its left vertical side is shown Fig. 2.11a and Fig. 2.11b. Result are shown with a amplification factor equal to 100. The size of the arrow is automatically bound, but the color gives the real length.

2.10 The System of Stokes for Fluids

In the case of a flow invariant with respect to the third coordinate (two-dimensional flow), flows at low Reynolds number (for instance micro-organisms) satisfy,

$$\begin{aligned} -\Delta \mathbf{u} + \nabla p &= 0 \\ \nabla \cdot \mathbf{u} &= 0 \end{aligned}$$

where $\mathbf{u} = (u_1, u_2)$ is the fluid velocity and p its pressure.

The driven cavity is a standard test. It is a box full of liquid with its lid moving horizontally at speed one. The pressure

and the velocity must be discretized in compatible fintie element spaces for the LBB conditions to be satisfied:

$$\sup_{p \in P_h} \frac{(\mathbf{u}, \nabla p)}{|p|} \geq \beta |\mathbf{u}| \quad \forall \mathbf{u} \in U_h$$

```
// Parameters
int nn = 30;

// Mesh
mesh Th = square(nn, nn);

// Fespace
fespace Uh(Th, P1b);
Uh u, v;
Uh uu, vv;

fespace Ph(Th, P1);
Ph p, pp;

// Problem
solve stokes ([u, v, p], [uu, vv, pp])
    = int2d(Th)(
          dx(u)*dx(uu)
        + dy(u)*dy(uu)
        + dx(v)*dx(vv)
        + dy(v)*dy(vv)
        + dx(p)*uu
        + dy(p)*vv
        + pp*(dx(u) + dy(v))
        - 1e-10*p*pp
    )
    + on(1, 2, 4, u=0, v=0)
    + on(3, u=1, v=0)
    ;

// Plot
plot([u, v], p, wait=1);
```

Note: We add a stabilization term $-\mathbf{10e} - \mathbf{10} * \mathbf{p} * \mathbf{pp}$ to fix the constant part of the pressure.

Results are shown on Fig. 2.12.

2.11 A projection algorithm for the Navier-Stokes equations

Summary : *Fluid flows require good algorithms and good triangultions. We show here an example of a complex algorithm and or first example of mesh adaptation.*

An incompressible viscous fluid satisfies:

$$\begin{aligned} \partial_t \mathbf{u} + \mathbf{u} \cdot \nabla \mathbf{u} + \nabla p - \nu \Delta \mathbf{u} &= 0 && \text{in } \Omega \times]0, T[\\ \nabla \cdot \mathbf{u} &= 0 && \text{in } \Omega \times]0, T[\\ \mathbf{u}|_{t=0} &= \mathbf{u}^0 \\ \mathbf{u}|_\Gamma &= \mathbf{u}_\Gamma \end{aligned}$$

Fig. 2.12: Solution of Stokes' equations for the driven cavity problem, showing the velocity field and the pressure level lines.

A possible algorithm, proposed by Chorin, is:

$$\begin{array}{rcl} \frac{1}{\delta t}[\mathbf{u}^{m+1} - \mathbf{u}^m o\mathbf{X}^m] + \nabla p^m - \nu\Delta\mathbf{u}^m & = & 0 \\ \mathbf{u}|_\Gamma & = & \mathbf{u}_\Gamma \\ \nu\partial_n\mathbf{u}|_{\Gamma_{out}} & = & 0 \end{array}$$

$$\begin{array}{rll} -\Delta p^{m+1} & = -\nabla\cdot\mathbf{u}^m o\mathbf{X}^m & \\ \partial_n p^{m+1} & = 0 & \text{on } \Gamma \\ p^{m+1} & = 0 & \text{on } \Gamma_{out} \end{array}$$

where $\mathbf{u}o\mathbf{X}(x) = \mathbf{u}(\mathbf{x} - \mathbf{u}(\mathbf{x})\delta t)$ since $\partial_t\mathbf{u} + \mathbf{u}\cdot\nabla\mathbf{u}$ is approximated by the method of characteristics, as in the previous section.

We use the Chorin's algorithm with free boundary condition at outlet (i.e. $p = 0, \nu\partial_n u = 0$), to compute a correction, q, to the pressure.

$$\begin{array}{rl} -\Delta q & = \nabla\cdot\mathbf{u} \\ q & = 0 \text{ on } \quad \Gamma_{out} \end{array}$$

and define

$$\begin{array}{rcl} \mathbf{u}^{m+1} & = & \tilde{\mathbf{u}} + P\nabla q\delta t \\ p^{m+1} & = & p^m - q \end{array}$$

where $\tilde{\mathbf{u}}$ is the $(\mathbf{u}^{m+1}, v^{m+1})$ of Chorin's algorithm, and where P is the L^2 projection with mass lumping (a sparse matrix).

The backward facing step

The geometry is that of a channel with a backward facing step so that the inflow section is smaller than the outflow section. This geometry produces a fluid recirculation zone that must be captured correctly.

This can only be done if the triangulation is sufficiently fine, or well adapted to the flow.

Note: There is a technical difficulty in the example: the output B.C. Here we put $p = 0$ and $\nu\partial_n u = 0$.

```
// Parameters
verbosity = 0;
int nn = 1;
real nu = 0.0025;
real dt = 0.2;
real epsv = 1e-6;
real epsu = 1e-6;
real epsp = 1e-6;

// Mesh
border a0(t=1, 0){x=-2; y=t; label=1;}
border a1(t=-2, 0){x=t; y=0; label=2;}
border a2(t=0, -0.5){x=0; y=t; label=2;}
border a3(t=0, 1){x=18*t^1.2; y=-0.5; label=2;}
border a4(t=-0.5, 1){x=18; y=t; label=3;}
border a5(t=1, 0){x=-2+20*t; y=1; label=4;}

mesh Th = buildmesh(a0(3*nn) + a1(20*nn) + a2(10*nn) + a3(150*nn) + a4(5*nn) +
↪a5(100*nn));
plot(Th);

// Fespace
```

(continues on next page)

(continued from previous page)

```
fespace Vh(Th, P1);
Vh w;
Vh u=0, v=0;
Vh p=0;
Vh q=0;

// Definition of Matrix dtMx and dtMy
matrix dtM1x, dtM1y;

// Macro
macro BuildMat()
{   /* for memory managenemt */
    varf vM(unused, v) = int2d(Th)(v);
    varf vdx(u, v) = int2d(Th)(v*dx(u)*dt);
    varf vdy(u, v) = int2d(Th)(v*dy(u)*dt);

    real[int] Mlump = vM(0, Vh);
    real[int] one(Vh.ndof); one = 1;
    real[int] M1 = one ./ Mlump;
    matrix dM1 = M1;
    matrix Mdx = vdx(Vh, Vh);
    matrix Mdy = vdy(Vh, Vh);
    dtM1x = dM1*Mdx;
    dtM1y = dM1*Mdy;
} //

// Build matrices
BuildMat

// Time iterations
real err = 1.;
real outflux = 1.;
for(int n = 0; n < 300; n++){
    // Update
    Vh uold=u, vold=v, pold=p;

    // Solve
    solve pb4u (u, w, init=n, solver=CG, eps=epsu)
        = int2d(Th)(
              u*w/dt
            + nu*(dx(u)*dx(w) + dy(u)*dy(w))
        )
        -int2d(Th)(
                convect([uold, vold], -dt, uold)/dt*w
            - dx(p)*w
        )
        + on(1, u=4*y*(1-y))
        + on(2, 4, u=0)
        ;

    plot(u);

    solve pb4v (v, w, init=n, solver=CG, eps=epsv)
        = int2d(Th)(
              v*w/dt
            + nu*(dx(v)*dx(w) + dy(v)*dy(w))
        )
```

(continues on next page)

(continued from previous page)

```
        -int2d(Th)(
                convect([uold,vold],-dt,vold)/dt*w
            - dy(p)*w
        )
        +on(1, 2, 3, 4, v=0)
        ;

    solve pb4p (q, w, solver=CG, init=n, eps=epsp)
        = int2d(Th)(
              dx(q)*dx(w)+dy(q)*dy(w)
        )
        - int2d(Th)(
              (dx(u)+ dy(v))*w/dt
        )
        + on(3, q=0)
        ;

    //to have absolute epsilon in CG algorithm.
    epsv = -abs(epsv);
    epsu = -abs(epsu);
    epsp = -abs(epsp);

    p = pold-q;
    u[] += dtM1x*q[];
    v[] += dtM1y*q[];

    // Mesh adaptation
    if (n%50 == 49){
        Th = adaptmesh(Th, [u, v], q, err=0.04, nbvx=100000);
        plot(Th, wait=true);
        BuildMat // Rebuild mat.
    }

    // Error & Outflux
    err = sqrt(int2d(Th)(square(u-uold)+square(v-vold))/Th.area);
    outflux = int1d(Th)([u,v]'*[N.x,N.y]);
    cout << " iter " << n << " Err L2 = " << err << " outflux = " << outflux << endl;
    if(err < 1e-3) break;
}

// Verification
assert(abs(outflux)< 2e-3);

// Plot
plot(p, wait=1, ps="NSprojP.eps");
plot(u, wait=1, ps="NSprojU.eps");
```

Rannacher's projection algorithm: result on an adapted mesh, Fig. 2.13a, showing the pressure, Fig. 2.13b, and the horizontal velocity Fig. 2.13c for a Reynolds number of 400 where mesh adaptation is done after 50 iterations on the first mesh.

(a) Adapted mesh

(b) Pressure

(c) Velocity

Fig. 2.13: Navier-Stokes projection

2.12 Newton Method for the Steady Navier-Stokes equations

The problem is find the velocity field $\mathbf{u} = (u_i)_{i=1}^d$ and the pressure p of a Flow satisfying in the domain $\Omega \subset \mathbb{R}^d (d = 2, 3)$:

$$\begin{aligned} (\mathbf{u} \cdot \nabla)\mathbf{u} - \nu \Delta \mathbf{u} + \nabla p &= 0 \\ \nabla \cdot \mathbf{u} &= 0 \end{aligned}$$

where ν is the viscosity of the fluid, $\nabla = (\partial_i)_{i=1}^d$, the dot product is $\cdot$, and $\Delta = \nabla \cdot \nabla$ with the same boundary conditions ($\mathbf{u}$ is given on Γ).

The weak form is find $\mathbf{u}, p$ such that for $\forall \mathbf{v}$ (zero on Γ), and $\forall q$:

$$\int_\Omega ((\mathbf{u} \cdot \nabla)\mathbf{u}).\mathbf{v} + \nu \nabla \mathbf{u} : \nabla \mathbf{v} - p \nabla \cdot \mathbf{v} - q \nabla \cdot \mathbf{u} = 0$$

The Newton Algorithm to solve nonlinear problem is:

Find $u \in V$ such that $F(u) = 0$ where $F : V \mapsto V$.

1. choose $u_0 \in \mathbb{R}^n$, ;
2. for ($i = 0$; $i <$ niter; $i = i + 1$)
    1. solve $DF(u_i)w_i = F(u_i)$;
    2. $u_{i+1} = u_i - w_i$;

break $||w_i|| < \varepsilon$.

Where $DF(u)$ is the differential of F at point u, this is a linear application such that:

$$F(u + \delta) = F(u) + DF(u)\delta + o(\delta)$$

For Navier Stokes, F and DF are:

$$\begin{aligned} F(\mathbf{u}, p) &= \int_\Omega ((\mathbf{u} \cdot \nabla)\mathbf{u}).\mathbf{v} + \nu \nabla \mathbf{u} : \nabla \mathbf{v} - p \nabla \cdot \mathbf{v} - q \nabla \cdot \mathbf{u} \\ DF(\mathbf{u}, p)(\delta \mathbf{u}, \delta p) &= \int_\Omega ((\delta \mathbf{u} \cdot \nabla)\mathbf{u}).\mathbf{v} + ((\mathbf{u} \cdot \nabla)\delta \mathbf{u}).\mathbf{v} \\ &+ \nu \nabla \delta \mathbf{u} : \nabla \mathbf{v} - \delta p \nabla \cdot \mathbf{v} - q \nabla \cdot \delta \mathbf{u} \end{aligned}$$

So the Newton algorithm become:

```
// Parameters
real R = 5.;
real L = 15.;

real nu = 1./50.;
real nufinal = 1/200.;
real cnu = 0.5;

real eps = 1e-6;

verbosity = 0;

// Mesh
border cc(t=0, 2*pi){x=cos(t)/2.; y=sin(t)/2.; label=1;}
border ce(t=pi/2, 3*pi/2){x=cos(t)*R; y=sin(t)*R; label=1;}
border beb(tt=0, 1){real t=tt^1.2; x=t*L; y=-R; label=1;}
border beu(tt=1, 0){real t=tt^1.2; x=t*L; y=R; label=1;}
border beo(t=-R, R){x=L; y=t; label=0;}
border bei(t=-R/4, R/4){x=L/2; y=t; label=0;}
mesh Th = buildmesh(cc(-50) + ce(30) + beb(20) + beu(20) + beo(10) + bei(10));
plot(Th);

//bounding box for the plot
func bb = [[-1,-2],[4,2]];

// Fespace
fespace Xh(Th, P2);
Xh u1, u2;
Xh v1,v2;
Xh du1,du2;
Xh u1p,u2p;

fespace Mh(Th,P1);
Mh p;
Mh q;
Mh dp;
Mh pp;

// Macro
macro Grad(u1,u2) [dx(u1), dy(u1), dx(u2),dy(u2)] //
macro UgradV(u1,u2,v1,v2) [[u1,u2]'*[dx(v1),dy(v1)],
                           [u1,u2]'*[dx(v2),dy(v2)]] //
macro div(u1,u2) (dx(u1) + dy(u2)) //

// Initialization
u1 = (x^2+y^2) > 2;
u2 = 0;

// Viscosity loop
while(1){
    int n;
    real err=0;
    // Newton loop
    for (n = 0; n < 15; n++){
        // Newton
        solve Oseen ([du1, du2, dp], [v1, v2, q])
            = int2d(Th)(
```

(continues on next page)

(continued from previous page)

```
                    nu * (Grad(du1,du2)' * Grad(v1,v2))
                + UgradV(du1,du2, u1, u2)' * [v1,v2]
                + UgradV( u1, u2,du1,du2)' * [v1,v2]
                - div(du1,du2) * q
                - div(v1,v2) * dp
                - 1e-8*dp*q //stabilization term
            )
            - int2d(Th) (
                    nu * (Grad(u1,u2)' * Grad(v1,v2))
                + UgradV(u1,u2, u1, u2)' * [v1,v2]
                - div(u1,u2) * q
                - div(v1,v2) * p
            )
            + on(1, du1=0, du2=0)
            ;

        u1[] -= du1[];
        u2[] -= du2[];
        p[] -= dp[];

        real Lu1=u1[].linfty, Lu2=u2[].linfty, Lp=p[].linfty;
        err = du1[].linfty/Lu1 + du2[].linfty/Lu2 + dp[].linfty/Lp;

        cout << n << " err = " << err << " " << eps << " rey = " << 1./nu << endl;
        if(err < eps) break; //converge
        if( n>3 && err > 10.) break; //blowup
    }

    if(err < eps){  //converge: decrease $\nu$ (more difficult)
        // Plot
        plot([u1, u2], p, wait=1, cmm=" rey = " + 1./nu , coef=0.3, bb=bb);

        // Change nu
        if( nu == nufinal) break;
        if( n < 4) cnu = cnu^1.5; //fast converge => change faster
        nu = max(nufinal, nu* cnu); //new viscosity

        // Update
        u1p = u1;
        u2p = u2;
        pp = p;
    }
    else{   //blowup: increase $\nu$ (more simple)
        assert(cnu< 0.95); //the method finally blowup

        // Recover nu
        nu = nu/cnu;
        cnu= cnu^(1./1.5); //no conv. => change lower
        nu = nu* cnu; //new viscosity
        cout << " restart nu = " << nu << " Rey = " << 1./nu << " (cnu = " << cnu <<
↪" ) \n";

        // Recover a correct solution
        u1 = u1p;
        u2 = u2p;
        p = pp;
    }
```

(continues on next page)

(a) Mesh **(b)** Velocity and pressure

Fig. 2.14: Naver-Stokes newton

(continued from previous page)

```
}
```

Note: We use a trick to make continuation on the viscosity ν, because the Newton method blowup owe start with the final viscosity ν.

ν is gradually increased to the desired value.

2.13 A Large Fluid Problem

A friend of one of us in Auroville-India was building a ramp to access an air conditioned room. As I was visiting the construction site he told me that he expected to cool air escaping by the door to the room to slide down the ramp and refrigerate the feet of the coming visitors. I told him "no way" and decided to check numerically.

The fluid velocity and pressure are solution of the Navier-Stokes equations with varying density function of the temperature.

The geometry is trapezoidal with prescribed inflow made of cool air at the bottom and warm air above and so are the initial conditions; there is free outflow, slip velocity at the top (artificial) boundary and no-slip at the bottom. However the Navier-Stokes cum temperature equations have a RANS $k-\epsilon$ model and a Boussinesq approximation for the buoyancy. This comes to :

$$
\begin{aligned}
\partial_t\theta + u\nabla\theta - \nabla\cdot(\kappa_T^m\nabla\theta) &= 0\\
\partial_t u + u\nabla u - \nabla\cdot(\mu_T\nabla u) + \nabla p + e(\theta-\theta_0)\vec{e}_2 &= 0\\
\nabla\cdot u &= 0\\
\mu_T &= c_\mu\frac{k^2}{\epsilon}\\
\kappa_T &= \kappa\mu_T\\
\partial_t k + u\nabla k + \epsilon - \nabla\cdot(\mu_T\nabla k) &= \frac{\mu_T}{2}|\nabla u + \nabla u^T|^2\\
\partial_t\epsilon + u\nabla\epsilon + c_2\frac{\epsilon^2}{k} - \frac{c_\epsilon}{c_\mu}\nabla\cdot(\mu_T\nabla\epsilon) &= \frac{c_1}{2}k|\nabla u + \nabla u^T|^2
\end{aligned}
$$

We use a time discretization which preserves positivity and uses the method of characteristics ($X^m(x) \approx x - u^m(x)\delta t$)

$$
\begin{aligned}
\frac{1}{\delta t}(\theta^{m+1} - \theta^m \circ X^m) - \nabla \cdot (\kappa_T^m \nabla \theta^{m+1}) &= 0 \\
\frac{1}{\delta t}(u^{m+1} - u^m \circ X^m) - \nabla \cdot (\mu_T^m \nabla u^{m+1}) + \nabla p^{m+1} + e(\theta^{m+1} - \theta_0)\vec{e}_2 &= 0 \\
\nabla \cdot u^{m+1} &= 0 \\
\frac{1}{\delta t}(k^{m+1} - k^m \circ X^m) + k^{m+1}\frac{\epsilon^m}{k^m} - \nabla \cdot (\mu_T^m \nabla k^{m+1}) &= \frac{\mu_T^m}{2}|\nabla u^m + \nabla u^{mT}|^2 \\
\frac{1}{\delta t}(\epsilon^{m+1} - \epsilon^m \circ X^m) + c_2 \epsilon^{m+1}\frac{\epsilon^m}{k^m} - \frac{c_\epsilon}{c_\mu}\nabla(\mu_T^m \nabla \epsilon^{m+1}) &= \frac{c_1}{2}k^m|\nabla u^m + \nabla u^{mT}|^2 \\
\mu_T^{m+1} &= c_\mu \frac{k^{m+1\,2}}{\epsilon^{m+1}} \\
\kappa_T^{m+1} &= \kappa \mu_T^{m+1}
\end{aligned}
$$

In variational form and with appropriated boundary conditions the problem is :

```
load "iovtk"

verbosity=0;

// Parameters
int nn = 15;
int nnPlus = 5;
real l = 1.;
real L = 15.;
real hSlope = 0.1;
real H = 6.;
real h = 0.5;

real reylnods =500;
real beta = 0.01;

real eps = 9.81/303.;
real nu = 1;
real numu = nu/sqrt(0.09);
real nuep = pow(nu,1.5)/4.1;
real dt = 0.;

real Penalty = 1.e-6;

// Mesh
border b1(t=0, l){x=t; y=0;}
border b2(t=0, L-l){x=1.+t; y=-hSlope*t;}
border b3(t=-hSlope*(L-l), H){x=L; y=t;}
border b4(t=L, 0){x=t; y=H;}
border b5(t=H, h){x=0; y=t;}
border b6(t=h, 0){x=0; y=t;}

mesh Th=buildmesh(b1(nnPlus*nn*l) + b2(nn*sqrt((L-l)^2+(hSlope*(L-l))^2)) + b3(nn*(H
↪+ hSlope*(L-l))) + b4(nn*L) + b5(nn*(H-h)) + b6(nnPlus*nn*h));
plot(Th);

// Fespaces
fespace Vh2(Th, P1b);
Vh2 Ux, Uy;
Vh2 Vx, Vy;
Vh2 Upx, Upy;

fespace Vh(Th,P1);
Vh p=0, q;
Vh Tp, T=35;
```

(continues on next page)

(continued from previous page)

```
Vh k=0.0001, kp=k;
Vh ep=0.0001, epp=ep;

fespace V0h(Th,P0);
V0h muT=1;
V0h prodk, prode;
Vh kappa=0.25e-4, stress;

// Macro
macro grad(u) [dx(u), dy(u)] //
macro Grad(U) [grad(U#x), grad(U#y)] //
macro Div(U) (dx(U#x) + dy(U#y)) //

// Functions
func g = (x) * (1-x) * 4;

// Problem
real alpha = 0.;

problem Temperature(T, q)
   = int2d(Th)(
        alpha * T * q
      + kappa* grad(T)' * grad(q)
   )
   + int2d(Th)(
      - alpha*convect([Upx, Upy], -dt, Tp)*q
   )
   + on(b6, T=25)
   + on(b1, b2, T=30)
   ;

problem KineticTurbulence(k, q)
   = int2d(Th)(
        (epp/kp + alpha) * k * q
      + muT* grad(k)' * grad(q)
   )
   + int2d(Th)(
        prodk * q
      - alpha*convect([Upx, Upy], -dt, kp)*q
   )
   + on(b5, b6, k=0.00001)
   + on(b1, b2, k=beta*numu*stress)
   ;

problem ViscosityTurbulence(ep, q)
   = int2d(Th)(
        (1.92*epp/kp + alpha) * ep * q
      + muT * grad(ep)' * grad(q)
   )
   + int1d(Th, b1, b2)(
        T * q * 0.001
   )
   + int2d(Th)(
        prode * q
      - alpha*convect([Upx, Upy], -dt, epp)*q
   )
   + on(b5, b6, ep=0.00001)
```

(continues on next page)

(continued from previous page)

```
   + on(b1, b2, ep=beta*nuep*pow(stress,1.5))
   ;

// Initialization with stationary solution
solve NavierStokes ([Ux, Uy, p], [Vx, Vy, q])
   = int2d(Th)(
        alpha * [Ux, Uy]' * [Vx, Vy]
      + muT * (Grad(U) : Grad(V))
      + p * q * Penalty
      - p * Div(V)
      - Div(U) * q
   )
   + int1d(Th, b1, b2, b4)(
        Ux * Vx * 0.1
   )
   + int2d(Th)(
        eps * (T-35) * Vx
      - alpha*convect([Upx, Upy], -dt, Upx)*Vx
      - alpha*convect([Upx, Upy], -dt, Upy)*Vy
   )
   + on(b6, Ux=3, Uy=0)
   + on(b5, Ux=0, Uy=0)
   + on(b1, b4, Uy=0)
   + on(b2, Uy=-Upx*N.x/N.y)
   + on(b3, Uy=0)
   ;

plot([Ux, Uy], p, value=true, coef=0.2, cmm="[Ux, Uy] - p");

{
   real[int] xx(21), yy(21), pp(21);
   for (int i = 0 ; i < 21; i++){
      yy[i] = i/20.;
      xx[i] = Ux(0.5,i/20.);
      pp[i] = p(i/20.,0.999);
   }
   cout << " " << yy << endl;
   plot([xx, yy], wait=true, cmm="Ux x=0.5 cup");
   plot([yy, pp], wait=true, cmm="p y=0.999 cup");
}

// Initialization
dt = 0.1; //probably too big
int nbiter = 3;
real coefdt = 0.25^(1./nbiter);
real coefcut = 0.25^(1./nbiter);
real cut = 0.01;
real tol = 0.5;
real coeftol = 0.5^(1./nbiter);
nu = 1./reylnods;

T = T - 10*((x<1)*(y<0.5) + (x>=1)*(y+0.1*(x-1)<0.5));

// Convergence loop
real T0 = clock();
for (int iter = 1; iter <= nbiter; iter++){
   cout << "Iteration " << iter << " - dt = " << dt << endl;
```

(continues on next page)

(continued from previous page)

```
   alpha = 1/dt;

   // Time loop
   real t = 0.;
   for (int i = 0; i <= 500; i++){
      t += dt;
      cout << "Time step " << i << " - t = " << t << endl;

      // Update
      Upx = Ux;
      Upy = Uy;
      kp = k;
      epp = ep;
      Tp = max(T, 25); //for beauty only should be removed
      Tp = min(Tp, 35); //for security only should be removed
      kp = max(k, 0.0001); epp = max(ep, 0.0001); // to be secure: should not be active
      muT = 0.09*kp*kp/epp;

      // Solve NS
      NavierStokes;

      // Update
      prode = -0.126*kp*(pow(2*dx(Ux),2)+pow(2*dy(Uy),2)+2*pow(dx(Uy)+dy(Ux),2))/2;
      prodk = -prode*kp/epp*0.09/0.126;
      kappa = muT/0.41;
      stress = abs(dy(Ux));

      // Solve k-eps-T
      KineticTurbulence;
      ViscosityTurbulence;
      Temperature;

      // Plot
      plot(T, value=true, fill=true);
      plot([Ux, Uy], p, coef=0.2, cmm=" [Ux, Uy] - p", WindowIndex=1);

      // Time
      cout << "\tTime = " << clock()-T0 << endl;
   }

   // Check
   if (iter >= nbiter) break;

   // Adaptmesh
   Th = adaptmesh(Th, [dx(Ux), dy(Ux), dx(Ux), dy(Uy)], splitpbedge=1, abserror=0, cutoff=cut, err=tol, inquire=0, ratio=1.5, hmin=1./1000);
   plot(Th);

   // Update
   dt = dt * coefdt;
   tol = tol * coeftol;
   cut = cut * coefcut;
}
cout << "Total Time = " << clock()-T0 << endl;
```

(a) Temperature at time step 100

(b) Velocity at time step 100

(c) Temperature at time step 200

(d) Velocity at time step 200

(e) Temperature at time step 300

(f) Velocity at time step 300

(g) Temperature at time step 400

(h) Velocity at time step 400

(i) Temperature at time step 500

(j) Velocity at time step 500

Fig. 2.15: A large fluid problem

2.14 An Example with Complex Numbers

In a microwave oven heat comes from molecular excitation by an electromagnetic field. For a plane monochromatic wave, amplitude is given by Helmholtz's equation:

$$\beta v + \Delta v = 0.$$

We consider a rectangular oven where the wave is emitted by part of the upper wall. So the boundary of the domain is made up of a part Γ_1 where $v = 0$ and of another part $\Gamma_2 = [c, d]$ where for instance $v = \sin\left(\pi \frac{y-c}{c-d}\right)$.

Within an object to be cooked, denoted by B, the heat source is proportional to v^2. At equilibrium, one has :

$$\begin{array}{rcl} -\Delta\theta & = & v^2 I_B \\ \theta_\Gamma & = & 0 \end{array}$$

where I_B is 1 in the object and 0 elsewhere.

In the program below $\beta = 1/(1 - i/2)$ in the air and $2/(1 - i/2)$ in the object ($i = \sqrt{-1}$):

```
// Parameters
int nn = 2;
real a = 20.;
real b = 20.;
real c = 15.;
real d = 8.;
real e = 2.;
real l = 12.;
real f = 2.;
real g = 2.;

// Mesh
border a0(t=0, 1){x=a*t; y=0; label=1;}
border a1(t=1, 2){x=a; y=b*(t-1); label=1;}
border a2(t=2, 3){ x=a*(3-t); y=b; label=1;}
border a3(t=3, 4){x=0; y=b-(b-c)*(t-3); label=1;}
border a4(t=4, 5){x=0; y=c-(c-d)*(t-4); label=2;}
border a5(t=5, 6){x=0; y=d*(6-t); label=1;}

border b0(t=0, 1){x=a-f+e*(t-1); y=g; label=3;}
border b1(t=1, 4){x=a-f; y=g+l*(t-1)/3; label=3;}
border b2(t=4, 5){x=a-f-e*(t-4); y=l+g; label=3;}
border b3(t=5, 8){x=a-e-f; y=l+g-l*(t-5)/3; label=3;}

mesh Th = buildmesh(a0(10*nn) + a1(10*nn) + a2(10*nn) + a3(10*nn) +a4(10*nn) +
↪a5(10*nn)
   + b0(5*nn) + b1(10*nn) + b2(5*nn) + b3(10*nn));
real meat = Th(a-f-e/2, g+l/2).region;
real air= Th(0.01,0.01).region;
plot(Th, wait=1);

// Fespace
fespace Vh(Th, P1);
Vh R=(region-air)/(meat-air);
Vh<complex> v, w;
Vh vr, vi;

fespace Uh(Th, P1);
```

(continues on next page)

(continued from previous page)

```
Uh u, uu, ff;

// Problem
solve muwave(v, w)
   = int2d(Th)(
        v*w*(1+R)
      - (dx(v)*dx(w) + dy(v)*dy(w))*(1 - 0.5i)
   )
   + on(1, v=0)
   + on(2, v=sin(pi*(y-c)/(c-d)))
   ;

vr = real(v);
vi = imag(v);

// Plot
plot(vr, wait=1, ps="rmuonde.ps", fill=true);
plot(vi, wait=1, ps="imuonde.ps", fill=true);

// Problem (temperature)
ff=1e5*(vr^2 + vi^2)*R;

solve temperature(u, uu)
   = int2d(Th)(
        dx(u)* dx(uu)+ dy(u)* dy(uu)
   )
   - int2d(Th)(
        ff*uu
   )
   + on(1, 2, u=0)
   ;

// Plot
plot(u, wait=1, ps="tempmuonde.ps", fill=true);
```

Results are shown on Fig. 2.16a, Fig. 2.16b and Fig. 2.16c.

2.15 Optimal Control

Thanks to the function `BFGS` it is possible to solve complex nonlinear optimization problem within **FreeFEM**. For example consider the following inverse problem

$$\begin{array}{rcl} \min_{b,c,d\in R} J & = & \int_E (u-u_d)^2 \\ -\nabla(\kappa(b,c,d)\cdot\nabla u) & = & 0 \\ u|_\Gamma & = & u_\Gamma \end{array}$$

where the desired state u_d, the boundary data u_Γ and the observation set $E \subset \Omega$ are all given. Furthermore let us assume that:

$$\kappa(x) = 1 + bI_B(x) + cI_C(x) + dI_D(x) \quad \forall x \in \Omega$$

where B, C, D are separated subsets of Ω.

To solve this problem by the quasi-Newton BFGS method we need the derivatives of J with respect to b, c, d. We self

(a) Real part

(b) Imaginary part

(c) Temperature

Fig. 2.16: Microwave

explanatory notations, if $\delta b, \delta c, \delta d$ are variations of b, c, d we have:

$$\begin{array}{rcl} \delta J & \approx & 2\int_E (u-u_d)\delta u \\ -\nabla(\kappa\cdot\nabla\delta u) & \approx & \nabla(\delta\kappa\cdot\nabla u) \\ \delta u|_\Gamma & = & 0 \end{array}$$

Obviously J'_b is equal to δJ when $\delta b = 1, \delta c = 0, \delta d = 0$, and so on for J'_c and J'_d.

All this is implemented in the following program:

```
// Mesh
border aa(t=0, 2*pi){x=5*cos(t); y=5*sin(t);};
border bb(t=0, 2*pi){x=cos(t); y=sin(t);};
border cc(t=0, 2*pi){x=-3+cos(t); y=sin(t);};
border dd(t=0, 2*pi){x=cos(t); y =-3+sin(t);};

mesh th = buildmesh(aa(70) + bb(35) + cc(35) + dd(35));

// Fespace
fespace Vh(th, P1);
Vh Ib=((x^2+y^2)<1.0001),
   Ic=(((x+3)^2+ y^2)<1.0001),
   Id=((x^2+(y+3)^2)<1.0001),
   Ie=(((x-1)^2+ y^2)<=4),
   ud, u, uh, du;

// Problem
real[int] z(3);
problem A(u, uh)
   = int2d(th)(
        (1+z[0]*Ib+z[1]*Ic+z[2]*Id)*(dx(u)*dx(uh) + dy(u)*dy(uh))
   )
   + on(aa, u=x^3-y^3)
   ;

// Solve
z[0]=2; z[1]=3; z[2]=4;
A;
ud = u;

ofstream f("J.txt");
func real J(real[int] & Z){
   for (int i = 0; i < z.n; i++)
      z[i] =Z[i];
   A;
   real s = int2d(th)(Ie*(u-ud)^2);
   f << s << " ";
   return s;
}

// Problem BFGS
real[int] dz(3), dJdz(3);
problem B (du, uh)
   = int2d(th)(
        (1+z[0]*Ib+z[1]*Ic+z[2]*Id)*(dx(du)*dx(uh) + dy(du)*dy(uh))
   )
   + int2d(th)(
        (dz[0]*Ib+dz[1]*Ic+dz[2]*Id)*(dx(u)*dx(uh) + dy(u)*dy(uh))
   )
```

(continues on next page)

(continued from previous page)

```
   +on(aa, du=0)
   ;

func real[int] DJ(real[int] &Z){
   for(int i = 0; i < z.n; i++){
      for(int j = 0; j < dz.n; j++)
         dz[j] = 0;
      dz[i] = 1;
      B;
      dJdz[i] = 2*int2d(th)(Ie*(u-ud)*du);
   }
   return dJdz;
}

real[int] Z(3);
for(int j = 0; j < z.n; j++)
   Z[j]=1;

BFGS(J, DJ, Z, eps=1.e-6, nbiter=15, nbiterline=20);
cout << "BFGS: J(z) = " << J(Z) << endl;
for(int j = 0; j < z.n; j++)
   cout << z[j] << endl;

// Plot
plot(ud, value=1, ps="u.eps");
```

In this example the sets B, C, D, E are circles of boundaries bb, cc, dd, ee and the domain Ω is the circle of boundary aa.

The desired state u_d is the solution of the PDE for $b = 2, c = 3, d = 4$. The unknowns are packed into array z.

Note: It is necessary to recopy Z into z because one is a local variable while the other one is global.

The program found $b = 2.00125, c = 3.00109, d = 4.00551$.

Fig. 2.17a and Fig. 2.17b show u at convergence and the successive function evaluations of J.

Note that an *adjoint state* could have been used. Define p by:

$$\begin{array}{rcl} -\nabla\cdot(\kappa\nabla p) & = & 2I_E(u-u_d) \\ p|_\Gamma & = & 0 \end{array}$$

Consequently:

$$\begin{array}{rcl} \delta J & = & -\int_\Omega(\nabla\cdot(\kappa\nabla p))\delta u \\ & = & \int_\Omega(\kappa\nabla p\cdot\nabla\delta u) \\ & = & -\int_\Omega(\delta\kappa\nabla p\cdot\nabla u) \end{array}$$

Then the derivatives are found by setting $\delta b = 1, \delta c = \delta d = 0$ and so on:

$$\begin{array}{rcl} J'_b & = & -\int_B \nabla p\cdot\nabla u \\ J'_c & = & -\int_C \nabla p\cdot\nabla u \\ J'_d & = & -\int_D \nabla p\cdot\nabla u \end{array}$$

Note: As BFGS stores an $M\times M$ matrix where M is the number of unknowns, it is dangerously expensive to use this method when the unknown x is a Finite Element Function. One should use another optimizer such as the NonLinear Conjugate Gradient `NLCG` (also a key word of **FreeFEM**).

(a) Level line of u.

(b) Successive evaluations of J by BFGS (5 values above 500 have been removed for readability)

Fig. 2.17: Optimal control

2.16 A Flow with Shocks

Compressible Euler equations should be discretized with Finite Volumes or FEM with flux up-winding scheme but these are not implemented in **FreeFEM**. Nevertheless acceptable results can be obtained with the method of characteristics provided that the mean values $\bar{f} = \frac{1}{2}\left(f^+ + f^-\right)$ are used at shocks in the scheme, and finally mesh adaptation.

$$\begin{aligned}
\partial_t \rho + \bar{u}\nabla\rho + \bar{\rho}\nabla\cdot u &= 0 \\
\bar{\rho}(\partial_t u + \tfrac{\bar{\rho}\bar{u}}{\bar{\rho}}\nabla u + \nabla p &= 0 \\
\partial_t p + \bar{u}\nabla p + (\gamma-1)\bar{p}\nabla\cdot u &= 0
\end{aligned}$$

One possibility is to couple u, p and then update ρ, i.e.:

$$\begin{aligned}
\tfrac{1}{(\gamma-1)\delta t\bar{p}^m}(p^{m+1} - p^m \circ X^m) + \nabla\cdot u^{m+1} &= 0 \\
\tfrac{\bar{\rho}^m}{\delta t}(u^{m+1} - u^m \circ \tilde{X}^m) + \nabla p^{m+1} &= 0 \\
\rho^{m+1} = \rho^m \circ X^m + \tfrac{\bar{\rho}^m}{(\gamma-1)\bar{p}^m}(p^{m+1} &- p^m \circ X^m)
\end{aligned}$$

A numerical result is given on Fig. 2.18 and the **FreeFEM** script is

```
// Parameters
verbosity = 1;
int anew = 1;
int m = 5;
real x0 = 0.5;
real y0 = 0.;
real rr = 0.2;
real dt = 0.01;
real u0 = 2.;
real err0 = 0.00625;
real pena = 2.;

// Mesh
border ccc(t=0, 2){x=2-t; y=1;};
```

(continues on next page)

(continued from previous page)

```
border ddd(t=0, 1){x=0; y=1-t;};
border aaa1(t=0, x0-rr){x=t; y=0;};
border cercle(t=pi, 0){x=x0+rr*cos(t); y=y0+rr*sin(t);}
border aaa2(t=x0+rr, 2){x=t; y=0;};
border bbb(t=0, 1){x=2; y=t;};

 mesh Th;
if(anew)
   Th = buildmesh (ccc(5*m) + ddd(3*m) + aaa1(2*m) + cercle(5*m) + aaa2(5*m) +
↪bbb(2*m));
else
   Th = readmesh("Th_circle.mesh"); plot(Th);

// fespace
fespace Wh(Th, P1);
Wh u, v;
Wh u1, v1;
Wh uh, vh;

fespace Vh(Th, P1);
Vh r, rh, r1;

// Macro
macro dn(u) (N.x*dx(u)+N.y*dy(u)) //

// Initialization
if(anew){
   u1 = u0;
   v1 = 0;
   r1 = 1;
}
else{
   ifstream g("u.txt"); g >> u1[];
   ifstream gg("v.txt"); gg >> v1[];
   ifstream ggg("r.txt"); ggg >> r1[];
   plot(u1, ps="eta.eps", value=1, wait=1);
   err0 = err0/10;
   dt = dt/10;
}

// Problem
problem euler(u, v, r, uh, vh, rh)
   = int2d(Th)(
         (u*uh + v*vh + r*rh)/dt
      + ((dx(r)*uh + dy(r)*vh) - (dx(rh)*u + dy(rh)*v))
   )
   + int2d(Th)(
      - (
            rh*convect([u1,v1],-dt,r1)
         + uh*convect([u1,v1],-dt,u1)
         + vh*convect([u1,v1],-dt,v1)
      )/dt
   )
   +int1d(Th, 6)(
         rh*u
   )
   + on(2, r=0)
```

(continues on next page)

(continued from previous page)

```
   + on(2, u=u0)
   + on(2, v=0)
   ;

// Iterations
int j = 80;
for(int k = 0; k < 3; k++){
   if(k==20){
      err0 = err0/10;
      dt = dt/10;
      j = 5;
   }

   // Solve
   for(int i = 0; i < j; i++){
      euler;
      u1=u;
      v1=v;
      r1=abs(r);
      cout << "k = " << k << " E = " << int2d(Th)(u^2+v^2+r) << endl;
      plot(r, value=1);
   }

   // Mesh adaptation
   Th = adaptmesh (Th, r, nbvx=40000, err=err0, abserror=1, nbjacoby=2, omega=1.8, ratio=1.8, nbsmooth=3, splitpbedge=1, maxsubdiv=5, rescaling=1);
   plot(Th);
   u = u;
   v = v;
   r = r;

   // Save
   savemesh(Th, "Th_circle.mesh");
   ofstream f("u.txt"); f << u[];
   ofstream ff("v.txt"); ff << v[];
   ofstream fff("r.txt"); fff << r[];
   r1 = sqrt(u*u+v*v);
   plot(r1, ps="mach.eps", value=1);
   r1 = r;
}
```

2.17 Time dependent schema optimization for heat equations

First, it is possible to define variational forms, and use this forms to build matrix and vector to make very fast script (4 times faster here).

For example solve the *ThermalConduction* problem, we must solve the temperature equation in Ω in a time interval (0,T).

$$
\begin{array}{rcll}
\partial_t u - \nabla \cdot (\kappa \nabla u) & = & 0 & \text{in } \Omega \times (0,T) \\
u(x,y,0) & = & u_0 + x u_1 & \\
u & = & 30 & \text{on } \Gamma_{24} \times (0,T) \\
\kappa \frac{\partial u}{\partial n} + \alpha (u - u_e) & = & 0 & \text{on } \Gamma \times (0,T)
\end{array}
$$

Fig. 2.18: Pressure for a Euler flow around a disk at Mach 2 computed by (2.6)

The variational formulation is in $L^2(0,T;H^1(\Omega))$; we shall seek u^n satisfying:

$$\forall w \in V_0;\ \int_\Omega \frac{u^n - u^{n-1}}{\delta t} w + \kappa \nabla u^n \nabla w) + \int_\Gamma \alpha(u^n - u_{ue}) w = 0$$

where $V_0 = \{w \in H^1(\Omega)/w_{|\Gamma_{24}} = 0\}$.

So, to code the method with the matrices $A = (A_{ij})$, $M = (M_{ij})$, and the vectors $u^n, b^n, b', b", b_{cl}$ (notation if w is a vector then w_i is a component of the vector).

$$u^n = A^{-1} b^n, \quad b' = b_0 + M u^{n-1}, \quad b" = \frac{1}{\varepsilon}\, b_{cl}, \quad b_i^n = \begin{cases} b_i'' & \text{if } i \in \Gamma_{24} \\ b_i' & \text{else} \end{cases}$$

Where with $\frac{1}{\varepsilon} = \texttt{tgv} = 10^{30}$:

$$\begin{array}{rcl} A_{ij} & = & \begin{cases} \frac{1}{\varepsilon} & \text{if } i \in \Gamma_{24}, \text{and} \quad j = i \\ \int_\Omega w_j w_i / dt + k(\nabla w_j . \nabla w_i) + \int_{\Gamma_{13}} \alpha w_j w_i & \text{else} \end{cases} \\ M_{ij} & = & \begin{cases} \frac{1}{\varepsilon} & \text{if } i \in \Gamma_{24}, \text{and} \quad j = i \\ n \int_\Omega w_j w_i / dt & \text{else} \end{cases} \\ b_{0,i} & = & n \int_{\Gamma_{13}} \alpha u_{ue} w_i \\ b_{cl} & = & u^0 \quad \text{the initial data} \end{array}$$

The Fast version script:

```
...
Vh u0=fu0, u=u0;
```

Create three variational formulation, and build the matrices A,M.

```
varf vthermic (u, v)
    = int2d(Th)(
          u*v/dt
        + k*(dx(u)*dx(v) + dy(u)*dy(v))
    )
    + int1d(Th, 1, 3)(
          alpha*u*v
    )
    + on(2,4,u=1)
    ;

varf vthermic0 (u, v)
    = int1d(Th, 1, 3)(
          alpha*ue*v
    )
    ;
varf vMass (u,v)
    = int2d(Th)(
          u*v/dt
    )
    + on(2, 4, u=1)
    ;

real tgv = 1e30;
matrix A = vthermic(Vh, Vh, tgv=tgv, solver=CG);
matrix M = vMass(Vh, Vh);
```

Now, to build the right hand size; we need 4 vectors.

```
real[int] b0 = vthermic0(0,Vh); //constant part of RHS
real[int] bcn = vthermic(0,Vh); //tgv on Dirichlet part
real[int] bcl = tgv*u0[];   //the Dirichlet B.C. part

// The fast loop
for(real t = 0; t < T; t += dt){
    real[int] b = b0;   //the RHS
    b += M*u[]; //add the the time dependent part
    b = bcn ? bcl : b; //do $\forall i$: b[i] = bcn[i] ? bcl[i] : b[i];
    u[] = A^-1*b; //solve linear problem
    plot(u);
}
```

2.18 Tutorial to write a transient Stokes solver in matrix form

Consider the following script to solve a time dependent Stokes problem in a cavity

```
// Parameters
real nu = 0.1;
real T=1.;
real dt = 0.1;

// Mesh
mesh Th = square(10, 10);

// Fespace
fespace Vh(Th, P2)
Vh u, v;
Vh uu, vv;
Vh uold=0, vold=0;

fespace Qh(Th, P1);
Qh p;
Qh pp;

// Problem
problem stokes (u, v, p, uu, vv, pp)
    = int2d(Th)(
          (u*uu+v*vv)/dt
        + nu*(dx(u)*dx(uu) + dy(u)*dy(uu) + dx(v)*dx(vv) + dy(v)*dy(vv))
        - p*pp*1.e-6
        - p*(dx(uu) + dy(vv))
        - pp*(dx(u) + dy(v))
    )
    - int2d(Th)(
          (uold*uu+vold*vv)/dt
    )
    + on(1, 2, 4, u=0, v=0)
    + on(3, u=1, v=0)
    ;

// Time loop
int m, M = T/dt;
for(m = 0; m < M; m++){
    stokes;
```

(continues on next page)

(continued from previous page)

```
    uold = u;
    vold = v;
}

// Plot
plot(p, [u, v], value=true, wait=true, cmm="t="+m*dt);
```

Every iteration is in fact of the form $A[u, v, p] = B[uold, vold, pold] + b$ where A, B are matrices and b is a vector containing the boundary conditions. A, B, b are constructed by:

```
fespace Xh(Th, [P2, P2, P1]);
varf aa ([u, v, p], [uu, vv, pp])
    = int2d(Th)(
          (u*uu+v*vv)/dt
        + nu*(dx(u)*dx(uu) + dy(u)*dy(uu) + dx(v)*dx(vv) + dy(v)*dy(vv))
        - p*pp*1.e-6
        - p*(dx(uu) + dy(vv))
        - pp*(dx(u) + dy(v))
    )
    + on(1, 2, 4, u=0, v=0)
    + on(3, u=1, v=0)
    ;

varf bb ([uold, vold, pold], [uu, vv, pp])
    = int2d(Th)(
          (uold*uu+vold*vv)/dt
    )
    //+ on(1, 2, 4, uold=0, vold=0)
    //+ on(3, uold=1, vold=0)
    ;

varf bcl ([uold, vold, pold], [uu, vv, pp])
    = on(1, 2, 4, uold=0, vold=0)
    + on(3, uold=1, vold=0)
    ;

matrix A = aa(Xh, Xh, solver=UMFPACK);
matrix B = bb(Xh, Xh);
real[int] b = bcl(0, Xh);
```

Note that the boundary conditions are not specified in bb. Removing the comment `//` would cause the compiler to multiply the diagonal terms corresponding to a Dirichlet degree of freedom by a very large term (`tgv`); if so b would not be needed, on the condition that $uold = 1$ on boundary 3 initially. Note also that b has a tgv on the Dirichlet nodes, by construction, and so does A.

The loop will then be:

```
real[int] sol(Xh.ndof), aux(Xh.ndof);
for (m = 0; m < M; m++){
    aux = B*sol; aux += b;
    sol = A^-1 * aux;
}
```

There is yet a difficulty with the initialization of `sol` and with the solution from `sol`. For this we need a temporary vector in X_h and here is a solution:

```
Xh [w1, w2, wp] = [uold, vold, pp];
sol = w1[]; //cause also the copy of w2 and wp
for (m = 0; m < M; m++){
    aux = B*sol; aux += b;
    sol = A^-1 * aux;
}
w1[]=sol; u=w1; v= w2; p=wp;
plot(p, [u, v], value=true, wait=true, cmm="t="+m*dt);
```

The freefem team agrees that the line `sol=w1[];` is mysterious as it copies also w2 and wp into sol. Structured data such as vectors of X_h here cannot be written component by component. Hence `w1=u` is not allowed.

2.19 Wifi Propagation

2.19.1 Summary

In this tutorial, we will study the wifi signal power in a flat. An awesome flat is especially designed for the experiment, with **2** walls:

Fig. 2.19: Flat

Even if the flat seems small enough to be covered by wifi everywhere, it is still interesting to study where the signal's power is the lowest. We will study where to put the hotspot to get the best coverage, and as we're a bit lazy we will only put it next to the left wall.

2.19.2 Physics

In a nutshell, the Wifi is a electromagnetic wave that contains a signal : Internet data. Electromagnetic waves are well know by physicists and are ruled by the **4 Maxwell equations** which give you the solution for E, the electrical field, and B, the magnetic field, in space but also in time.

We don't care about the time here, because the signal period is really short so our internet quality will not change with time. Without time, we're looking for stationaries solutions, and the Maxwell equations can be simplified to one equation, the Helmholtz one :

$$\nabla^2 E + \frac{k^2}{n^2} E = 0$$

Where k is the angular wavenumber of the wifi signal, and n the refractive index of the material the wave is in.

Indeed, the main point of this study is the impact of **walls** on the signal's power, where the n is different from air (where it is 1). In walls, the refractive index is a complex number in which the two parts have a physic interpretation:

- The *real part* defines the **reflexion** of the wall (the amount of signal that doesn't pass).
- The *imaginary part* defines the **absorption** of the wall (the amount that disappears).

The wifi hotspot (simulated by a simple circle) will be the boundary condition, with a non null value for our electrical field.

2.19.3 Coding

The domain

In order to create the domain of experimentation, we need to create `border` objects, like this :

```
real a = 40, b = 40, c = 0.5;
border a00(t=0, 1) {x=a*t; y=0; label=1;}
border a10(t=0, 1) {x=a; y=b*t; label=1;}
border a20(t=1, 0) {x=a*t; y=b; label=1;}
border a30(t=1, 0) {x=0; y=b*t; label=1;}
border a01(t=0, 1) {x=c+(a-c*2)*t; y=c; label=1;}
border a11(t=0, 1) {x=a-c; y=c+(b-c*2)*t; label=1;}
border a21(t=1, 0) {x=c+(a-c*2)*t; y=b-c; label=1;}
border a31(t=1, 0) {x=c; y=c+(b-c*2)*t; label=1;}

real p = 5, q = 20, d = 34, e = 1;
border b00(t=0, 1) {x=p+d*t; y=q; label=3;}
border b10(t=0, 1) {x=p+d; y=q+e*t; label=3;}
border b20(t=1, 0) {x=p+d*t; y=q+e; label=3;}
border b30(t=1, 0) {x=p; y=q+e*t; label=3;}

real r = 30, s =1 , j = 1, u = 15;
border c00(t=0, 1) {x=r+j*t; y=s; label=3;}
border c10(t=0, 1) {x=r+j; y=s+u*t; label=3;}
border c20(t=1, 0) {x=r+j*t; y=s+u; label=3;}
border c30(t=1, 0) {x=r; y=s+u*t; label=3;}
```

Let's create a mesh

```
int n=13;
mesh Sh = buildmesh(a00(10*n) + a10(10*n) + a20(10*n) + a30(10*n)
    + a01(10*n) + a11(10*n) + a21(10*n) + a31(10*n)
    + b00(5*n) + b10(5*n) + b20(5*n) + b30(5*n)
    + c00(5*n) + c10(5*n) + c20(5*n) + c30(5*n));
plot(Sh, wait=1);
```

So we are creating a `mesh`, and plotting it :

There is currently no wifi hotspot, and as we want to resolve the equation for a multiple number of position next to the left wall, let's do a `for` loop:

```
int bx;
for (bx = 1; bx <= 7; bx++){
    border C(t=0, 2*pi){x=2+cos(t); y=bx*5+sin(t); label=2;}

```

(continues on next page)

Fig. 2.20: Mesh

(continued from previous page)

```
    mesh Th = buildmesh(a00(10*n) + a10(10*n) + a20(10*n) + a30(10*n)
        + a01(10*n) + a11(10*n) + a21(10*n) + a31(10*n) + C(10)
        + b00(5*n) + b10(5*n) + b20(5*n) + b30(5*n)
        + c00(5*n) + c10(5*n) + c20(5*n) + c30(5*n));
```

The border `C` is our hotspot and as you can see a simple circle. `Th` is our final mesh, with all borders and the hotspot. Let's resolve this equation !

```
fespace Vh(Th, P1);
func real wall() {
   if (Th(x,y).region == Th(0.5,0.5).region || Th(x,y).region == Th(7,20.5).region ||
↪Th(x,y).region == Th(30.5,2).region) { return 1; }
   else { return 0; }
}

Vh<complex> v,w;

randinit(900);
Vh wallreflexion = randreal1();
Vh<complex> wallabsorption = randreal1()*0.5i;
Vh k = 6;

cout << "Reflexion of walls : " << wallreflexion << "\n";
cout << "Absorption of walls : " << wallabsorption << "\n";

problem muwave(v,w) =
    int2d(Th)(
          (v*w*k^2)/(1+(wallreflexion+wallabsorption)*wall())^2
        - (dx(v)*dx(w)+dy(v)*dy(w))
    )
    + on(2, v=1)
```

(continues on next page)

(continued from previous page)

```
    ;

muwave;
Vh vm = log(real(v)^2 + imag(v)^2);
plot(vm, wait=1, fill=true, value=0, nbiso=65);
}
```

A bit of understanding here :

- The `fespace` keyword defines a finite elements space, no need to know more here.
- The function `wall` return 0 if in air and 1 if in a wall (x and y are global variables).
- For this example, random numbers are used for the reflexion and the absorption.
- The problem is defined with `problem` and we solve it by calling it.

Finally, I plotted the log of the module of the solution `v` to see the signal's power, and here we are :

Fig. 2.21: Solution

Beautiful isn't it ? This is the first position for the hotspot, but there are 6 others, and the electrical field is evolving depending on the position. You can see the other positions here :

2.20 Plotting in Matlab and Octave

2.20.1 Overview

In order to create a plot of a **FreeFEM** simulation in Matlab© or Octave two steps are necessary:

- The mesh, the finite element space connectivity and the simulation data must be exported into files
- The files must be imported into the Matlab / Octave workspace. Then the data can be visualized with the ffmatlib library

(a) Point 2 **(b)** Point 3 **(c)** Point 4

(d) Point 5 **(e)** Point 6 **(f)** Point 7

Fig. 2.22: Wifi propagation

The steps are explained in more detail below using the example of a stripline capacitor.

Note: Finite element variables must be in P1 or P2. The simulation data can be 2D or 3D.

2.20.2 2D Problem

Consider a stripline capacitor problem which is also shown in Fig. 2.23. On the two boundaries (the electrodes) C_A, C_K a Dirichlet condition and on the enclosure C_B a Neumann condition is set. The electrostatic potential u between the two electrodes is given by the Laplace equation

$$\Delta u(x, y) = 0$$

and the electrostatic field $\mathbf{E}$ is calculated by

$$\mathbf{E} = -\nabla u$$

```
int CA=3, CK=4, CB=5;
real w2=1.0, h=0.4, d2=0.5;

border bottomA(t=-w2,w2){ x=t; y=d2; label=CA;};
border rightA(t=d2,d2+h){ x=w2; y=t; label=CA;};
border topA(t=w2,-w2){ x=t; y=d2+h; label=CA;};
border leftA(t=d2+h,d2){ x=-w2; y=t; label=CA;};

border bottomK(t=-w2,w2){ x=t; y=-d2-h; label=CK;};
border rightK(t=-d2-h,-d2){ x=w2; y=t; label=CK;};
border topK(t=w2,-w2){ x=t; y=-d2; label=CK;};
border leftK(t=-d2,-d2-h){ x=-w2; y=t; label=CK;};

border enclosure(t=0,2*pi){x=5*cos(t); y=5*sin(t); label=CB;}

int n=15;
mesh Th = buildmesh(enclosure(3*n)+
             bottomA(-w2*n)+topA(-w2*n)+rightA(-h*n)+leftA(-h*n)+
             bottomK(-w2*n)+topK(-w2*n)+rightK(-h*n)+leftK(-h*n));

fespace Vh(Th,P1);

Vh u,v;
real u0=2.0;

problem Laplace(u,v,solver=LU) =
          int2d(Th)(dx(u)*dx(v) + dy(u)*dy(v))
        + on(CA,u=u0)+on(CK,u=0);

real error=0.01;
for (int i=0;i<1;i++){
   Laplace;
   Th=adaptmesh(Th,u,err=error);
   error=error/2.0;
}
Laplace;

Vh Ex, Ey;
```

(continues on next page)

(continued from previous page)

```
Ex = -dx(u);
Ey = -dy(u);

plot(u,[Ex,Ey],wait=true);
```

2.20.3 Exporting Data

The mesh is stored with the **FreeFEM** command *savemesh()*, while the connectivity of the finite element space and the simulation data are stored with the macro commands `ffSaveVh()` and `ffSaveData()`. These two commands are located in the `ffmatlib.idp` file which is included in the `ffmatlib`. Therefore, to export the stripline capacitor data the following statement sequence must be added to the **FreeFEM** code:

```
include "ffmatlib.idp"

//Save mesh
savemesh(Th,"capacitor.msh");
//Save finite element space connectivity
ffSaveVh(Th,Vh,"capacitor_vh.txt");
//Save some scalar data
ffSaveData(u,"capacitor_potential.txt");
//Save a 2D vector field
ffSaveData2(Ex,Ey,"capacitor_field.txt");
```

2.20.4 Importing Data

The mesh file can be loaded into the Matlab / Octave workspace using the `ffreadmesh()` command. A mesh file consists of *three main sections*:

1. The mesh points as nodal coordinates
2. A list of boundary edges including boundary labels
3. List of triangles defining the mesh in terms of connectivity

The three data sections mentioned are returned in the variables `p`, `b` and `t`. The finite element space connectivity and the simulation data can be loaded using the `ffreaddata()` command. Therefore, to load the example data the following statement sequence must be executed in Matlab / Octave:

```
%Add ffmatlib to the search path
addpath('add here the link to the ffmatlib');
%Load the mesh
[p,b,t,nv,nbe,nt,labels]=ffreadmesh('capacitor.msh');
%Load the finite element space connectivity
vh=ffreaddata('capacitor_vh.txt');
%Load scalar data
u=ffreaddata('capacitor_potential.txt');
%Load 2D vector field data
[Ex,Ey]=ffreaddata('capacitor_field.txt');
```

2.20.5 2D Plot Examples

`ffpdeplot()` is a plot solution for creating patch, contour, quiver, mesh, border, and region plots of 2D geometries. The basic syntax is:

```
[handles,varargout] = ffpdeplot(p,b,t,varargin)
```

`varargin` specifies parameter name / value pairs to control the plot behaviour. A table showing all options can be found in the ffmatlib documentation. A small selection of possible plot commands is given as follows:

- Plot of the boundary and the mesh:

```
ffpdeplot(p,b,t,'Mesh','on','Boundary','on');
```

Fig. 2.23: Boundary and Mesh

- Patch plot (2D map or density plot) including mesh and boundary:

```
ffpdeplot(p,b,t,'VhSeq',vh,'XYData',u,'Mesh','on','Boundary','on', ...
          'XLim',[-2 2],'YLim',[-2 2]);
```

Fig. 2.24: Patch Plot with Mesh

- 3D surf plot:

```
ffpdeplot(p,b,t,'VhSeq',vh,'XYData',u,'ZStyle','continuous', ...
          'Mesh','off');
lighting gouraud;
view([-47,24]);
camlight('headlight');
```

Fig. 2.25: 3D Surf Plot

- Contour (isovalue) and quiver (vector field) plot:

```
ffpdeplot(p,b,t,'VhSeq',vh,'XYData',u,'Mesh','off','Boundary','on', ...
          'XLim',[-2 2],'YLim',[-2 2],'Contour','on','CColor','b', ...
          'XYStyle','off', 'CGridParam',[150, 150],'ColorBar','off', ...
          'FlowData',[Ex,Ey],'FGridParam',[24, 24]);
```

Fig. 2.26: Contour and Quiver Plot

Download run through example:

`Matlab / Octave file`

`FreeFEM script`

2.20.6 3D Plot Examples

3D problems are handled by the `ffpdeplot3D()` command, which works similarly to the `ffpdeplot()` command. In particular in three-dimensions cross sections of the solution can be created. The following example shows a cross-sectional problem of a three-dimensional parallel plate capacitor.

Fig. 2.27: Slice on a 3D Parallel Plate Capacitor

Download run through example:

`Matlab / Octave file`

`FreeFEM script`

2.20.7 References

- Octave
- Matlab
- ffmatlib

CHAPTER

THREE

DOCUMENTATION

The fruit of a long maturing process, **freefem**, in its last avatar, **FreeFEM** , is a high level integrated development environment (IDE) for numerically solving partial differential equations (PDE) in dimension 2 and 3. It is the ideal tool for teaching the finite element method but it is also perfect for research to quickly test new ideas or multi-physics and complex applications.

FreeFEM has an advanced automatic mesh generator, capable of a posteriori mesh adaptation; it has a general purpose elliptic solver interfaced with fast algorithms, such as the multi-frontal method UMFPACK, SuperLU, MUMPS. Hyperbolic and parabolic problems are solved by iterative algorithms prescribed by the user with the high level language of **FreeFEM**. It has several triangular finite elements, including discontinuous elements. Everything is there in **FreeFEM** to prepare research quality reports with online color display, zooming and other features as well as postscript printouts.

This manual is meant for students at a Masters level, for researchers at any level, and for engineers (including financial engineering) with some understanding of variational methods for partial differential equations.

Introduction

A partial differential equation is a relation between a function of several variables and its (partial) derivatives. Many problems in physics, engineering, mathematics and even banking are modeled by one or several partial differential equations.

FreeFEM is a software to solve these equations numerically. As its name implies, it is a free software (see the copyrights for full detail) based on the Finite Element Method; it is not a package, it is an integrated product with its own high level programming language. This software runs on all UNIX OS (with g++ 3.3 or later, and OpenGL), on Window XP, Vista and 7, 8, 10 and on MacOS 10 intel.

Moreover **FreeFEM** is highly adaptive. Many phenomena involve several coupled systems. Fluid-structure interactions, Lorentz forces for aluminum casting and ocean-atmosphere problems are three such systems. These require different finite element approximations and polynomial degrees, possibly on different meshes. Some algorithms like the Schwarz' domain decomposition method also requires data interpolation on multiple meshes within one program. **FreeFEM** can handle these difficulties, i.e. arbitrary finite element spaces on arbitrary unstructured and adapted bi-dimensional meshes.

The characteristics of **FreeFEM** are:

- Problem description (real or complex valued) by their variational formulations, with access to the internal vectors and matrices if needed.
- Multi-variables, multi-equations, bi-dimensional and three-dimensional static or time dependent, linear or non-linear coupled systems; however the user is required to describe the iterative procedures which reduce the problem to a set of linear problems.
- Easy geometric input by analytic description of boundaries by pieces; however this part is not a CAD system; for instance when two boundaries intersect, the user must specify the intersection points.

- Automatic mesh generator, based on the Delaunay-Voronoi algorithm; the inner point density is proportional to the density of points on the boundaries [GEORGE1996].
- Metric-based anisotropic mesh adaptation. The metric can be computed automatically from the Hessian of any **FreeFEM** function [HECHT1998].
- High level user friendly typed input language with an algebra of analytic and finite element functions.
- Multiple finite element meshes within one application with automatic interpolation of data on different meshes and possible storage of the interpolation matrices.
- A large variety of triangular finite elements: linear, quadratic Lagrangian elements and more, discontinuous P1 and Raviart-Thomas elements, elements of a non-scalar type, the mini-element,... (but no quadrangles).
- Tools to define discontinuous Galerkin finite element formulations P0, P1dc, P2dc and keywords: jump, mean, intalledges.
- A large variety of linear direct and iterative solvers (LU, Cholesky, Crout, CG, GMRES, UMFPACK, MUMPS, SuperLU, ...) and eigenvalue and eigenvector solvers (ARPARK) .
- Near optimal execution speed (compared with compiled `C++` implementations programmed directly).
- Online graphics, generation of ,.txt,.eps,.gnu, mesh files for further manipulations of input and output data.
- Many examples and tutorials: elliptic, parabolic and hyperbolic problems, Navier-Stokes flows, elasticity, fluid structure interactions, Schwarz's domain decomposition method, eigenvalue problem, residual error indicator, ...
- A parallel version using MPI

History

The project has evolved from MacFem, PCfem, written in Pascal. The first C version lead to `freefem 3.4`; it offered mesh adaptivity on a single mesh only.

A thorough rewriting in `C++` led to `freefem+` (`freefem+` 1.2.10 was its last release), which included interpolation over multiple meshes (functions defined on one mesh can be used on any other mesh); this software is no longer maintained but is still in use because it handles a problem description using the strong form of the PDEs. Implementing the interpolation from one unstructured mesh to another was not easy because it had to be fast and non-diffusive; for each point, one had to find the containing triangle. This is one of the basic problems of computational geometry (see [PREPARATA1985] for example). Doing it in a minimum number of operations was the challenge. Our implementation is $\mathcal{O}(nlogn)$ and based on a quadtree. This version also grew out of hand because of the evolution of the template syntax in `C++`.

We have been working for a few years now on **FreeFEM** , entirely re-written again in `C++` with a thorough usage of template and generic programming for coupled systems of unknown size at compile time. Like all versions of `freefem`, it has a high level user friendly input language which is not too far from the mathematical writing of the problems.

The `freefem` language allows for a quick specification of any partial differential system of equations. The language syntax of **FreeFEM** is the result of a new design which makes use of the STL [STROUSTRUP2000], templates, and bison for its implementation; more details can be found in [HECHT2002]. The outcome is a versatile software in which any new finite elements can be included in a few hours; but a recompilation is then necessary. Therefore the library of finite elements available in **FreeFEM** will grow with the version number and with the number of users who program more new elements. So far we have discontinuous P_0 elements,linear P_1 and quadratic P_2 Lagrangian elements, discontinuous P_1 and Raviart-Thomas elements and a few others like bubble elements.

3.1 Notations

Here mathematical expressions and corresponding **FreeFEM** commands are explained.

3.1.1 Generalities

- [δ_{ij}] Kronecker delta (0 if $i \neq j$, 1 if $i = j$ for integers i, j)
- [$\forall$] for all
- [$\exists$] there exists
- [i.e.] that is
- [PDE] partial differential equation (with boundary conditions)
- [$\emptyset$] the empty set
- [$\mathbb{N}$] the set of integers ($a \in \mathbb{N} \Leftrightarrow$ `int a`), `int` means `long int` inside **FreeFEM**
- [$\mathbb{R}$] the set of real numbers ($a \in \mathbb{R} \Leftrightarrow$ `real a`), `double` inside **FreeFEM**
- [$\mathbb{C}$] the set of complex numbers ($a \in \mathbb{C} \Leftrightarrow$ `complex a`), `complex<double>`
- [$\mathbb{R}^d$] d-dimensional Euclidean space

3.1.2 Sets, Mappings, Matrices, Vectors

Let E, F, G be three sets and A the subset of E.

- [$\{x \in E|\ P\}$] the subset of E consisting of the elements possessing the property P
- [$E \cup F$] the set of elements belonging to E or F
- [$E \cap F$] the set of elements belonging to E and F
- [$E \setminus A$] the set $\{x \in E|\ x \notin A\}$
- [$E + F$] $E \cup F$ with $E \cap F = \emptyset$
- [$E \times F$] the Cartesian product of E and F
- [E^n] the n-th power of E ($E^2 = E \times E$, $E^n = E \times E^{n-1}$)
- [$f:\ E \to F$] the mapping form E into F, i.e., $E \ni x \mapsto f(x) \in F$
- [I_E or I] the identity mapping in E,i.e., $I(x) = x \quad \forall x \in E$
- [$f \circ g$] for $f:\ F \to G$ and $g:\ E \to F$, $E \ni x \mapsto (f \circ g)(x) = f(g(x)) \in G$ (see *Elementary function*)
- [$f|_A$] the restriction of $f:\ E \to F$ to the subset A of E
- [$\{a_k\}$] column vector with components a_k
- [(a_k)] row vector with components a_k
- [$(a_k)^T$] denotes the transpose of a matrix (a_k), and is $\{a_k\}$
- [$\{a_{ij}\}$] matrix with components a_{ij}, and $(a_{ij})^T = (a_{ji})$

3.1.3 Numbers

For two real numbers a, b

- [a, b] is the interval $\{x \in \mathbb{R}|\ a \leq x \leq b\}$
- $]a, b]$ is the interval $\{x \in \mathbb{R}|\ a < x \leq b\}$
- $[a, b[$ is the interval $\{x \in \mathbb{R}|\ a \leq x < b\}$

- $]a, b[$ is the interval $\{x \in \mathbb{R} |\ a < x < b\}$

3.1.4 Differential Calculus

- [$\partial f/\partial x$] the partial derivative of $f : \mathbb{R}^d \to \mathbb{R}$ with respect to x (`dx(f)`)
- [∇f] the gradient of $f : \Omega \to \mathbb{R}$,i.e., $\nabla f = (\partial f/\partial x,\ \partial f/\partial y)$
- [$\mathrm{div}(\mathbf{f})$ or $\nabla.\mathbf{f}$] the divergence of $\mathbf{f} : \Omega \to \mathbb{R}^d$, i.e., $\mathrm{div}(\mathbf{f}) = \partial f_1/\partial x + \partial f_2/\partial y$
- [Δf] the Laplacian of $f :\ \Omega \to \mathbb{R}$, i.e., $\Delta f = \partial^2 f/\partial x^2 + \partial^2 f/\partial y^2$

3.1.5 Meshes

- [Ω] usually denotes a domain on which PDE is defined
- [Γ] denotes the boundary of Ω,i.e., $\Gamma = \partial\Omega$ (keyword `border`, see *Border*)
- [$\mathcal{T}_h$] the triangulation of Ω, i.e., the set of triangles T_k, where h stands for mesh size (keyword `mesh`, `buildmesh`, see *Mesh Generation*)
- [n_t] the number of triangles in $\mathcal{T}_h$ (get by `Th.nt`)
- [Ω_h] denotes the approximated domain $\Omega_h = \cup_{k=1}^{n_t} T_k$ of Ω. If Ω is polygonal domain, then it will be $\Omega = \Omega_h$
- [Γ_h] the boundary of Ω_h
- [n_v] the number of vertices in $\mathcal{T}_h$ (get by `Th.nv`)
- [n_{be}] the number of boundary element in $\mathcal{T}_h$ (get by `Th.nbe`)
- [$|\Omega_h|$] the measure (area or volume) in $\mathcal{T}_h$ (get by `Th.measure`)
- [$|\partial\Omega_h|$] the measure of the border (length or area) in $\mathcal{T}_h$ (get by `Th.bordermeasure`)
- [h_{min}] the minimum edge size of $\mathcal{T}_h$ (get by `Th.hmin`)
- [h_{max}] the maximum edge size of $\mathcal{T}_h$ (get by `Th.hmax`)
- [$[q^i q^j]$] the segment connecting q^i and q^j
- [$q^{k_1}, q^{k_2}, q^{k_3}$] the vertices of a triangle T_k with anti-clock direction (get the coordinate of q^{k_j} by `(Th[k-1][j-1].x, Th[k-1][j-1].y))`
- [I_Ω] the set $\{i \in \mathbb{N} |\ q^i \notin \Gamma_h\}$

3.1.6 Finite Element Spaces

- [$L^2(\Omega)$] the set $\left\{ w(x,y) \,\middle|\, \int_\Omega |w(x,y)|^2 \mathrm{d}x\mathrm{d}y < \infty \right\}$

$$\text{norm: } \|w\|_{0,\Omega} = \left(\int_\Omega |w(x,y)|^2 \mathrm{d}x\mathrm{d}y \right)^{1/2}$$

$$\text{scalar product: } (v,w) = \int_\Omega vw$$

- [$H^1(\Omega)$] the set $\left\{ w \in L^2(\Omega) \,\middle|\, \int_\Omega \left(|\partial w/\partial x|^2 + |\partial w/\partial y|^2 \right) \mathrm{d}x\mathrm{d}y < \infty \right\}$

$$\text{norm: } \|w\|_{1,\Omega} = \left(\|w\|_{0,\Omega}^2 + \|\nabla u\|_{0.\Omega}^2\right)^{1/2}$$

- $[H^m(\Omega)]$ the set $\left\{ w \in L^2(\Omega) \,\middle|\, \int_\Omega \frac{\partial^{|\alpha|} w}{\partial x^{\alpha_1} \partial y^{\alpha_2}} \in L^2(\Omega) \quad \forall \alpha = (\alpha_1, \alpha_2) \in \mathbb{N}^2,\ |\alpha| = \alpha_1 + \alpha_2 \right\}$

$$\text{scalar product: } (v, w)_{1,\Omega} = \sum_{|\alpha| \le m} \int_\Omega D^\alpha v D^\alpha w$$

- $[H_0^1(\Omega)]$ the set $\left\{ w \in H^1(\Omega) \mid u = 0 \quad \text{on } \Gamma \right\}$

  $[L^2(\Omega)^2]$ denotes $L^2(\Omega) \times L^2(\Omega)$, and also $H^1(\Omega)^2 = H^1(\Omega) \times H^1(\Omega)$
- $[V_h]$ denotes the finite element space created by `fespace Vh(Th, *)` in **FreeFEM** (see *Finite Elements* for `*`)
- $[\Pi_h f]$ the projection of the function f into V_h (`func f=x^2*y^3; Vh v = f;`) means $v = Pi_h(f) * [\{v\}]$ for FE-function v in V_h means the column vector $(v_1, \cdots, v_M)^T$ if $v = v_1\phi_1 + \cdots + v_M\phi_M$, which is shown by `fespace Vh(Th, P2); Vh v; cout << v[] << endl;`

3.2 Mesh Generation

Let us begin with the two important keywords: `border` and `buildmesh`.

3.2.1 Square

The command `square` triangulates the unit square.

The following generates a 4×5 grid in the unit square $[0, 1]^2$. The labels of the boundaries are shown in Fig. 3.1.

```
mesh Th = square(4, 5);
```

Fig. 3.1: Boundary labels of the mesh by `square(10,10)`

To construct a $n \times m$ grid in the rectangle $[x_0, x_1] \times [y_0, y_1]$, proceed as follows:

```
real x0 = 1.2;
real x1 = 1.8;
real y0 = 0;
real y1 = 1;
int n = 5;
real m = 20;
mesh Th = square(n, m, [x0+(x1-x0)*x, y0+(y1-y0)*y]);
```

Note: Adding the named parameter `flags=icase` with icase:

0. will produce a mesh where all quads are split with diagonal $x - y = constant$
1. will produce *Union Jack flag* type of mesh
2. will produce a mesh where all quads are split with diagonal $x + y = constant$
3. same as in case 0, except two corners where the triangles are the same as case 2 to avoid having 3 vertices on the boundary
4. same as in case 2, except two corners where the triangles are the same as case 0 to avoid having 3 vertices on the boundary

```
mesh Th = square(n, m, [x0+(x1-x0)*x, y0+(y1-y0)*y], flags=icase);
```

Note: Adding the named parameter `label=labs` will change the 4 default label numbers to `labs[i-1]`, for example `int[int] labs=[11, 12, 13, 14]`, and adding the named parameter `region=10` will change the region number to 10, for instance (v 3.8).

To see all of these flags at work, check *Square mesh example*:

```
for (int i = 0; i < 5; ++i){
   int[int] labs = [11, 12, 13, 14];
   mesh Th = square(3, 3, flags=i, label=labs, region=10);
   plot(Th, wait=1, cmm="square flags = "+i );
}
```

3.2.2 Border

Boundaries are defined piecewise by parametrized curves. The pieces can only intersect at their endpoints, but it is possible to join more than two endpoints. This can be used to structure the mesh if an area touches a border and create new regions by dividing larger ones:

```
int upper = 1;
int others = 2;
int inner = 3;

border C01(t=0, 1){x=0; y=-1+t; label=upper;}
border C02(t=0, 1){x=1.5-1.5*t; y=-1; label=upper;}
border C03(t=0, 1){x=1.5; y=-t; label=upper;}
border C04(t=0, 1){x=1+0.5*t; y=0; label=others;}
border C05(t=0, 1){x=0.5+0.5*t; y=0; label=others;}
border C06(t=0, 1){x=0.5*t; y=0; label=others;}
border C11(t=0, 1){x=0.5; y=-0.5*t; label=inner;}
```

(continues on next page)

(a) Multiple border ends intersect

(b) Generated mesh

Fig. 3.2: Border

(continued from previous page)

```
border C12(t=0, 1){x=0.5+0.5*t; y=-0.5; label=inner;}
border C13(t=0, 1){x=1; y=-0.5+0.5*t; label=inner;}

int n = 10;
plot(C01(-n) + C02(-n) + C03(-n) + C04(-n) + C05(-n)
  + C06(-n) + C11(n) + C12(n) + C13(n), wait=true);

mesh Th = buildmesh(C01(-n) + C02(-n) + C03(-n) + C04(-n) + C05(-n)
  + C06(-n) + C11(n) + C12(n) + C13(n));

plot(Th, wait=true);

cout << "Part 1 has region number " << Th(0.75, -0.25).region << endl;
cout << "Part 2 has redion number " << Th(0.25, -0.25).region << endl;
```

Borders and mesh are respectively shown in Fig. 3.2a and Fig. 3.2b.

Triangulation keywords assume that the domain is defined as being on the *left* (resp *right*) of its oriented parameterized boundary

$$\Gamma_j = \{(x, y) \,|\, x = \varphi_x(t),\, y = \varphi_y(t),\, a_j \le t \le b_j\}$$

To check the orientation plot $t \mapsto (\varphi_x(t), \varphi_y(t))$, $t_0 \le t \le t_1$. If it is as in Fig. 3.3, then the domain lies on the shaded area, otherwise it lies on the opposite side.

The general expression to define a triangulation with `buildmesh` is

```
mesh Mesh_Name = buildmesh(Gamma1(m1)+...+GammaJ(mj), OptionalParameter);
```

where m_j are positive or negative numbers to indicate how many vertices should be on Γ_j, $\Gamma = \cup_{j=1}^{J} \Gamma_J$, and the optional parameter (see also *References*), separated with a comma, can be:

- `nbvx= int`, to set the maximum number of vertices in the mesh.
- `fixedborder= bool`, to say if the mesh generator can change the boundary mesh or not (by default the boundary mesh can change; beware that with periodic boundary conditions (see. *Finite Element*), it can be dangerous.

The orientation of boundaries can be changed by changing the sign of m_j.

Fig. 3.3: Orientation of the boundary defined by $(\phi_x(t), \phi_y(t))$

(a) Mesh without hole

(b) Mesh with hole

Fig. 3.4: Mesh with a hole

The following example shows how to change the orientation. The example generates the unit disk with a small circular hole, and assigns "1" to the unit disk ("2" to the circle inside). The boundary label **must be non-zero**, but it can also be omitted.

```
1 border a(t=0, 2*pi){x=cos(t); y=sin(t); label=1;}
2 border b(t=0, 2*pi){x=0.3+0.3*cos(t); y=0.3*sin(t); label=2;}
3 plot(a(50) + b(30)); //to see a plot of the border mesh
4 mesh Thwithouthole = buildmesh(a(50) + b(30));
5 mesh Thwithhole = buildmesh(a(50) + b(-30));
6 plot(Thwithouthole, ps="Thwithouthole.eps");
7 plot(Thwithhole, ps="Thwithhole.eps");
```

Note: Notice that the orientation is changed by `b(-30)` in the 5th line. In the 7th line, `ps="fileName"` is used to generate a postscript file with identification shown on the figure.

Note: Borders are evaluated only at the time `plot` or `buildmesh` is called so the global variables are defined at this time. In this case, since r is changed between the two border calls, the following code will not work because the first border will be computed with r=0.3:

```
real r=1;
border a(t=0, 2*pi){x=r*cos(t); y=r*sin(t); label=1;}
r=0.3;
border b(t=0, 2*pi){x=r*cos(t); y=r*sin(t); label=1;}
mesh Thwithhole = buildmesh(a(50) + b(-30)); // bug (a trap) because
    // the two circles have the same radius = :math:`0.3`
```

3.2.3 Multi-Border

Sometimes it can be useful to make an array of the border, but unfortunately it is incompatible with the **FreeFEM** syntax. To bypass this problem, if the number of segments of the discretization n is an array, we make an implicit loop on all of the values of the array, and the index variable i of the loop is defined after the parameter definition, like in `border a(t=0, 2*pi; i)...`

A first very small example:

```
border a(t=0, 2*pi; i){x=(i+1)*cos(t); y=(i+1)*sin(t); label=1;}
int[int] nn = [10, 20, 30];
plot(a(nn)); //plot 3 circles with 10, 20, 30 points
```

And a more complex example to define a square with small circles:

```
real[int] xx = [0, 1, 1, 0],
          yy = [0, 0, 1, 1];
//radius, center of the 4 circles
real[int] RC = [0.1, 0.05, 0.05, 0.1],
          XC = [0.2, 0.8, 0.2, 0.8],
          YC = [0.2, 0.8, 0.8, 0.2];
int[int] NC = [-10,-11,-12,13]; //list number of :math:`\pm` segments of the 4␣
↪circles borders

border bb(t=0, 1; i)
{
    // i is the index variable of the multi border loop
    int ii = (i+1)%4;
    real t1 = 1-t;
    x = xx[i]*t1 + xx[ii]*t;
    y = yy[i]*t1 + yy[ii]*t;
    label = 0;
}

border cc(t=0, 2*pi; i)
{
    x = RC[i]*cos(t) + XC[i];
    y = RC[i]*sin(t) + YC[i];
    label = i + 1;
}
int[int] nn = [4, 4, 5, 7]; //4 border, with 4, 4, 5, 7 segment respectively
plot(bb(nn), cc(NC), wait=1);
```

(continues on next page)

(continued from previous page)

```
mesh th = buildmesh(bb(nn) + cc(NC));
plot(th, wait=1);
```

3.2.4 Data Structures and Read/Write Statements for a Mesh

Users who want to read a triangulation made elsewhere should see the structure of the file generated below:

```
border C(t=0, 2*pi){x=cos(t); y=sin(t);}
mesh Th = buildmesh(C(10));
savemesh(Th, "mesh.msh");
```

The mesh is shown on Fig. 3.5.

The information about `Th` are saved in the file `mesh.msh` whose structure is shown on Table 3.1.

There, n_v denotes the number of vertices, n_t the number of triangles and n_s the number of edges on boundary.

For each vertex q^i, $i = 1, \cdots, n_v$, denoted by (q^i_x, q^i_y) the x-coordinate and y-coordinate.

Each triangle $T_k, k = 1, \cdots, n_t$ has three vertices q^{k_1}, q^{k_2}, q^{k_3} that are oriented counter-clockwise.

The boundary consists of 10 lines L_i, $i = 1, \cdots, 10$ whose end points are q^{i_1}, q^{i_2}.

Fig. 3.5: Mesh by `buildmesh(C(10))`

In the Fig. 3.5, we have the following.

$n_v = 14, n_t = 16, n_s = 10$

$q^1 = (-0.309016994375, 0.951056516295)$

$\cdots$

$q^{14} = (-0.309016994375, -0.951056516295)$

The vertices of T_1 are q^9, q^{12}, q^{10}.

$\cdots$

The vertices of T_{16} are q^9, q^{10}, q^6.

The edge of the 1st side L_1 are q^6, q^5.

...

The edge of the 10th side L_{10} are q^{10}, q^6.

Table 3.1: The structure of `mesh_sample.msh`

Content of the file	Explanation
14 16 10	n_v n_t n_e
-0.309016994375 0.951056516295 1 0.309016994375 0.951056516295 1 ... -0.309016994375 -0.951056516295 1	q_x^1 q_y^1 boundary label $= 1$ q_x^2 q_y^2 boundary label $= 1$ ... q_x^{14} q_y^{14} boundary label $= 1$
9 12 10 0 5 9 6 0 ... 9 10 6 0	1_1 1_2 1_3 region label $= 0$ 2_1 2_2 2_3 region label $= 0$ ... 16_1 16_2 16_3 region label $= 0$
6 5 1 5 2 1 ... 10 6 1	1_1 1_2 boundary label $= 1$ 2_1 2_2 boundary label $= 1$ ... 10_1 10_2 boundary label $= 1$

In **FreeFEM** there are many mesh file formats available for communication with other tools such as `emc2`, `modulef`, ... (see *Mesh format chapter*).

The extension of a file implies its format. More details can be found on the file format .msh in the article by F. Hecht "bamg : a bidimensional anisotropic mesh generator" [HECHT1998_2].

A mesh file can be read into **FreeFEM** except that the names of the borders are lost and only their reference numbers are kept. So these borders have to be referenced by the number which corresponds to their order of appearance in the program, unless this number is overwritten by the keyword `label`. Here are some examples:

```
border floor(t=0, 1){x=t; y=0; label=1;}
border right(t=0, 1){x=1; y=t; label=5;}
border ceiling(t=1, 0){x=t; y=1; label=5;}
border left(t=1, 0){x=0; y=t; label=5;}

int n = 10;
mesh th = buildmesh(floor(n) + right(n) + ceiling(n) + left(n));
savemesh(th, "toto.am_fmt"); //"formatted Marrocco" format
savemesh(th, "toto.Th"); //"bamg"-type mesh
savemesh(th, "toto.msh"); //freefem format
savemesh(th, "toto.nopo"); //modulef format
mesh th2 = readmesh("toto.msh"); //read the mesh
```

```
// Parameters
int n = 10;

// Mesh
border floor(t=0, 1){x=t; y=0; label=1;};
border right(t=0, 1){x=1; y=t; label=5;};
border ceiling(t=1, 0){x=t; y=1; label=5;};
border left(t=1, 0){x=0; y=t; label=5;};

mesh th = buildmesh(floor(n) + right(n) + ceiling(n) + left(n));
```

(continues on next page)

(continued from previous page)

```

//save mesh in different formats
savemesh(th, "toto.am_fmt"); // format "formated Marrocco"
savemesh(th, "toto.Th"); // format database db mesh "bamg"
savemesh(th, "toto.msh"); // format freefem
savemesh(th, "toto.nopo"); // modulef format

// Fespace
fespace femp1(th, P1);
femp1 f = sin(x)*cos(y);
femp1 g;

//save the fespace function in a file
{
  ofstream file("f.txt");
  file << f[] << endl;
} //the file is automatically closed at the end of the block
//read a file and put it in a fespace function
{
  ifstream file("f.txt");
  file >> g[] ;
}//the file is equally automatically closed

// Plot
plot(g);

// Mesh 2
//read the mesh for freefem format saved mesh
mesh th2 = readmesh("toto.msh");

// Fespace 2
fespace Vh2(th2, P1);
Vh2 u, v;

// Problem
//solve:
//  $u + \Delta u = g$ in $\Omega $
//  $u=0$ on $\Gamma_1$
//  $\frac{\partial u }{\partial n} = g$ on $\Gamma_2$
solve Problem(u, v)
  = int2d(th2)(
      u*v
    - dx(u)*dx(v)
    - dy(u)*dy(v)
  )
  + int2d(th2)(
    - g*v
  )
  + int1d(th2, 5)(
      g*v
  )
  + on(1, u=0)
  ;

// Plot
plot(th2, u);
```

3.2.5 Mesh Connectivity and data

The following example explains methods to obtain mesh information.

```
// Mesh
mesh Th = square(2, 2);

cout << "// Get data of the mesh" << endl;
{
    int NbTriangles = Th.nt;
    real MeshArea = Th.measure;
    real BorderLenght = Th.bordermeasure;

    cout << "Number of triangle(s) = " << NbTriangles << endl;
    cout << "Mesh area = " << MeshArea << endl;
    cout << "Border length = " << BorderLenght << endl;

    // Th(i) return the vextex i of Th
    // Th[k] return the triangle k of Th
    // Th[k][i] return the vertex i of the triangle k of Th
    for (int i = 0; i < NbTriangles; i++)
        for (int j = 0; j < 3; j++)
            cout << i << " " << j << " - Th[i][j] = " << Th[i][j]
                 << ", x = " << Th[i][j].x
                 << ", y= " << Th[i][j].y
                 << ", label=" << Th[i][j].label << endl;
}

cout << "// Hack to get vertex coordinates" << endl;
{
    fespace femp1(Th, P1);
    femp1 Thx=x,Thy=y;

    int NbVertices = Th.nv;
    cout << "Number of vertices = " << NbVertices << endl;

    for (int i = 0; i < NbVertices; i++)
        cout << "Th(" << i << ") : " << Th(i).x << " " << Th(i).y << " " << Th(i).
↪label
             << endl << "\told method: " << Thx[][i] << " " << Thy[][i] << endl;
}

cout << "// Method to find information of point (0.55,0.6)" << endl;
{
    int TNumber = Th(0.55, 0.6).nuTriangle; //the triangle number
    int RLabel = Th(0.55, 0.6).region; //the region label

    cout << "Triangle number in point (0.55, 0.6): " << TNumber << endl;
    cout << "Region label in point (0.55, 0.6): " << RLabel << endl;
}

cout << "// Information of triangle" << endl;
{
    int TNumber = Th(0.55, 0.6).nuTriangle;
    real TArea = Th[TNumber].area; //triangle area
    real TRegion = Th[TNumber].region; //triangle region
    real TLabel = Th[TNumber].label; //triangle label, same as region for triangles

```

(continues on next page)

(continued from previous page)

```
    cout << "Area of triangle " << TNumber << ": " << TArea << endl;
    cout << "Region of triangle " << TNumber << ": " << TRegion << endl;
    cout << "Label of triangle " << TNumber << ": " << TLabel << endl;
}

cout << "// Hack to get a triangle containing point x, y or region number (old method)" << endl;
{
    fespace femp0(Th, P0);
    femp0 TNumbers; //a P0 function to get triangle numbering
    for (int i = 0; i < Th.nt; i++)
        TNumbers[][i] = i;
    femp0 RNumbers = region; //a P0 function to get the region number

    int TNumber = TNumbers(0.55, 0.6); // Number of the triangle containing (0.55, 0,6)
    int RNumber = RNumbers(0.55, 0.6); // Number of the region containing (0.55, 0,6)

    cout << "Point (0.55,0,6) :" << endl;
    cout << "\tTriangle number = " << TNumber << endl;
    cout << "\tRegion number = " << RNumber << endl;
}

cout << "// New method to get boundary information and mesh adjacent" << endl;
{
    int k = 0;
    int l=1;
    int e=1;

    // Number of boundary elements
    int NbBoundaryElements = Th.nbe;
    cout << "Number of boundary element = " << NbBoundaryElements << endl;
    // Boundary element k in {0, ..., Th.nbe}
    int BoundaryElement = Th.be(k);
    cout << "Boundary element " << k << " = " << BoundaryElement << endl;
    // Vertice l in {0, 1} of boundary element k
    int Vertex = Th.be(k)[l];
    cout << "Vertex " << l << " of boundary element " << k << " = " << Vertex << endl;
    // Triangle containg the boundary element k
    int Triangle = Th.be(k).Element;
    cout << "Triangle containing the boundary element " << k << " = " << Triangle << endl;
    // Triangle egde nubmer containing the boundary element k
    int Edge = Th.be(k).whoinElement;
    cout << "Triangle edge number containing the boundary element " << k << " = " << Edge << endl;
    // Adjacent triangle of the triangle k by edge e
    int Adjacent = Th[k].adj(e); //The value of e is changed to the corresponding edge in the adjacent triangle
    cout << "Adjacent triangle of the triangle " << k << " by edge " << e << " = " << Adjacent << endl;
    cout << "\tCorresponding edge = " << e << endl;
    // If there is no adjacent triangle by edge e, the same triangle is returned
    //Th[k] == Th[k].adj(e)
    // Else a different triangle is returned
    //Th[k] != Th[k].adj(e)
}
```

(continues on next page)

(continued from previous page)

```

cout << "// Print mesh connectivity " << endl;
{
    int NbTriangles = Th.nt;
    for (int k = 0; k < NbTriangles; k++)
        cout << k << " : " << int(Th[k][0]) << " " << int(Th[k][1])
             << " " << int(Th[k][2])
             << ", label " << Th[k].label << endl;

    for (int k = 0; k < NbTriangles; k++)
        for (int e = 0, ee; e < 3; e++)
            //set ee to e, and ee is change by method adj,
            cout << k << " " << e << " <=> " << int(Th[k].adj((ee=e))) << " " << ee
                 << ", adj: " << (Th[k].adj((ee=e)) != Th[k]) << endl;

    int NbBoundaryElements = Th.nbe;
    for (int k = 0; k < NbBoundaryElements; k++)
        cout << k << " : " << Th.be(k)[0] << " " << Th.be(k)[1]
             << " , label " << Th.be(k).label
             << ", triangle " << int(Th.be(k).Element)
             << " " << Th.be(k).whoinElement << endl;

    real[int] bb(4);
    boundingbox(Th, bb);
    // bb[0] = xmin, bb[1] = xmax, bb[2] = ymin, bb[3] =ymax
    cout << "boundingbox:" << endl;
    cout << "xmin = " << bb[0]
         << ", xmax = " << bb[1]
         << ", ymin = " << bb[2]
         << ", ymax = " << bb[3] << endl;
}
```

The output is:

```
// Get data of the mesh
Number of triangle = 8
Mesh area = 1
Border length = 4
0 0 - Th[i][j] = 0, x = 0, y= 0, label=4
0 1 - Th[i][j] = 1, x = 0.5, y= 0, label=1
0 2 - Th[i][j] = 4, x = 0.5, y= 0.5, label=0
1 0 - Th[i][j] = 0, x = 0, y= 0, label=4
1 1 - Th[i][j] = 4, x = 0.5, y= 0.5, label=0
1 2 - Th[i][j] = 3, x = 0, y= 0.5, label=4
2 0 - Th[i][j] = 1, x = 0.5, y= 0, label=1
2 1 - Th[i][j] = 2, x = 1, y= 0, label=2
2 2 - Th[i][j] = 5, x = 1, y= 0.5, label=2
3 0 - Th[i][j] = 1, x = 0.5, y= 0, label=1
3 1 - Th[i][j] = 5, x = 1, y= 0.5, label=2
3 2 - Th[i][j] = 4, x = 0.5, y= 0.5, label=0
4 0 - Th[i][j] = 3, x = 0, y= 0.5, label=4
4 1 - Th[i][j] = 4, x = 0.5, y= 0.5, label=0
4 2 - Th[i][j] = 7, x = 0.5, y= 1, label=3
5 0 - Th[i][j] = 3, x = 0, y= 0.5, label=4
5 1 - Th[i][j] = 7, x = 0.5, y= 1, label=3
5 2 - Th[i][j] = 6, x = 0, y= 1, label=4
6 0 - Th[i][j] = 4, x = 0.5, y= 0.5, label=0
```

(continues on next page)

(continued from previous page)

```
6 1 - Th[i][j] = 5, x = 1, y= 0.5, label=2
6 2 - Th[i][j] = 8, x = 1, y= 1, label=3
7 0 - Th[i][j] = 4, x = 0.5, y= 0.5, label=0
7 1 - Th[i][j] = 8, x = 1, y= 1, label=3
7 2 - Th[i][j] = 7, x = 0.5, y= 1, label=3
// Hack to get vertex coordinates
Number of vertices = 9
Th(0) : 0 0 4
   old method: 0 0
Th(1) : 0.5 0 1
   old method: 0.5 0
Th(2) : 1 0 2
   old method: 1 0
Th(3) : 0 0.5 4
   old method: 0 0.5
Th(4) : 0.5 0.5 0
   old method: 0.5 0.5
Th(5) : 1 0.5 2
   old method: 1 0.5
Th(6) : 0 1 4
   old method: 0 1
Th(7) : 0.5 1 3
   old method: 0.5 1
Th(8) : 1 1 3
   old method: 1 1
// Method to find the information of point (0.55,0.6)
Triangle number in point (0.55, 0.6): 7
Region label in point (0.55, 0.6): 0
// Information of a triangle
Area of triangle 7: 0.125
Region of triangle 7: 0
Label of triangle 7: 0
// Hack to get a triangle containing point x, y or region number (old method)
Point (0.55,0,6) :
   Triangle number = 7
   Region number = 0
// New method to get boundary information and mesh adjacent
Number of boundary element = 8
Boundary element 0 = 0
Vertex 1 of boundary element 0 = 1
Triangle containing the boundary element 0 = 0
Triangle edge number containing the boundary element 0 = 2
Adjacent triangle of the triangle 0 by edge 1 = 1
   Corresponding edge = 2
// Print mesh connectivity
0 : 0 1 4, label 0
1 : 0 4 3, label 0
2 : 1 2 5, label 0
3 : 1 5 4, label 0
4 : 3 4 7, label 0
5 : 3 7 6, label 0
6 : 4 5 8, label 0
7 : 4 8 7, label 0
0 0 <=> 3 1, adj: 1
0 1 <=> 1 2, adj: 1
0 2 <=> 0 2, adj: 0
1 0 <=> 4 2, adj: 1
```

(continues on next page)

(continued from previous page)

```
1 1 <=> 1 1, adj: 0
1 2 <=> 0 1, adj: 1
2 0 <=> 2 0, adj: 0
2 1 <=> 3 2, adj: 1
2 2 <=> 2 2, adj: 0
3 0 <=> 6 2, adj: 1
3 1 <=> 0 0, adj: 1
3 2 <=> 2 1, adj: 1
4 0 <=> 7 1, adj: 1
4 1 <=> 5 2, adj: 1
4 2 <=> 1 0, adj: 1
5 0 <=> 5 0, adj: 0
5 1 <=> 5 1, adj: 0
5 2 <=> 4 1, adj: 1
6 0 <=> 6 0, adj: 0
6 1 <=> 7 2, adj: 1
6 2 <=> 3 0, adj: 1
7 0 <=> 7 0, adj: 0
7 1 <=> 4 0, adj: 1
7 2 <=> 6 1, adj: 1
0 : 0 1 , label 1, triangle 0 2
1 : 1 2 , label 1, triangle 2 2
2 : 2 5 , label 2, triangle 2 0
3 : 5 8 , label 2, triangle 6 0
4 : 6 7 , label 3, triangle 5 0
5 : 7 8 , label 3, triangle 7 0
6 : 0 3 , label 4, triangle 1 1
7 : 3 6 , label 4, triangle 5 1
boundingbox:
xmin = 0, xmax = 1, ymin = 0, ymax = 1
```

The real characteristic function of a mesh `Th` is `chi(Th)` in 2D and 3D where:

`chi(Th)(P)=1` if $P \in Th$

`chi(Th)(P)=0` if $P \notin Th$

3.2.6 The keyword "triangulate"

FreeFEM is able to build a triangulation from a set of points. This triangulation is a Delaunay mesh of the convex hull of the set of points. It can be useful to build a mesh from a table function.

The coordinates of the points and the value of the table function are defined separately with rows of the form: `x y f(x,y)` in a file such as:

```
0.51387 0.175741 0.636237
0.308652 0.534534 0.746765
0.947628 0.171736 0.899823
0.702231 0.226431 0.800819
0.494773 0.12472 0.580623
0.0838988 0.389647 0.456045
...............
```

The third column of each line is left untouched by the `triangulate` command. But you can use this third value to define a table function with rows of the form: `x y f(x,y)`.

(a) Delaunay mesh of the convex hull of point set in file xy

(b) Isolvalue of table function

Fig. 3.6: Triangulate

The following example shows how to make a mesh from the file `xyf` with the format stated just above. The command `triangulate` only uses the 1st and 2nd columns.

```
// Build the Delaunay mesh of the convex hull
mesh Thxy=triangulate("xyf"); //points are defined by the first 2 columns of file `xyf`

// Plot the created mesh
plot(Thxy);

// Fespace
fespace Vhxy(Thxy, P1);
Vhxy fxy;

// Reading the 3rd column to define the function fxy
{
    ifstream file("xyf");
    real xx, yy;
    for(int i = 0; i < fxy.n; i++)
        file >> xx >> yy >> fxy[][i]; //to read third row only.
                                      //xx and yy are just skipped
}

// Plot
plot(fxy);
```

One new way to build a mesh is to have two arrays: one for the x values and the other for the y values.

```
//set two arrays for the x's and y's
Vhxy xx=x, yy=y;
//build the mesh
mesh Th = triangulate(xx[], yy[]);
```

(a) The empty mesh with boundary

(b) An empty mesh defined from a pseudo region numbering of triangle

Fig. 3.7: Empty mesh

Boundary FEM Spaces Built as Empty Meshes

To define a Finite Element space on a boundary, we came up with the idea of a mesh with no internal points (called empty mesh). It can be useful to handle Lagrange multipliers in mixed and mortar methods.

So the function `emptymesh` removes all the internal points of a mesh except points on internal boundaries.

```
{
    border a(t=0, 2*pi){x=cos(t); y=sin(t); label=1;}
    mesh Th = buildmesh(a(20));
    Th = emptymesh(Th);
    plot(Th);
}
```

It is also possible to build an empty mesh of a pseudo subregion with `emptymesh(Th, ssd)` using the set of edges from the mesh `Th`; an edge e is in this set when, with the two adjacent triangles $e = t1 \cap t2$ and $ssd[T1] \neq ssd[T2]$ where ssd refers to the pseudo region numbering of triangles, they are stored in the `int[int]` array of size "the number of triangles".

```
{
    mesh Th = square(10, 10);
    int[int] ssd(Th.nt);
    //build the pseudo region numbering
    for(int i = 0; i < ssd.n; i++){
        int iq = i/2; //because 2 triangles per quad
        int ix = iq%10;
        int iy = iq/10;
        ssd[i] = 1 + (ix>=5) + (iy>=5)*2;
    }
    //build emtpy with all edges $e=T1 \cap T2$ and $ssd[T1] \neq ssd[T2]$
    Th = emptymesh(Th, ssd);
    //plot
    plot(Th);
    savemesh(Th, "emptymesh.msh");
}
```

Remeshing

3.2.7 Movemesh

Meshes can be translated, rotated, and deformed by `movemesh`; this is useful for elasticity to watch the deformation due to the displacement $\mathbf{\Phi}(x, y) = (\Phi_1(x, y), \Phi_2(x, y))$ of shape.

It is also useful to handle free boundary problems or optimal shape problems.

If Ω is triangulated as $T_h(\Omega)$, and $\mathbf{\Phi}$ is a displacement vector then $\mathbf{\Phi}(T_h)$ is obtained by:

```
mesh Th = movemesh(Th,[Phi1, Phi2]);
```

Sometimes the transformed mesh is invalid because some triangles have flipped over (meaning it now has a negative area). To spot such problems, one may check the minimum triangle area in the transformed mesh with `checkmovemesh` before any real transformation.

For example:

$$\begin{array}{rcl} \Phi_1(x, y) & = & x + k * \sin(y * \pi)/10) \\ \Phi_2(x, y) & = & y + k * \cos(y\pi)/10) \end{array}$$

for a big number $k > 1$.

```
verbosity = 4;

// Parameters
real coef = 1;

// Mesh
border a(t=0, 1){x=t; y=0; label=1;};
border b(t=0, 0.5){x=1; y=t; label=1;};
border c(t=0, 0.5){x=1-t; y=0.5; label=1;};
border d(t=0.5, 1){x=0.5; y=t; label=1;};
border e(t=0.5, 1){x=1-t; y=1; label=1;};
border f(t=0, 1){x=0; y=1-t; label=1;};
mesh Th = buildmesh(a(6) + b(4) + c(4) + d(4) + e(4) + f(6));
plot(Th, wait=true, fill=true, ps="Lshape.eps");

// Function
func uu = sin(y*pi)/10;
func vv = cos(x*pi)/10;

// Checkmovemesh
real minT0 = checkmovemesh(Th, [x, y]); //return the min triangle area
while(1){ // find a correct move mesh
    real minT = checkmovemesh(Th, [x+coef*uu, y+coef*vv]);
    if (minT > minT0/5) break; //if big enough
    coef /= 1.5;
}

// Movemesh
Th = movemesh(Th, [x+coef*uu, y+coef*vv]);
plot(Th, wait=true, fill=true, ps="MovedMesh.eps");
```

Note: Consider a function u defined on a mesh `Th`. A statement like `Th=movemesh(Th...)` does not change u and so the old mesh still exists. It will be destroyed when no function uses it. A statement like $u = u$ redefines u on

(a) L-shape **(b)** Moved L-shape

Fig. 3.8: Move mesh

the new mesh `Th` with interpolation and therefore destroys the old `Th`, if u was the only function using it.

Now, we give an example of moving a mesh with a Lagrangian function u defined on the moving mesh.

```
// Parameters
int nn = 10;
real dt = 0.1;

// Mesh
mesh Th = square(nn, nn);

// Fespace
fespace Vh(Th, P1);
Vh u=y;

// Loop
real t=0;
for (int i = 0; i < 4; i++){
    t = i*dt;
    Vh f=x*t;
    real minarea = checkmovemesh(Th, [x, y+f]);
    if (minarea > 0) //movemesh will be ok
    Th = movemesh(Th, [x, y+f]);

    cout << " Min area = " << minarea << endl;

    real[int] tmp(u[].n);
    tmp = u[]; //save the value
    u = 0;//to change the FEspace and mesh associated with u
    u[] = tmp;//set the value of u without any mesh update
    plot(Th, u, wait=true);
}
// In this program, since u is only defined on the last mesh, all the
```

(continues on next page)

(continued from previous page)

```
// previous meshes are deleted from memory.
```

Regular Triangulation: hTriangle

For a set S, we define the diameter of S by

$$\text{diam}(S) = \sup\{|\mathbf{x} - \mathbf{y}|;\ \mathbf{x},\ \mathbf{y} \in S\}$$

The sequence $\{\mathcal{T}_h\}_{h\to 0}$ of Ω is called *regular* if they satisfy the following:

1. $\lim_{h\to 0} \max\{\text{diam}(T_k)|\ T_k \in \mathcal{T}_h\} = 0$
2. There is a number $\sigma > 0$ independent of h such that $\frac{\rho(T_k)}{\text{diam}(T_k)} \geq \sigma$ for all $T_k \in \mathcal{T}_h$ where $\rho(T_k)$ are the diameter of the inscribed circle of T_k.

We put $h(\mathcal{T}_h) = \max\{\text{diam}(T_k)|\ T_k \in \mathcal{T}_h\}$, which is obtained by

```
mesh Th = ......;
fespace Ph(Th, P0);
Ph h = hTriangle;
cout << "size of mesh = " << h[].max << endl;
```

Adaptmesh

The function:

$$f(x, y) = 10.0x^3 + y^3 + \tan^{-1}[\varepsilon/(\sin(5.0y) - 2.0x)],\ \varepsilon = 0.0001$$

sharply varies in value and the initial mesh given by one of the commands in the *Mesh Generation part* cannot reflect its sharp variations.

```
// Parameters
real eps = 0.0001;
real h = 1;
real hmin = 0.05;
func f = 10.0*x^3 + y^3 + h*atan2(eps, sin(5.0*y)-2.0*x);

// Mesh
mesh Th = square(5, 5, [-1+2*x, -1+2*y]);

// Fespace
fespace Vh(Th,P1);
Vh fh = f;
plot(fh);

// Adaptmesh
for (int i = 0; i < 2; i++){
    Th = adaptmesh(Th, fh);
    fh = f; //old mesh is deleted
    plot(Th, fh, wait=true);
}
```

FreeFEM uses a variable metric/Delaunay automatic meshing algorithm.

The command:

Fig. 3.9: 3D graphs for the initial mesh and 1st and 2nd mesh adaptations

```
mesh ATh = adaptmesh(Th, f);
```

create the new mesh `ATh` adapted to the Hessian

$$D^2 f = (\partial^2 f/\partial x^2, \partial^2 f/\partial x \partial y, \partial^2 f/\partial y^2)$$

of a function (formula or FE-function).

Mesh adaptation is a very powerful tool when the solution of a problem varies locally and sharply.

Here we solve the *Poisson's problem*, when $f = 1$ and Ω is a L-shape domain.

Tip: The solution has the singularity $r^{3/2}$, $r = |x - \gamma|$ at the point γ of the intersection of two lines bc and bd (see Fig. 3.10a).

```
// Parameters
real error = 0.1;

// Mesh
border ba(t=0, 1){x=t; y=0; label=1;}
border bb(t=0, 0.5){x=1; y=t; label=1;}
border bc(t=0, 0.5){x=1-t; y=0.5; label=1;}
border bd(t=0.5, 1){x=0.5; y=t; label=1;}
border be(t=0.5, 1){x=1-t; y=1; label=1;}
border bf(t=0, 1){x=0; y=1-t; label=1;}
mesh Th = buildmesh(ba(6) + bb(4) + bc(4) + bd(4) + be(4) + bf(6));

// Fespace
```

(continues on next page)

(a) L-shape domain and its boundary name

(b) Final solution after 4-times adaptation

Fig. 3.10: Mesh adaptation

(continued from previous page)

```
fespace Vh(Th, P1);
Vh u, v;

// Function
func f = 1;

// Problem
problem Poisson(u, v, solver=CG, eps=1.e-6)
    = int2d(Th)(
          dx(u)*dx(v)
        + dy(u)*dy(v)
    )
    - int2d(Th)(
          f*v
    )
    + on(1, u=0);

// Adaptmesh loop
for (int i = 0; i < 4; i++){
    Poisson;
    Th = adaptmesh(Th, u, err=error);
    error = error/2;
}

// Plot
plot(u);
```

To speed up the adaptation, the default parameter `err` of `adaptmesh` is changed by hand; it specifies the required precision, so as to make the new mesh finer or coarser.

The problem is coercive and symmetric, so the linear system can be solved with the conjugate gradient method (parameter `solver=CG`) with the stopping criteria on the residual, here `eps=1.e-6`).

By `adaptmesh`, the slope of the final solution is correctly computed near the point of intersection of bc and bd as in Fig. 3.10b.

This method is described in detail in [HECHT1998]. It has a number of default parameters which can be modified.

If `f1,f2` are functions and `thold, Thnew` are meshes:

```
Thnew = adaptmesh(Thold, f1 ... );
Thnew = adaptmesh(Thold, f1,f2 ... ]);
Thnew = adaptmesh(Thold, [f1,f2] ... );
```

The additional parameters of `adaptmesh` are:

See *Reference part* for more inforamtions

- **`hmin`= Minimum edge size.** Its default is related to the size of the domain to be meshed and the precision of the mesh generator.
- **`hmax`= Maximum edge size.** It defaults to the diameter of the domain to be meshed.
- `err`= P_1 interpolation error level (0.01 is the default).
- **`errg`= Relative geometrical error.** By default this error is 0.01, and in any case it must be lower than $1/\sqrt{2}$. Meshes created with this option may have some edges smaller than the `-hmin` due to geometrical constraints.
- `nbvx`= Maximum number of vertices generated by the mesh generator (9000 is the default).
- `nbsmooth`= number of iterations of the smoothing procedure (5 is the default).
- `nbjacoby`= number of iterations in a smoothing procedure during the metric construction, 0 means no smoothing, 6 is the default.
- **`ratio`= ratio for a prescribed smoothing on the metric.** If the value is 0 or less than 1.1 no smoothing is done on the metric. 1.8 is the default. If `ratio > 1.1`, the speed of mesh size variations is bounded by $log(\texttt{ratio})$.

  Note: As `ratio` gets closer to 1, the number of generated vertices increases. This may be useful to control the thickness of refined regions near shocks or boundary layers.

- `omega`= relaxation parameter for the smoothing procedure. 1.0 is the default.
- `iso`= If true, forces the metric to be isotropic. `false` is the default.
- **`abserror`= If false, the metric is evaluated using the criteria of equi-repartion of relative error.** `false` is the default. In this case the metric is defined by:

$$\mathcal{M} = \left(\frac{1}{\texttt{err coef}^2} \quad \frac{|\mathcal{H}|}{max(\texttt{CutOff}, |\eta|)} \right)^p$$

  Otherwise, the metric is evaluated using the criteria of equi-distribution of errors. In this case the metric is defined by:

$$\mathcal{M} = \left(\frac{1}{\texttt{err coef}^2} \quad \frac{|\mathcal{H}|}{\sup(\eta) - \inf(\eta)} \right)^p .$$

- `cutoff`= lower limit for the relative error evaluation. 1.0e-6 is the default.
- **`verbosity`= informational messages level (can be chosen between 0 and ∞).** Also changes the value of the global variable verbosity (obsolete).
- `inquire`= To inquire graphically about the mesh. `false` is the default.

- **`splitpbedge=` If true, splits all internal edges in half with two boundary vertices.** `true` is the default.
- **`maxsubdiv=` Changes the metric such that the maximum subdivision of a background edge is bound by `val`.** Always limited by 10, and 10 is also the default.
- **`rescaling=` if true, the function, with respect to which the mesh is adapted, is rescaled to be between 0 and 1.** `true` is the default.
- **`keepbackvertices=` if true, tries to keep as many vertices from the original mesh as possible.** `true` is the default.
- **`IsMetric=` if true, the metric is defined explicitly.** `false` is the default. If the 3 functions m_{11}, m_{12}, m_{22} are given, they directly define a symmetric matrix field whose Hessian is computed to define a metric. If only one function is given, then it represents the isotropic mesh size at every point.

  For example, if the partial derivatives `fxx` ($= \partial^2 f/\partial x^2$), `fxy` ($= \partial^2 f/\partial x \partial y$), `fyy` ($= \partial^2 f/\partial y^2$) are given, we can set `Th = adaptmesh(Th, fxx, fxy, fyy, IsMetric=1, nbvx=10000, hmin=hmin);`
- **`power=` exponent power of the Hessian used to compute the metric.** 1 is the default.
- **`thetamax=` minimum corner angle in degrees.** Default is $10°$ where the corner is ABC and the angle is the angle of the two vectors AB, BC, (0 imply no corner, 90 imply perpendicular corner, ...).
- **`splitin2=` boolean value.** If true, splits all triangles of the final mesh into 4 sub-triangles.
- **`metric=` an array of 3 real arrays to set or get metric data information.** The size of these three arrays must be the number of vertices. So if `m11,m12,m22` are three P1 finite elements related to the mesh to adapt, you can write: `metric=[m11[],m12[],m22[]]` (see file `convect-apt.edp` for a full example)
- `nomeshgeneration=` If true, no adapted mesh is generated (useful to compute only a metric).
- **`periodic=` Writing `periodic=[[4,y],[2,y],[1,x],[3,x]];` builds an adapted periodic mesh.** The sample builds a biperiodic mesh of a square. (see *periodic finite element spaces*, and see *the Sphere example* for a full example)

We can use the command `adaptmesh` to build a uniform mesh with a constant mesh size. To build a mesh with a constant mesh size equal to $\frac{1}{30}$ try:

```
mesh Th=square(2, 2); //the initial mesh
plot(Th, wait=true, ps="square-0.eps");

Th = adaptmesh(Th, 1./30., IsMetric=1, nbvx=10000);
plot(Th, wait=true, ps="square-1.eps");

Th = adaptmesh(Th, 1./30., IsMetric=1, nbvx=10000); //More the one time du to
Th = adaptmesh(Th, 1./30., IsMetric=1, nbvx=10000); //Adaptation bound `maxsubdiv=`
plot(Th, wait=true, ps="square-2.eps");
```

Trunc

Two operators have been introduced to remove triangles from a mesh or to divide them. Operator `trunc` has two parameters:

- `label=` sets the label number of new boundary item, one by default.
- **`split=` sets the level n of triangle splitting.** Each triangle is split in $n \times n$, one by default.

To create the mesh `Th3` where all triangles of a mesh `Th` are split in 3×3, just write:

(a) Initial mesh

(b) First iteration

(c) Last iteration

Fig. 3.11: Mesh adaptation

(a) Mesh of support the function P1 number 0, split in 5×5 **(b)** Mesh of support the function P1 number 6, split in 5×5

Fig. 3.12: Trunc

```
mesh Th3 = trunc(Th, 1, split=3);
```

The following example construct all "trunced" meshes to the support of the basic function of the space `Vh` (cf. `abs(u)>0`), split all the triangles in 5×5, and put a label number to 2 on a new boundary.

```
// Mesh
mesh Th = square(3, 3);

// Fespace
fespace Vh(Th, P1);
Vh u=0;

// Loop on all degrees of freedom
int n=u.n;
for (int i = 0; i < n; i++){
    u[][i] = 1; // The basis function i
    plot(u, wait=true);
    mesh Sh1 = trunc(Th, abs(u)>1.e-10, split=5, label=2);
    plot(Th, Sh1, wait=true, ps="trunc"+i+".eps");
    u[][i] = 0; // reset
}
```

Splitmesh

Another way to split mesh triangles is to use `splitmesh`, for example:

```
// Mesh
border a(t=0, 2*pi){x=cos(t); y=sin(t); label=1;}
mesh Th = buildmesh(a(20));
plot(Th, wait=true, ps="NotSplittedMesh.eps");

// Splitmesh
Th = splitmesh(Th, 1 + 5*(square(x-0.5) + y*y));
plot(Th, wait=true, ps="SplittedMesh.eps");
```

Meshing Examples

Tip: Two rectangles touching by a side

(a) Initial mesh

(b) All left mesh triangle is split conformaly in `int(1+5*(square(x-0.5)+y*y)^2` triangles

Fig. 3.13: Split mesh

```
border a(t=0, 1){x=t; y=0;};
border b(t=0, 1){x=1; y=t;};
border c(t=1, 0){x=t; y=1;};
border d(t=1, 0){x=0; y=t;};
border c1(t=0, 1){x=t; y=1;};
border e(t=0, 0.2){x=1; y=1+t;};
border f(t=1, 0){x=t; y=1.2;};
border g(t=0.2, 0){x=0; y=1+t;};
int n=1;
mesh th = buildmesh(a(10*n) + b(10*n) + c(10*n) + d(10*n));
mesh TH = buildmesh(c1(10*n) + e(5*n) + f(10*n) + g(5*n));
plot(th, TH, ps="TouchSide.esp");
```

Fig. 3.14: Two rectangles touching by a side

Tip: NACA0012 Airfoil

```
border upper(t=0, 1){x=t; y=0.17735*sqrt(t) - 0.075597*t - 0.212836*(t^2) + 0.
↪17363*(t^3) - 0.06254*(t^4);}
border lower(t=1, 0){x = t; y=-(0.17735*sqrt(t) -0.075597*t - 0.212836*(t^2) + 0.
↪17363*(t^3) - 0.06254*(t^4));}
border c(t=0, 2*pi){x=0.8*cos(t) + 0.5; y=0.8*sin(t);}
mesh Th = buildmesh(c(30) + upper(35) + lower(35));
plot(Th, ps="NACA0012.eps", bw=true);
```

Fig. 3.15: NACA0012 Airfoil

Tip: Cardioid

```
real b = 1, a = b;
border C(t=0, 2*pi){x=(a+b)*cos(t)-b*cos((a+b)*t/b); y=(a+b)*sin(t)-b*sin((a+b)*t/b);}
mesh Th = buildmesh(C(50));
plot(Th, ps="Cardioid.eps", bw=true);
```

Tip: Cassini Egg

```
border C(t=0, 2*pi) {x=(2*cos(2*t)+3)*cos(t); y=(2*cos(2*t)+3)*sin(t);}
mesh Th = buildmesh(C(50));
plot(Th, ps="Cassini.eps", bw=true);
```

Tip: By cubic Bezier curve

```
// A cubic Bezier curve connecting two points with two control points
func real bzi(real p0, real p1, real q1, real q2, real t){
    return p0*(1-t)^3 + q1*3*(1-t)^2*t + q2*3*(1-t)*t^2 + p1*t^3;
}
```

(continues on next page)

Fig. 3.16: Domain with Cardioid curve boundary

Fig. 3.17: Domain with Cassini egg curve boundary

(continued from previous page)

```

real[int] p00 = [0, 1], p01 = [0, -1], q00 = [-2, 0.1], q01 = [-2, -0.5];
real[int] p11 = [1,-0.9], q10 = [0.1, -0.95], q11=[0.5, -1];
real[int] p21 = [2, 0.7], q20 = [3, -0.4], q21 = [4, 0.5];
real[int] q30 = [0.5, 1.1], q31 = [1.5, 1.2];
border G1(t=0, 1){
    x=bzi(p00[0], p01[0], q00[0], q01[0], t);
    y=bzi(p00[1], p01[1], q00[1], q01[1], t);
}
border G2(t=0, 1){
    x=bzi(p01[0], p11[0], q10[0], q11[0], t);
    y=bzi(p01[1], p11[1], q10[1], q11[1], t);
}
border G3(t=0, 1){
    x=bzi(p11[0], p21[0], q20[0], q21[0], t);
    y=bzi(p11[1], p21[1], q20[1], q21[1], t);
}
border G4(t=0, 1){
    x=bzi(p21[0], p00[0], q30[0], q31[0], t);
    y=bzi(p21[1], p00[1], q30[1], q31[1], t);
}
int m = 5;
mesh Th = buildmesh(G1(2*m) + G2(m) + G3(3*m) + G4(m));
plot(Th, ps="Bezier.eps", bw=true);
```

Fig. 3.18: Boundary drawn by Bezier curves

Tip: Section of Engine

```
real a = 6., b = 1., c = 0.5;

border L1(t=0, 1){x=-a; y=1+b-2*(1+b)*t;}
border L2(t=0, 1){x=-a+2*a*t; y=-1-b*(x/a)*(x/a)*(3-2*abs(x)/a );}
border L3(t=0, 1){x=a; y=-1-b+(1+b)*t; }
border L4(t=0, 1){x=a-a*t; y=0;}
border L5(t=0, pi){x=-c*sin(t)/2; y=c/2-c*cos(t)/2;}
border L6(t=0, 1){x=a*t; y=c;}
border L7(t=0, 1){x=a; y=c+(1+b-c)*t;}
border L8(t=0, 1){x=a-2*a*t; y=1+b*(x/a)*(x/a)*(3-2*abs(x)/a);}
mesh Th = buildmesh(L1(8) + L2(26) + L3(8) + L4(20) + L5(8) + L6(30) + L7(8) +
↪L8(30));
```

(continues on next page)

(continued from previous page)

```
plot(Th, ps="Engine.eps", bw=true);
```

Fig. 3.19: Section of Engine

Tip: Domain with U-shape channel

```
real d = 0.1; //width of U-shape
border L1(t=0, 1-d){x=-1; y=-d-t;}
border L2(t=0, 1-d){x=-1; y=1-t;}
border B(t=0, 2){x=-1+t; y=-1;}
border C1(t=0, 1){x=t-1; y=d;}
border C2(t=0, 2*d){x=0; y=d-t;}
border C3(t=0, 1){x=-t; y=-d;}
border R(t=0, 2){x=1; y=-1+t;}
border T(t=0, 2){x=1-t; y=1;}
int n = 5;
mesh Th = buildmesh(L1(n/2) + L2(n/2) + B(n) + C1(n) + C2(3) + C3(n) + R(n) + T(n));
plot(Th, ps="U-shape.eps", bw=true);
```

Fig. 3.20: Domain with U-shape channel changed by :freefem‘d‘

Tip: Domain with V-shape cut

```
real dAg = 0.02; //angle of V-shape
border C(t=dAg, 2*pi-dAg){x=cos(t); y=sin(t);};
real[int] pa(2), pb(2), pc(2);
```

(continues on next page)

(continued from previous page)

```
pa[0] = cos(dAg);
pa[1] = sin(dAg);
pb[0] = cos(2*pi-dAg);
pb[1] = sin(2*pi-dAg);
pc[0] = 0;
pc[1] = 0;
border seg1(t=0, 1){x=(1-t)*pb[0]+t*pc[0]; y=(1-t)*pb[1]+t*pc[1];};
border seg2(t=0, 1){x=(1-t)*pc[0]+t*pa[0]; y=(1-t)*pc[1]+t*pa[1];};
mesh Th = buildmesh(seg1(20) + C(40) + seg2(20));
plot(Th, ps="V-shape.eps", bw=true);
```

Fig. 3.21: Domain with V-shape cut changed by `dAg`

Tip: Smiling face

```
real d=0.1; int m = 5; real a = 1.5, b = 2, c = 0.7, e = 0.01;

border F(t=0, 2*pi){x=a*cos(t); y=b*sin(t);}
border E1(t=0, 2*pi){x=0.2*cos(t)-0.5; y=0.2*sin(t)+0.5;}
border E2(t=0, 2*pi){x=0.2*cos(t)+0.5; y=0.2*sin(t)+0.5;}
func real st(real t){
    return sin(pi*t) - pi/2;
}
border C1(t=-0.5, 0.5){x=(1-d)*c*cos(st(t)); y=(1-d)*c*sin(st(t));}
border C2(t=0, 1){x=((1-d)+d*t)*c*cos(st(0.5)); y=((1-d)+d*t)*c*sin(st(0.5));}
border C3(t=0.5, -0.5){x=c*cos(st(t)); y=c*sin(st(t));}
border C4(t=0, 1){x=(1-d*t)*c*cos(st(-0.5)); y=(1-d*t)*c*sin(st(-0.5));}
border C0(t=0, 2*pi){x=0.1*cos(t); y=0.1*sin(t);}

mesh Th=buildmesh(F(10*m) + C1(2*m) + C2(3) + C3(2*m) + C4(3)
    + C0(m) + E1(-2*m) + E2(-2*m));
plot(Th, ps="SmileFace.eps", bw=true);
```

Fig. 3.22: Smiling face (Mouth is changeable)

. tip:: 3 points bending

```
// Square for Three-Point Bend Specimens fixed on Fix1, Fix2
// It will be loaded on Load.
real a = 1, b = 5, c = 0.1;
int n = 5, m = b*n;
border Left(t=0, 2*a){x=-b; y=a-t;}
border Bot1(t=0, b/2-c){x=-b+t; y=-a;}
border Fix1(t=0, 2*c){x=-b/2-c+t; y=-a;}
border Bot2(t=0, b-2*c){x=-b/2+c+t; y=-a;}
border Fix2(t=0, 2*c){x=b/2-c+t; y=-a;}
border Bot3(t=0, b/2-c){x=b/2+c+t; y=-a;}
border Right(t=0, 2*a){x=b; y=-a+t;}
border Top1(t=0, b-c){x=b-t; y=a;}
border Load(t=0, 2*c){x=c-t; y=a;}
border Top2(t=0, b-c){x=-c-t; y=a;}
mesh Th = buildmesh(Left(n) + Bot1(m/4) + Fix1(5) + Bot2(m/2)
    + Fix2(5) + Bot3(m/4) + Right(n) + Top1(m/2) + Load(10) + Top2(m/2));
plot(Th, ps="ThreePoint.eps", bw=true);
```

How to change the label of elements and border elements of a mesh

Changing the label of elements and border elements will be done using the keyword `change`. The parameters for this command line are for two dimensional and three dimensional cases:

- `label` = is a vector of integer that contains successive pairs of the old label number to the new label number.
- `region` = is a vector of integer that contains successive pairs of the old region number to new region number.

Fig. 3.23: Domain for three-point bending test

- `flabel` = is an integer function given the new value of the label.
- `fregion`= is an integer function given the new value of the region.

These vectors are composed of n_l successive pairs of numbers O, N where n_l is the number (label or region) that we want to change. For example, we have :

$$\begin{aligned} \texttt{label} &= [O_1, N_1, ..., O_{n_l}, N_{n_l}] \\ \texttt{region} &= [O_1, N_1, ..., O_{n_l}, N_{n_l}] \end{aligned} \tag{3.1}$$

An example of using this function is given here:

```
verbosity=3;

// Mesh
mesh Th1 = square(10, 10);
mesh Th2 = square(20, 10, [x+1, y]);

int[int] r1=[2,0];
plot(Th1, wait=true);

Th1 = change(Th1, label=r1); //change the label of Edges 2 in 0.
plot(Th1, wait=true);

int[int] r2=[4,0];
Th2 = change(Th2, label=r2); //change the label of Edges 4 in 0.
plot(Th2, wait=true);

mesh Th = Th1 + Th2; //'gluing together' of meshes Th1 and Th2
cout << "nb lab = " << int1d(Th1,1,3,4)(1./lenEdge)+int1d(Th2,1,2,3)(1./lenEdge)
     << " == " << int1d(Th,1,2,3,4)(1./lenEdge) << " == " << ((10+20)+10)*2 << endl;
plot(Th, wait=true);

fespace Vh(Th, P1);
Vh u, v;

macro Grad(u) [dx(u),dy(u)] // Definition of a macro

solve P(u, v)
    = int2d(Th)(
          Grad(u)'*Grad(v)
    )
    -int2d(Th)(
```

(continues on next page)

(continued from previous page)

```
32            v
33      )
34      + on(1, 3, u=0)
35      ;
36
37  plot(u, wait=1);
```

"gluing" different mesh In line 17 of the previous file, the method to "gluing" different meshes of the same dimension in **FreeFEM** is using. This function is the operator "+" between meshes. The method implemented needs the point in adjacent meshes to be the same.

Mesh in three dimensions

3.2.8 Cube

A new function `cube` like the function `square` in 2d is the simple way to a build cubic object, in plugin `msh3` (need `load "msh3"`).

The following code generates a $3 \times 4 \times 5$ grid in the unit cube $[0,1]^3$.

```
1  mesh3 Th = cube(3, 4, 5);
```

By default the labels are :

1. face $y = 0$,
2. face $x = 1$,
3. face $y = 1$,
4. face $x = 0$,
5. face $z = 0$,
6. face $z = 1$

and the region number is 0.

A full example of this function to build a mesh of cube $]-1,1[^3$ with face label given by $(ix+4*(iy+1)+16*(iz+1))$ where (ix, iy, iz) are the coordinates of the barycenter of the current face, is given below.

```
1  load "msh3"
2
3  int[int] l6 = [37, 42, 45, 40, 25, 57];
4  int r11 = 11;
5  mesh3 Th = cube(4, 5, 6, [x*2-1, y*2-1, z*2-1], label=l6, flags =3, region=r11);
6
7  cout << "Volume = " << Th.measure << ", border area = " << Th.bordermeasure << endl;
8
9  int err = 0;
10 for(int i = 0; i < 100; ++i){
11     real s = int2d(Th,i)(1.);
12     real sx = int2d(Th,i)(x);
13     real sy = int2d(Th,i)(y);
14     real sz = int2d(Th,i)(z);
15
16     if(s){
17         int ix = (sx/s+1.5);
```

(continues on next page)

(continued from previous page)

```
        int iy = (sy/s+1.5);
        int iz = (sz/s+1.5);
        int ii = (ix + 4*(iy+1) + 16*(iz+1) );
        //value of ix,iy,iz => face min 0, face max 2, no face 1
        cout << "Label = " << i << ", s = " << s << " " << ix << iy << iz << " : " << ii << endl;
        if( i != ii ) err++;
    }
}
real volr11 = int3d(Th,r11)(1.);
cout << "Volume region = " << 11 << ": " << volr11 << endl;
if((volr11 - Th.measure )>1e-8) err++;
plot(Th, fill=false);
cout << "Nb err = " << err << endl;
assert(err==0);
```

The output of this script is:

```
Enter: BuildCube: 3
  kind = 3 n tet Cube = 6 / n slip 6 19
Cube  nv=210 nt=720 nbe=296
Out:  BuildCube
Volume = 8, border area = 24
Label = 25, s = 4 110 : 25
Label = 37, s = 4 101 : 37
Label = 40, s = 4 011 : 40
Label = 42, s = 4 211 : 42
Label = 45, s = 4 121 : 45
Label = 57, s = 4 112 : 57
Volume region = 11: 8
Nb err = 0
```

Fig. 3.24: The mesh 3d of function `cube(4, 5, 6, flags=3)`

3.2.9 Read/Write Statements for a Mesh in 3D

In three dimensions, the file mesh format supported for input and output files by **FreeFEM** are the extension .msh and .mesh. These formats are described in the *Mesh Format section*.

Extension file .msh The structure of the files with extension .msh in 3D is given by:

$$
\begin{array}{ccccc}
n_v & n_{tet} & n_{tri} & & \\
q_x^1 & q_y^1 & q_z^1 & Vertexlabel & \\
q_x^2 & q_y^2 & q_z^2 & Vertexlabel & \\
\vdots & \vdots & \vdots & \vdots & \\
q_x^{n_v} & q_y^{n_v} & q_z^{n_v} & Vertexlabel & \\
1_1 & 1_2 & 1_3 & 1_4 & regionlabel \\
2_1 & 2_2 & 2_3 & 2_4 & regionlabel \\
\vdots & \vdots & \vdots & \vdots & \vdots \\
(n_{tet})_1 & (n_{tet})_2 & (n_{tet})_3 & (n_{tet})_4 & regionlabel \\
1_1 & 1_2 & 1_3 & boundarylabel & \\
2_1 & 2_2 & 2_3 & boundarylabel & \\
\vdots & \vdots & \vdots & \vdots & \\
(n_tri)_1 & (n_{tri})_2 & (n_{tri})_3 & boundarylabel &
\end{array}
$$

In this structure, n_v denotes the number of vertices, n_{tet} the number of tetrahedra and n_{tri} the number of triangles.

For each vertex q^i, $i = 1, \cdots, n_v$, we denote by (q_x^i, q_y^i, q_z^i) the x-coordinate, the y-coordinate and the z-coordinate.

Each tetrahedra $T_k, k = 1, \cdots, n_{tet}$ has four vertices q^{k_1}, q^{k_2}, q^{k_3}, q^{k_4}.

The boundary consists of a union of triangles. Each triangle $be_j, j = 1, \cdots, n_{tri}$ has three vertices q^{j_1}, q^{j_2}, q^{j_3}.

extension file .mesh The data structure for a three dimensional mesh is composed of the data structure presented in *Mesh Format section* and a data structure for the tetrahedra. The tetrahedra of a three dimensional mesh are referred using the following field:

```
Tetrahedra
NbTetrahedra
Vertex1 Vertex2 Vertex3 Vertex4 Label
...
Vertex1 Vertex2 Vertex3 Vertex4 Label
```

This field is express with the notation of *Mesh Format section*.

3.2.10 TetGen: A tetrahedral mesh generator

TetGen

TetGen is a software developed by Dr. Hang Si of Weierstrass Institute for Applied Analysis and Stochastics in Berlin, Germany [HANG2006]. TetGen is free for research and non-commercial use. For any commercial license utilization, a commercial license is available upon request to Hang Si.

This software is a tetrahedral mesh generator of a three dimensional domain defined by its boundary. The input domain takes into account a polyhedral or a piecewise linear complex. This tetrahedralization is a constrained Delaunay tetrahedralization.

The method used in TetGen to control the quality of the mesh is a Delaunay refinement due to Shewchuk [SHEWCHUK1998]. The quality measure of this algorithm is the Radius-Edge Ratio (see Section 1.3.1 [HANG2006] for more details). A theoretical bound of this ratio of the Shewchuk algorithm is obtained for a given complex of

vertices, constrained segments and facets of surface mesh, with no input angle less than 90 degrees. This theoretical bound is 2.0.

The launch of TetGen is done with the keyword `tetg`. The parameters of this command line is:

- `reftet=` sets the label of tetrahedra.
- **`label=` is a vector of integers that contains the old labels number at index $2i$ and the new labels number at index $2i+1$ c**
  This parameter is initialized as a label for the keyword *change*.
- **`switch=` A string expression.** This string corresponds to the command line switch of TetGen see Section 3.2 of [HANG2006].
- `nbofholes=` Number of holes (default value: "size of `holelist` / 3").
- **`holelist=` This array corresponds to `holelist` of TetGenio data structure [HANG2006].** A real vector of size `3 * nbofholes`. In TetGen, each hole is associated with a point inside this domain. This vector is $x_1^h, y_1^h, z_1^h, x_2^h, y_2^h, z_2^h, \cdots$, where x_i^h, y_i^h, z_i^h is the associated point with the ith hole.
- `nbofregions=` Number of regions (default value: "size of `regionlist` / 5").
- `regionlist=` This array corresponds to `regionlist` of TetGenio data structure [HANG2006].

  The attribute and the volume constraint of region are given in this real vector of size `5 * nbofregions`. The ith region is described by five elements: $x-$coordinate, $y-$coordinate and $z-$coordinate of a point inside this domain (x_i, y_i, z_i); the attribute (at_i) and the maximum volume for tetrahedra $(mvol_i)$ for this region.

  The `regionlist` vector is: $x_1, y_1, z_1, at_1, mvol_1, x_2, y_2, z_2, at_2, mvol_2, \cdots$.
- `nboffacetcl=` Number of facets constraints "size of `facetcl` / 2").
- `facetcl=` This array corresponds to `facetconstraintlist` of TetGenio data structure [HANG2006].

  The ith facet constraint is defined by the facet marker Ref_i^{fc} and the maximum area for faces $marea_i^{fc}$. The `facetcl` array is: $Ref_1^{fc}, marea_1^{fc}, Ref_2^{fc}, marea_2^{fc}, \cdots$.

  This parameters has no effect if switch `q` is not selected.

Principal switch parameters in TetGen:

- `p` Tetrahedralization of boundary.
- **`q` Quality mesh generation.** The bound of Radius-Edge Ratio will be given after the option `q`. By default, this value is 2.0.
- **`a` Constructs with the volume constraints on tetrahedra.** These volumes constraints are defined with the bound of the previous switch `q` or in the parameter `regionlist`.
- **`A` Attributes reference to region given in the `regionlist`.** The other regions have label 0.

  The option `AA` gives a different label at each region. This switch works with the option `p`. If option :freefem'r' is used, this switch has no effect.
- **`r` Reconstructs and Refines a previously generated mesh.** This character is only used with the command line `tetgreconstruction`.
- `Y` This switch preserves the mesh on the exterior boundary.

  This switch must be used to ensure a conformal mesh between two adjacent meshes.
- `YY` This switch preserves the mesh on the exterior and interior boundary.
- `C` The consistency of the result's mesh is testing by TetGen.
- `CC` The consistency of the result's mesh is testing by TetGen and also constrained checks of Delaunay mesh (if `p` switch is selected) or the consistency of Conformal Delaunay (if `q` switch is selected).

- **`V` Give information of the work of TetGen.** More information can be obtained in specified `VV` or `VVV`.
- `Q` Quiet: No terminal output except errors
- `M` The coplanar facets are not merging.
- **`T` Sets a tolerance for coplanar test.** The default value is $1e-8$.
- `d` Intersections of facets are detected.

To obtain a tetrahedral mesh with TetGen, we need the surface mesh of a three dimensional domain. We now give the command line in **FreeFEM** to construct these meshes.

keyword: movemesh23

A simple method to construct a surface is to place a two dimensional domain in a three dimensional space. This corresponds to moving the domain by a displacement vector of this form $\mathbf{\Phi}(\mathbf{x},\mathbf{y}) = (\Phi1(x,y), \Phi2(x,y), \Phi3(x,y))$.

The result of moving a two dimensional mesh Th2 by this three dimensional displacement is obtained using:

```
mesh3 Th3 = movemesh23(Th2, transfo=[Phi(1), Phi(2), Phi(3)]);
```

The parameters of this command line are:

- `transfo=` [$\Phi1$, $\Phi2$, $\Phi3$] sets the displacement vector of transformation $\mathbf{\Phi}(\mathbf{x},\mathbf{y}) = [\Phi1(x,y), \Phi2(x,y), \Phi3(x,y)]$.
- `label=` sets an integer label of triangles.
- `orientation=` sets an integer orientation of mesh.
- **`ptmerge=` A real expression.** When you transform a mesh, some points can be merged. This parameter is the criteria to define two merging points. By default, we use

$$ptmerge \; = \; 1e-7 \; Vol(B),$$

  where B is the smallest axis, parallel boxes containing the discretized domain of Ω and $Vol(B)$ is the volume of this box.

We can do a "gluing" of surface meshes using the process given in *Change section*. An example to obtain a three dimensional mesh using the command line `tetg` and `movemesh23` is given below.

```
load "msh3"
load "tetgen"

// Parameters
real x10 = 1.;
real x11 = 2.;
real y10 = 0.;
real y11 = 2.*pi;

func ZZ1min = 0;
func ZZ1max = 1.5;
func XX1 = x;
func YY1 = y;

real x20 = 1.;
real x21 = 2.;
real y20=0.;
real y21=1.5;

```

(continues on next page)

(continued from previous page)

```
func ZZ2 = y;
func XX2 = x;
func YY2min = 0.;
func YY2max = 2*pi;

real x30=0.;
real x31=2*pi;
real y30=0.;
real y31=1.5;

func XX3min = 1.;
func XX3max = 2.;
func YY3 = x;
func ZZ3 = y;

// Mesh
mesh Thsq1 = square(5, 35, [x10+(x11-x10)*x, y10+(y11-y10)*y]);
mesh Thsq2 = square(5, 8, [x20+(x21-x20)*x, y20+(y21-y20)*y]);
mesh Thsq3 = square(35, 8, [x30+(x31-x30)*x, y30+(y31-y30)*y]);

// Mesh 2D to 3D surface
mesh3 Th31h = movemesh23(Thsq1, transfo=[XX1, YY1, ZZ1max]);
mesh3 Th31b = movemesh23(Thsq1, transfo=[XX1, YY1, ZZ1min]);

mesh3 Th32h = movemesh23(Thsq2, transfo=[XX2, YY2max, ZZ2]);
mesh3 Th32b = movemesh23(Thsq2, transfo=[XX2, YY2min, ZZ2]);

mesh3 Th33h = movemesh23(Thsq3, transfo=[XX3max, YY3, ZZ3]);
mesh3 Th33b = movemesh23(Thsq3, transfo=[XX3min, YY3, ZZ3]);

// Gluing surfaces
mesh3 Th33 = Th31h + Th31b + Th32h + Th32b + Th33h + Th33b;
plot(Th33, cmm="Th33");

// Tetrahelize the interior of the cube with TetGen
real[int] domain =[1.5, pi, 0.75, 145, 0.0025];
mesh3 Thfinal = tetg(Th33, switch="paAAQY", regionlist=domain);
plot(Thfinal, cmm="Thfinal");

// Build a mesh of a half cylindrical shell of interior radius 1, and exterior radius␣
↪2 and a height of 1.5
func mv2x = x*cos(y);
func mv2y = x*sin(y);
func mv2z = z;
mesh3 Thmv2 = movemesh3(Thfinal, transfo=[mv2x, mv2y, mv2z]);
plot(Thmv2, cmm="Thmv2");
```

The command `movemesh3` is described in the following section.

The keyword tetgtransfo

This keyword corresponds to a composition of command line `tetg` and `movemesh23`.

```
tetgtransfo(Th2, transfo=[Phi(1), Phi(2), Phi(3)]), ...) = tetg(Th3surf, ...),
```

where `Th3surf = movemesh23(Th2, transfo=[Phi(1), Phi(2), Phi(3)])` and `Th2` is the input two dimensional mesh of `tetgtransfo`.

The parameters of this command line are, on one hand, the parameters `label`, `switch`, `regionlist`, `nboffacetcl`, `facetcl` of keyword `tetg` and on the other hand, the parameter `ptmerge` of keyword `movemesh23`.

Note: To use `tetgtransfo`, the result's mesh of `movemesh23` must be a closed surface and define one region only. Therefore, the parameter `regionlist` is defined for one region.

An example of this keyword can be found in line 61 of the *Build layer mesh example*.

The keyword tetgconvexhull

FreeFEM, using TetGen, is able to build a tetrahedralization from a set of points. This tetrahedralization is a Delaunay mesh of the convex hull of the set of points.

The coordinates of the points can be initialized in two ways. The first is a file that contains the coordinate of points $X_i = (x_i, y_i, z_i)$. This file is organized as follows:

$$\begin{array}{ccc} n_v & & \\ x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ \vdots & \vdots & \vdots \\ x_{n_v} & y_{n_v} & z_{n_v} \end{array}$$

The second way is to give three arrays that correspond respectively to the $x-$coordinates, $y-$coordinates and $z-$coordinates.

The parameters of this command line are :

- **`switch=` A string expression.** This string corresponds to the command line `switch` of TetGen see Section 3.2 of [HANG2006].
- **`reftet=` An integer expression.** Set the label of tetrahedra.
- **`label=` An integer expression.** Set the label of triangles.

In the string `switch`, we can't used the option `p` and `q` of TetGen.

3.2.11 Reconstruct/Refine a three dimensional mesh with TetGen

Meshes in three dimension can be refined using TetGen with the command line `tetgreconstruction`.

The parameter of this keyword are

- **`region=` an integer array that changes the region number of tetrahedra.** This array is defined as the parameter `reftet` in the keyword `change`.
- **`label=` an integer array that changes the label of boundary triangles.** This array is defined as the parameter `label` in the keyword `change`.
- **`sizeofvolume=` a reel function.** This function constraints the volume size of the tetrahedra in the domain (see *Isotrope mesh adaption section* to build a 3d adapted mesh).

The parameters `switch`, `nbofregions`, `regionlist`, `nboffacetcl` and `facetcl` of the command line which call TetGen (`tetg`) is used for `tetgrefine`.

In the parameter `switch=`, the character `r` should be used without the character `p`.

For instance, see the manual of TetGen [HANG2006] for effect of `r` to other character.

The parameter `regionlist` defines a new volume constraint in the region. The label in the `regionlist` will be the previous label of region.

This parameter and `nbofregions` can't be used with the parameter `sizeofvolume`.

Example refinesphere.edp

```
load "msh3"
load "TetGen"
load "medit"

mesh Th = square(10, 20, [x*pi-pi/2, 2*y*pi]); // $]-pi/2, pi/2[X]0, 2pi[ $

// A parametrization of a sphere
func f1 = cos(x)*cos(y);
func f2 = cos(x)*sin(y);
func f3 = sin(x);
// Partial derivative of the parametrization DF
func f1x = sin(x)*cos(y);
func f1y = -cos(x)*sin(y);
func f2x = -sin(x)*sin(y);
func f2y = cos(x)*cos(y);
func f3x = cos(x);
func f3y = 0;
// M = DF^t DF
func m11 = f1x^2 + f2x^2 + f3x^2;
func m21 = f1x*f1y + f2x*f2y + f3x*f3y;
func m22 = f1y^2 + f2y^2 + f3y^2;

// Mesh adaptation
func perio = [[4, y], [2, y], [1, x], [3, x]];
real hh = 0.1;
real vv = 1/square(hh);
verbosity = 2;
Th = adaptmesh(Th, m11*vv, m21*vv, m22*vv, IsMetric=1, periodic=perio);
Th = adaptmesh(Th, m11*vv, m21*vv, m22*vv, IsMetric=1, periodic=perio);
plot(Th, wait=true);

// Construction of the surface of spheres
real Rmin = 1.;
func f1min = Rmin*f1;
func f2min = Rmin*f2;
func f3min = Rmin*f3;

mesh3 Th3 = movemesh23(Th, transfo=[f1min, f2min, f3min]);

// Contruct the volume
real[int] domain = [0., 0., 0., 145, 0.01];
mesh3 Th3sph = tetg(Th3, switch="paAAQYY", nbofregions=1, regionlist=domain);

// Refine
int[int] newlabel = [145, 18];
real[int] domainrefine = [0., 0., 0., 145, 0.0001];
mesh3 Th3sphrefine = tetgreconstruction(Th3sph, switch="raAQ", reftet=newlabel,
    nbofregions=1, regionlist=domain, sizeofvolume=0.0001);

// Re-Refine
int[int] newlabel2 = [145, 53];
func fsize = 0.01/((1 + 5*sqrt((x-0.5)^2+(y-0.5)^2+(z-0.5)^2))^3);
```

(continues on next page)

(continued from previous page)

```
mesh3 Th3sphrefine2 = tetgreconstruction(Th3sph, switch="raAQ", reftet=newlabel2,
    sizeofvolume=fsize);

// Medit
medit("sphere", Th3sph);
medit("isotroperefine", Th3sphrefine);
medit("anisotroperefine", Th3sphrefine2);
```

3.2.12 Moving mesh in three dimensions

Meshes in three dimensions can be translated, rotated, and deformed using the command line `movemesh` as in the 2D case (see *section movemesh*). If Ω is tetrahedrized as $T_h(\Omega)$, and $\Phi(x,y) = (\Phi1(x,y,z), \Phi2(x,y,z), \Phi3(x,y,z))$ is a displacement vector then $\Phi(T_h)$ is obtained by:

```
mesh3 Th = movemesh(Th, [Phi1, Phi2, Phi3], ...);
```

The parameters of movemesh in three dimensions are:

- **`region=` sets the integer labels of the tetrahedra.** 0 by default.
- **`label=` sets the labels of the border faces.** This parameter is initialized as the label for the keyword *change*.
- **`facemerge=` An integer expression.** When you transform a mesh, some faces can be merged. This parameter equals to one if the merges' faces is considered. Otherwise it equals to zero. By default, this parameter is equal to 1.
- **`ptmerge =` A real expression.** When you transform a mesh, some points can be merged. This parameter is the criteria to define two merging points. By default, we use

  $$ptmerge \; = \; 1e-7 \; Vol(B),$$

  where B is the smallest axis parallel boxes containing the discretion domain of Ω and $Vol(B)$ is the volume of this box.
- `orientation =` An integer expression (1 by default), to reverse or not to reverse the orientation of the tetrahedra if it is not positive.

An example of this command can be found in the *Poisson's equation 3D example*.

3.2.13 Layer mesh

In this section, we present the command line to obtain a Layer mesh: `buildlayers`. This mesh is obtained by extending a two dimensional mesh in the z-axis.

The domain Ω_{3d} defined by the layer mesh is equal to $\Omega_{3d} = \Omega_{2d} \times [zmin, zmax]$ where Ω_{2d} is the domain defined by the two dimensional meshes. $zmin$ and $zmax$ are functions of Ω_{2d} in $\mathbb{R}$ that defines respectively the lower surface and upper surface of Ω_{3d}.

For a vertex of a two dimensional mesh $V_i^{2d} = (x_i, y_i)$, we introduce the number of associated vertices in the $z-$axis $M_i + 1$.

We denote by M the maximum of M_i over the vertices of the two dimensional mesh. This value is called the number of layers (if $\forall i, \; M_i = M$ then there are M layers in the mesh of Ω_{3d}). V_i^{2d} generated $M+1$ vertices which are defined by:

$$\forall j = 0, \ldots, M, \quad V_{i,j}^{3d} = (x_i, y_i, \theta_i(z_{i,j})),$$

Fig. 3.25: Example of Layer mesh in three dimensions.

where $(z_{i,j})_{j=0,\dots,M}$ are the $M+1$ equidistant points on the interval $[zmin(V_i^{2d}), zmax(V_i^{2d})]$:

$$z_{i,j} = j\ \delta\alpha + zmin(V_i^{2d}), \quad \delta\alpha = \frac{zmax(V_i^{2d}) - zmin(V_i^{2d})}{M}.$$

The function θ_i, defined on $[zmin(V_i^{2d}), zmax(V_i^{2d})]$, is given by:

$$\theta_i(z) = \begin{cases} \theta_{i,0} & \text{if } z = zmin(V_i^{2d}), \\ \theta_{i,j} & \text{if } z \in]\theta_{i,j-1}, \theta_{i,j}], \end{cases}$$

with $(\theta_{i,j})_{j=0,\dots,M_i}$ are the M_i+1 equidistant points on the interval $[zmin(V_i^{2d}), zmax(V_i^{2d})]$.

Set a triangle $K = (V_{i1}^{2d}, V_{i2}^{2d}, V_{i3}^{2d})$ of the two dimensional mesh. K is associated with a triangle on the upper surface (resp. on the lower surface) of layer mesh:

$(V_{i1,M}^{3d}, V_{i2,M}^{3d}, V_{i3,M}^{3d})$ (resp. $(V_{i1,0}^{3d}, V_{i2,0}^{3d}, V_{i3,0}^{3d})$).

Also K is associated with M volume prismatic elements which are defined by:

$$\forall j = 0, \dots, M, \quad H_j = (V_{i1,j}^{3d}, V_{i2,j}^{3d}, V_{i3,j}^{3d}, V_{i1,j+1}^{3d}, V_{i2,j+1}^{3d}, V_{i3,j+1}^{3d}).$$

Theses volume elements can have some merged point:

- 0 merged point : prism
- 1 merged points : pyramid
- 2 merged points : tetrahedra
- 3 merged points : no elements

The elements with merged points are called degenerate elements. To obtain a mesh with tetrahedra, we decompose the pyramid into two tetrahedra and the prism into three tetrahedra. These tetrahedra are obtained by cutting the quadrilateral face of pyramid and prism with the diagonal which have the vertex with the maximum index (see [HECHT1992] for the reason of this choice).

The triangles on the middle surface obtained with the decomposition of the volume prismatic elements are the triangles generated by the edges on the border of the two dimensional mesh. The label of triangles on the border elements and tetrahedra are defined with the label of these associated elements.

The arguments of `buildlayers` is a two dimensional mesh and the number of layers M.

The parameters of this command are:

- **`zbound=`** $[zmin, zmax]$ **where** $zmin$ **and** $zmax$ **are functions expression.** Theses functions define the lower surface mesh and upper mesh of surface mesh.
- **`coef=` A function expression between [0,1].** This parameter is used to introduce degenerate element in mesh.

  The number of associated points or vertex V_i^{2d} is the integer part of $coef(V_i^{2d})M$.
- `region=` This vector is used to initialize the region of tetrahedra.

  This vector contains successive pairs of the 2d region number at index $2i$ and the corresponding 3d region number at index $2i+1$, like *change*.
- `labelmid=` This vector is used to initialize the 3d labels number of the vertical face or mid face from the 2d label number.

  This vector contains successive pairs of the 2d label number at index $2i$ and the corresponding 3d label number at index $2i+1$, like *change*.
- `labelup=` This vector is used to initialize the 3d label numbers of the upper/top face from the 2d region number.

  This vector contains successive pairs of the 2d region number at index $2i$ and the corresponding 3d label number at index $2i+1$, like *change*.
- `labeldown=` Same as the previous case but for the lower/down face label.

Moreover, we also add post processing parameters that allow to moving the mesh. These parameters correspond to parameters `transfo`, `facemerge` and `ptmerge` of the command line `movemesh`.

The vector `region`, `labelmid`, `labelup` and `labeldown` These vectors are composed of n_l successive pairs of number O_i, N_l where n_l is the number (label or region) that we want to get.

An example of this command is given in the *Build layer mesh example*.

Tip: Cube

```
//Cube.idp
load "medit"
load "msh3"

func mesh3 Cube (int[int] &NN, real[int, int] &BB, int[int, int] &L){
    real x0 = BB(0,0), x1 = BB(0,1);
    real y0 = BB(1,0), y1 = BB(1,1);
    real z0 = BB(2,0), z1 = BB(2,1);

    int nx = NN[0], ny = NN[1], nz = NN[2];

    // 2D mesh
    mesh Thx = square(nx, ny, [x0+(x1-x0)*x, y0+(y1-y0)*y]);

    // 3D mesh
    int[int] rup = [0, L(2,1)], rdown=[0, L(2,0)];
    int[int] rmid=[1, L(1,0), 2, L(0,1), 3, L(1,1), 4, L(0,0)];
    mesh3 Th = buildlayers(Thx, nz, zbound=[z0,z1],
```

(continues on next page)

(continued from previous page)

```
    labelmid=rmid, labelup = rup, labeldown = rdown);

    return Th;
}
```

Tip: Unit cube

```
include "Cube.idp"

int[int] NN = [10,10,10]; //the number of step in each direction
real [int, int] BB = [[0,1],[0,1],[0,1]]; //the bounding box
int [int, int] L = [[1,2],[3,4],[5,6]]; //the label of the 6 face left,right, front,
↪back, down, right
mesh3 Th = Cube(NN, BB, L);
medit("Th", Th);
```

Fig. 3.26: The mesh of a cube made with `cube.edp`

Tip: Cone

An axisymtric mesh on a triangle with degenerateness

```
load "msh3"
load "medit"

// Parameters
real RR = 1;
real HH = 1;

int nn=10;
```

(continues on next page)

(continued from previous page)

```

// 2D mesh
border Taxe(t=0, HH){x=t; y=0; label=0;}
border Hypo(t=1, 0){x=HH*t; y=RR*t; label=1;}
border Vert(t=0, RR){x=HH; y=t; label=2;}
mesh Th2 = buildmesh(Taxe(HH*nn) + Hypo(sqrt(HH*HH+RR*RR)*nn) + Vert(RR*nn));
plot(Th2, wait=true);

// 3D mesh
real h = 1./nn;
int MaxLayersT = (int(2*pi*RR/h)/4)*4;//number of layers
real zminT = 0;
real zmaxT = 2*pi; //height 2*pi
func fx = y*cos(z);
func fy = y*sin(z);
func fz = x;
int[int] r1T = [0,0], r2T = [0,0,2,2], r4T = [0,2];
//trick function:
//The function defined the proportion
//of number layer close to axis with reference MaxLayersT
func deg = max(.01, y/max(x/HH, 0.4)/RR);
mesh3 Th3T = buildlayers(Th2, coef=deg, MaxLayersT,
    zbound=[zminT, zmaxT], transfo=[fx, fy, fz],
    facemerge=0, region=r1T, labelmid=r2T);
medit("cone", Th3T);
```

Fig. 3.27: The mesh of a cone made with `cone.edp`

Tip: Buildlayer mesh

```
load "msh3"
load "TetGen"
load "medit"

// Parameters
```

(continues on next page)

(continued from previous page)

```
int C1 = 99;
int C2 = 98;

// 2D mesh
border C01(t=0, pi){x=t; y=0; label=1;}
border C02(t=0, 2*pi){ x=pi; y=t; label=1;}
border C03(t=0, pi){ x=pi-t; y=2*pi; label=1;}
border C04(t=0, 2*pi){ x=0; y=2*pi-t; label=1;}

border C11(t=0, 0.7){x=0.5+t; y=2.5; label=C1;}
border C12(t=0, 2){x=1.2; y=2.5+t; label=C1;}
border C13(t=0, 0.7){x=1.2-t; y=4.5; label=C1;}
border C14(t=0, 2){x=0.5; y=4.5-t; label=C1;}

border C21(t=0, 0.7){x=2.3+t; y=2.5; label=C2;}
border C22(t=0, 2){x=3; y=2.5+t; label=C2;}
border C23(t=0, 0.7){x=3-t; y=4.5; label=C2;}
border C24(t=0, 2){x=2.3; y=4.5-t; label=C2;}

mesh Th = buildmesh(C01(10) + C02(10) + C03(10) + C04(10)
    + C11(5) + C12(5) + C13(5) + C14(5)
    + C21(-5) + C22(-5) + C23(-5) + C24(-5));

mesh Ths = buildmesh(C01(10) + C02(10) + C03(10) + C04(10)
    + C11(5) + C12(5) + C13(5) + C14(5));

// Construction of a box with one hole and two regions
func zmin = 0.;
func zmax = 1.;
int MaxLayer = 10;

func XX = x*cos(y);
func YY = x*sin(y);
func ZZ = z;

int[int] r1 = [0, 41], r2 = [98, 98, 99, 99, 1, 56];
int[int] r3 = [4, 12];  //the triangles of uppper surface mesh
                        //generated by the triangle in the 2D region
                        //of mesh Th of label 4 as label 12
int[int] r4 = [4, 45];  //the triangles of lower surface mesh
                        //generated by the triangle in the 2D region
                        //of mesh Th of label 4 as label 45.

mesh3 Th3 = buildlayers(Th, MaxLayer, zbound=[zmin, zmax], region=r1,
    labelmid=r2, labelup=r3, labeldown=r4);
    medit("box 2 regions 1 hole", Th3);

// Construction of a sphere with TetGen
func XX1 = cos(y)*sin(x);
func YY1 = sin(y)*sin(x);
func ZZ1 = cos(x);

real[int] domain = [0., 0., 0., 0, 0.001];
string test = "paACQ";
cout << "test = " << test << endl;
mesh3 Th3sph = tetgtransfo(Ths, transfo=[XX1, YY1, ZZ1],
    switch=test, nbofregions=1, regionlist=domain);
```

(continues on next page)

(continued from previous page)

```
medit("sphere 2 regions", Th3sph);
```

Meshing examples

Tip: Lake

```
load "msh3"
load "medit"

// Parameters
int nn = 5;

// 2D mesh
border cc(t=0, 2*pi){x=cos(t); y=sin(t); label=1;}
mesh Th2 = buildmesh(cc(100));

// 3D mesh
int[int] rup = [0, 2], rlow = [0, 1];
int[int] rmid = [1, 1, 2, 1, 3, 1, 4, 1];
func zmin = 2-sqrt(4-(x*x+y*y));
func zmax = 2-sqrt(3.);

mesh3 Th = buildlayers(Th2, nn,
    coef=max((zmax-zmin)/zmax, 1./nn),
    zbound=[zmin,zmax],
    labelmid=rmid,
    labelup=rup,
    labeldown=rlow);

medit("Th", Th);
```

Tip: Hole region

```
load "msh3"
load "TetGen"
load "medit"

// 2D mesh
mesh Th = square(10, 20, [x*pi-pi/2, 2*y*pi]); // ]-pi/2, pi/2[X]0,2pi[

// 3D mesh
//parametrization of a sphere
func f1 = cos(x)*cos(y);
func f2 = cos(x)*sin(y);
func f3 = sin(x);
//partial derivative of the parametrization
func f1x = sin(x)*cos(y);
func f1y = -cos(x)*sin(y);
func f2x = -sin(x)*sin(y);
func f2y = cos(x)*cos(y);
func f3x = cos(x);
```

(continues on next page)

(continued from previous page)

```
func f3y = 0;
//M = DF^t DF
func m11 = f1x^2 + f2x^2 + f3x^2;
func m21 = f1x*f1y + f2x*f2y + f3x*f3y;
func m22 = f1y^2 + f2y^2 + f3y^2;

func perio = [[4, y], [2, y], [1, x], [3, x]];
real hh = 0.1;
real vv = 1/square(hh);
verbosity = 2;
Th = adaptmesh(Th, m11*vv, m21*vv, m22*vv, IsMetric=1, periodic=perio);
Th = adaptmesh(Th, m11*vv, m21*vv, m22*vv, IsMetric=1, periodic=perio);
plot(Th, wait=true);

//construction of the surface of spheres
real Rmin = 1.;
func f1min = Rmin*f1;
func f2min = Rmin*f2;
func f3min = Rmin*f3;

mesh3 Th3sph = movemesh23(Th, transfo=[f1min, f2min, f3min]);

real Rmax = 2.;
func f1max = Rmax*f1;
func f2max = Rmax*f2;
func f3max = Rmax*f3;

mesh3 Th3sph2 = movemesh23(Th, transfo=[f1max, f2max, f3max]);

//gluing meshse
mesh3 Th3 = Th3sph + Th3sph2;

cout << " TetGen call without hole " << endl;
real[int] domain2 = [1.5, 0., 0., 145, 0.001, 0.5, 0., 0., 18, 0.001];
mesh3 Th3fin = tetg(Th3, switch="paAAQYY", nbofregions=2, regionlist=domain2);
medit("Sphere with two regions", Th3fin);

cout << " TetGen call with hole " << endl;
real[int] hole = [0.,0.,0.];
real[int] domain = [1.5, 0., 0., 53, 0.001];
mesh3 Th3finhole = tetg(Th3, switch="paAAQYY",
    nbofholes=1, holelist=hole, nbofregions=1, regionlist=domain);
medit("Sphere with a hole", Th3finhole);
```

3.2.14 Build a 3d mesh of a cube with a balloon

First the `MeshSurface.idp` file to build boundary mesh of a Hexaedra and of a Sphere:

```
func mesh3 SurfaceHex (int[int] &N, real[int, int] &B, int[int, int] &L, int
↪orientation){
    real x0 = B(0, 0), x1 = B(0, 1);
    real y0 = B(1, 0), y1 = B(1, 1);
    real z0 = B(2, 0), z1 = B(2, 1);

```

(continues on next page)

(continued from previous page)

```
    int nx = N[0], ny = N[1], nz = N[2];

    mesh Thx = square(ny, nz, [y0+(y1-y0)*x, z0+(z1-z0)*y]);
    mesh Thy = square(nx, nz, [x0+(x1-x0)*x, z0+(z1-z0)*y]);
    mesh Thz = square(nx, ny, [x0+(x1-x0)*x, y0+(y1-y0)*y]);

    int[int] refx = [0, L(0,0)], refX = [0, L(0,1)]; //Xmin, Ymax faces labels renumbering
    int[int] refy = [0, L(1,0)], refY = [0, L(1,1)]; //Ymin, Ymax faces labesl renumbering
    int[int] refz = [0, L(2,0)], refZ = [0, L(2,1)]; //Zmin, Zmax faces labels renumbering

    mesh3 Thx0 = movemesh23(Thx, transfo=[x0, x, y], orientation=-orientation, label=refx);
    mesh3 Thx1 = movemesh23(Thx, transfo=[x1, x, y], orientation=+orientation, label=refX);
    mesh3 Thy0 = movemesh23(Thy, transfo=[x, y0, y], orientation=+orientation, label=refy);
    mesh3 Thy1 = movemesh23(Thy, transfo=[x, y1, y], orientation=-orientation, label=refY);
    mesh3 Thz0 = movemesh23(Thz, transfo=[x, y, z0], orientation=-orientation, label=refz);
    mesh3 Thz1 = movemesh23(Thz, transfo=[x, y, z1], orientation=+orientation, label=refZ);
    mesh3 Th = Thx0 + Thx1 + Thy0 + Thy1 + Thz0 + Thz1;

    return Th;
}

func mesh3 Sphere (real R, real h, int L, int orientation){
    mesh Th=square(10, 20, [x*pi-pi/2, 2*y*pi]); //]-pi/2, pi/2[X]0,2pi[

    func f1 = cos(x)*cos(y);
    func f2 = cos(x)*sin(y);
    func f3 = sin(x);

    func f1x = sin(x)*cos(y);
    func f1y = -cos(x)*sin(y);
    func f2x = -sin(x)*sin(y);
    func f2y = cos(x)*cos(y);
    func f3x = cos(x);
    func f3y = 0;

    func m11 = f1x^2 + f2x^2 + f3x^2;
    func m21 = f1x*f1y + f2x*f2y + f3x*f3y;
    func m22 = f1y^2 + f2y^2 + f3y^2;

    func perio = [[4, y], [2, y], [1, x], [3, x]]; //to store the periodic condition

    real hh = h/R; //hh mesh size on unite sphere
    real vv = 1/square(hh);
    Th = adaptmesh(Th, m11*vv, m21*vv, m22*vv, IsMetric=1, periodic=perio);
    Th = adaptmesh(Th, m11*vv, m21*vv, m22*vv, IsMetric=1, periodic=perio);
    Th = adaptmesh(Th, m11*vv, m21*vv, m22*vv, IsMetric=1, periodic=perio);
    Th = adaptmesh(Th, m11*vv, m21*vv, m22*vv, IsMetric=1, periodic=perio);
    int[int] ref = [0, L];
```

(continues on next page)

(a) The surface mesh of the hex with internal sphere **(b)** The tetrahedral mesh of the cube with internal ball

Fig. 3.28: Cube sphere

(continued from previous page)

```

    mesh3 ThS = movemesh23(Th, transfo=[f1*R, f2*R, f3*R], orientation=orientation,␣
↪refface=ref);

    return ThS;
}
```

The test of the two functions and the call to `TetGen` mesh generator:

```
load "msh3"
load "TetGen"
load "medit"
include "MeshSurface.idp"

// Parameters
real hs = 0.1; //mesh size on sphere
int[int] N = [20, 20, 20];
real [int,int] B = [[-1, 1], [-1, 1], [-1, 1]];
int [int,int] L = [[1, 2], [3, 4], [5, 6]];

// Mesh
mesh3 ThH = SurfaceHex(N, B, L, 1);
mesh3 ThS = Sphere(0.5, hs, 7, 1);

mesh3 ThHS = ThH + ThS;
medit("Hex-Sphere", ThHS);

real voltet = (hs^3)/6.;
cout << "voltet = " << voltet << endl;
real[int] domain = [0, 0, 0, 1, voltet, 0, 0, 0.7, 2, voltet];
mesh3 Th = tetg(ThHS, switch="pqaAAYYQ", nbofregions=2, regionlist=domain);
medit("Cube with ball", Th);
```

Medit

The keyword `medit` allows to display a mesh alone or a mesh and one or several functions defined on the mesh using the Pascal Frey's freeware `medit`. `medit` opens its own window and uses OpenGL extensively. Naturally to use this command `medit` must be installed.

A vizualisation with `medit` of scalar solutions $f1$ and $f2$ continuous, piecewise linear and known at the vertices of the mesh Th is obtained using:

```
medit("sol1 sol2", Th, f1, f2, order=1);
```

The first plot named `sol1` display f1. The second plot names `sol2` display f2.

The arguments of the function `medit` are the name of the differents scenes (separated by a space) of `medit`, a mesh and solutions.

Each solution is associated with one scene. The scalar, vector and symmetric tensor solutions are specified in the format described in the section dealing with the keyword `savesol`.

The parameters of this command line are :

- **`order`= 0 if the solution is given at the center of gravity of elements.** 1 is the solution is given at the vertices of elements.
- **`meditff`= set the name of execute command of medit.** By default, this string is medit.
- `save`= set the name of a file `.sol` or `.solb` to save solutions.

This command line allows also to represent two differents meshes and solutions on them in the same windows. The nature of solutions must be the same. Hence, we can vizualize in the same window the different domains in a domain decomposition method for instance. A vizualisation with medit of scalar solutions $h1$ and $h2$ at vertices of the mesh Th1 and Th2 respectively are obtained using:

```
medit("sol2domain", Th1, h1, Th2, h2, order=1);
```

Tip: Medit

```
load "medit"

// Initial Problem:
// Resolution of the following EDP:
// -Delta u_s = f on \Omega = { (x,y) | 1 <= sqrt(x^2+y^2) <= 2 }
// -Delta u_1 = f1 on \Omega_1 = { (x,y) | 0.5 <= sqrt(x^2+y^2) <= 1. }
// u = 1 on Gamma
// Null Neumman condition on Gamma_1 and on Gamma_2
// We find the solution u by solving two EDP defined on domain Omega and Omega_1
// This solution is visualize with medit

verbosity=3;

// Mesh
border Gamma(t=0, 2*pi){x=cos(t); y=sin(t); label=1;};
border Gamma1(t=0, 2*pi){x=2*cos(t); y=2*sin(t); label=2;};
border Gamma2(t=0, 2*pi){x=0.5*cos(t); y=0.5*sin(t); label=3;};

mesh Th = buildmesh(Gamma1(40) + Gamma(-40)); //Omega
mesh Th1 = buildmesh(Gamma(40) + Gamma2(-40)); //Omega_1

// Fespace
```

(continues on next page)

(continued from previous page)

```
fespace Vh(Th, P2);
func f = sqrt(x*x + y*y);
Vh us, v;

fespace Vh1(Th1, P2);
func f1 = 10*sqrt(x*x+y*y);
Vh1 u1, v1;

// Macro
macro Grad2(us) [dx(us), dy(us)] // EOM

// Problem
problem Lap2dOmega (us, v, init=false)
    = int2d(Th)(
        Grad2(v)' * Grad2(us)
    )
    - int2d(Th)(
        f*v
    )
    +on(1, us=1)
    ;

problem Lap2dOmega1 (u1, v1, init=false)
    = int2d(Th1)(
        Grad2(v1)' * Grad2(u1)
    )
    - int2d(Th1)(
        f1*v1
    )
    + on(1, u1=1)
    ;

// Solve
Lap2dOmega;
Lap2dOmega1;

// Plot with medit
medit("solution", Th, us, Th1, u1, order=1, save="testsavemedit.solb");
```

Mshmet

Mshmet is a software developed by P. Frey that allows to compute an anisotropic metric based on solutions (i.e. Hessian-based). This software can return also an isotropic metric. Moreover, `mshmet` can also construct a metric suitable for levelset interface capturing. The solution can be defined on 2D or 3D structured/unstructured meshes. For example, the solution can be an error estimate of a FE solution.

Solutions for `mshmet` are given as an argument. The solution can be a `func`, a vector `func`, a symmetric tensor, a `fespace` function, a `fespace` vector function and a `fespace` symmetric tensor. The symmetric tensor argument is defined as this type of data for datasol argument. This software accepts more than one solution.

For example, the metric M computed with `mshmet` for the solution u defined on the mesh Th is obtained by writing:

```
fespace Vh(Th, P1);
Vh u; //a scalar fespace function
real[int] M = mshmet(Th, u);
```

The parameters of the keyword `mshmet` are :

- `normalization` = (b) do a normalization of all solution in $[0, 1]$.
- `aniso` = (b) build anisotropic metric if 1 (default 0: isotropic)
- `levelset` = (b) build metric for levelset method (default: `false`)
- `verbosity` = (l) level of verbosity
- `nbregul` = (l) number of regularization's iteration of solutions given (default 0).
- `hmin` = (d)
- `hmax` = (d)
- `err` = (d) level of error.
- `width` = (d) the width
- **`metric` = a vector of double.** This vector contains an initial metric given to `mshmet`. The structure of the metric vector is described in the next paragraph.
- **`loptions` = a vector of integer of size 7.** This vector contains the integer parameters of `mshmet` (for expert only).
  - loptions(0): normalization (default 1).
  - **loptions(1): isotropic parameters (default 0).** 1 for isotropic metric results otherwise 0.
  - **loptions(2): level set parameters (default 0).** 1 for building level set metric otherwise 0.
  - **loptions(3): debug parameters (default 0).** 1 for turning on debug mode otherwise 0.
  - loptions(4): level of verbosity (default 10).
  - loptions(5): number of regularization's iteration of solutions given (default 0).
  - **loptions(6): previously metric parameter (default 0).** 1 for using previous metric otherwise 0.
- **`doptions`= a vector of double of size 4.** This vector contains the real parameters of `mshmet` (for expert only).
  - doptions(0): hmin : min size parameters (default 0.01).
  - doptions(1): hmax : max size parameters (default 1.0).
  - doptions(2): eps : tolerance parameters (default 0.01).
  - doptions(2): width : relative width for Level Set ($0 < w < 1$) (default 0.05).

The result of the keyword `mshmet` is a `real[int]` which contains the metric computed by `mshmet` at the different vertices V_i of the mesh.

With nv is the number of vertices, the structure of this vector is:

$$M_{iso} = (m(V_0), m(V_1), \ldots, m(V_{nv}))^t$$

for a isotropic metric m. For a symmetric tensor metric $h = \begin{pmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{pmatrix}$, the parameters `metric` is:

$$M_{aniso} = (H(V_0), \ldots, H(V_{nv}))^t$$

where $H(V_i)$ is the vector of size 6 defined by $[m11, m21, m22, m31, m32, m33]$

Tip: mshmet

```
load "mshmet"
load "medit"
load "msh3"

// Parameters
real error = 0.01;
func zmin = 0;
func zmax = 1;
int MaxLayer = 10;

// Mesh
border a(t=0, 1.0){x=t; y=0; label=1;};
border b(t=0, 0.5){x=1; y=t; label=2;};
border c(t=0, 0.5){x=1-t; y=0.5; label=3;};
border d(t=0.5, 1){x=0.5; y=t; label=4;};
border e(t=0.5, 1){x=1-t; y=1; label=5;};
border f(t=0.0, 1){x=0; y=1-t; label=6;};
mesh Th = buildmesh(a(6) + b(4) + c(4) + d(4) + e(4) + f(6));
mesh3 Th3 = buildlayers(Th, MaxLayer, zbound=[zmin, zmax]);

// Fespace
fespace Vh3(Th3, P2);
Vh3 u3, v3;

fespace Vh3P1(Th3, P1);
Vh3P1 usol;

// Problem
problem Problem2(u3, v3, solver=sparsesolver)
    = int3d(Th3)(
        u3*v3*1.0e-10
        + dx(u3)*dx(v3)
        + dy(u3)*dy(v3)
        + dz(u3)*dz(v3)
    )
    - int3d(Th3)(
        v3
    )
    +on(0, 1, 2, 3, 4, 5, 6, u3=0)
    ;

// Solve
Problem2;
cout << u3[].min << " " << u3[].max << endl;

medit("Sol", Th3, u3);

real[int] bb = mshmet(Th3,u3);
cout << "Metric:" << bb << endl;
for (int ii = 0; ii < Th3.nv; ii++)
    usol[][ii] = bb[ii];

medit("Metric", Th3, usol);
```

FreeYams

FreeYams is a surface mesh adaptation software which is developed by P. Frey. This software is a new version of yams. The adapted surface mesh is constructed with a geometric metric tensor field. This field is based on the intrinsic properties of the discrete surface.

Also, this software allows to construct a simplification of a mesh. This decimation is based on the Hausdorff distance between the initial and the current triangulation. Compared to the software yams, FreeYams can be used also to produce anisotropic triangulations adapted to levelset simulations. A technical report on freeYams documentation is available here.

To call FreeYams in **FreeFEM**, we used the keyword `freeyams`. The arguments of this function are the initial mesh and/or metric. The metric with `freeyams` are a `func`, a `fespace` function, a symmetric tensor function, a symmetric tensor `fespace` function or a vector of double (`real[int]`). If the metric is a vector of double, this data must be given in `metric` parameter. Otherwise, the metric is given in the argument.

For example, the adapted mesh of `Thinit` defined by the metric u defined as `fespace` function is obtained by writing:

```
fespace Vh(Thinit, P1);
Vh u;
mesh3 Th = freeyams(Thinit, u);
```

The symmetric tensor argument for `freeyams` keyword is defined as this type of data for datasol argument.

- `aniso=` (b) aniso or iso metric (default 0, iso)
- `mem=` (l) memory of for freeyams in Mb (default -1, freeyams choose)
- `hmin=` (d)
- `hmax=` (d)
- `gradation=` (d)
- `option=` (l)
  - 0 : mesh optimization (smoothing+swapping)
  - 1 : decimation+enrichment adaptated to a metric map. (default)
  - -1 : decimation adaptated to a metric map.
  - 2 : decimation+enrichment with a Hausdorff-like method
  - -2 : decimation with a Hausdorff-like method
  - 4 : split triangles recursively.
  - 9 : No-Shrinkage Vertex Smoothing
- `ridgeangle=` (d)
- `absolute=` (b)
- `verbosity=` (i)
- **`metric=` vector expression.** This parameters contains the metric at the different vertices on the initial mesh. With nv is the number of vertices, this vector is:

$$M_{iso} = (m(V_0), m(V_1), \ldots, m(V_{nv}))^t$$

for a scalar metric m. For a symmetric tensor metric $h = \begin{pmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{pmatrix}$, the parameters `metric` is:

$$M_{aniso} = (H(V_0), \ldots, H(V_{nv}))^t$$

where $H(V_i)$ is the vector of size 6 defined by $[m11, m21, m22, m31, m32, m33]$

- **`loptions`= a vector of integer of size 13.** This vectors contains the integer options of FreeYams. (just for the expert)
  - **loptions(0): anisotropic parameter (default 0).** If you give an anisotropic metric 1 otherwise 0.
  - **loptions(1): Finite Element correction parameter (default 0).** 1 for *no* Finite Element correction otherwise 0.
  - **loptions(2): Split multiple connected points parameter (default 1).** 1 for splitting multiple connected points otherwise 0.
  - loptions(3): maximum value of memory size in Mbytes (default -1: the size is given by freeyams).
  - **loptions(4): set the value of the connected component which we want to obtain.** (Remark: freeyams give an automatic value at each connected component).
  - loptions(5): level of verbosity
  - **loptions(6): Create point on straight edge (no mapping) parameter (default 0).** 1 for creating point on straight edge otherwise 0.
  - **loptions(7): validity check during smoothing parameter.** This parameter is only used with No-Shrinkage Vertex Smoothing optimization (optimization option parameter 9). 1 for No validity checking during smoothing otherwise 0.
  - loptions(8): number of desired's vertices (default -1).
  - loptions(9): number of iteration of optimizations (default 30).
  - **loptions(10): no detection parameter (default 0).** 1 for detecting the ridge on the mesh otherwise 0. The ridge definition is given in the parameter doptions(12).
  - **loptions(11): no vertex smoothing parameter (default 0).** 1 for smoothing the vertices otherwise 0.
  - loptions(12): Optimization level parameter (default 0).
  - 0 : mesh optimization (smoothing+swapping)
  - 1 : decimation+enrichment adaptated to a metric map.
  - -1: decimation adaptated to a metric map.
  - 2 : decimation+enrichment with a Hausdorff-like method
  - -2: decimation with a Hausdorff-like method
  - 4 : split triangles recursively.
  - 9 : No-Shrinkage Vertex Smoothing
- **`doptions`= a vector of double of size 11.** This vectors contains the real options of freeyams.
  - doptions(0): Set the geometric approximation (Tangent plane deviation) (default 0.01).
  - doptions(1): Set the lamda parameter (default -1).
  - doptions(2): Set the mu parmeter (default -1).

- doptions(3): Set the gradation value (Mesh density control) (default 1.3).
- doptions(4): Set the minimal size(hmin) (default -2.0: the size is automatically computed).
- doptions(5): Set the maximal size(hmax) (default -2.0: the size is automatically computed).
- doptions(6): Set the tolerance of the control of Chordal deviation (default -2.0).
- doptions(7): Set the quality of degradation (default 0.599).
- doptions(8): Set the declic parameter (default 2.0).
- doptions(9): Set the angular walton limitation parameter (default 45 degree).
- doptions(10): Set the angular ridge detection (default 45 degree).

Tip: freeyams

```
load "msh3"
load "medit"
load "freeyams"

// Parameters
int nn = 20;
real zmin = 0;
real zmax = 1;

// Mesh
mesh Th2 = square(nn, nn);
int[int] rup = [0, 2], rdown = [0, 1];
int[int] rmid = [1, 1, 2, 1, 3, 1, 4, 1];
mesh3 Th = buildlayers(Th2, nn, zbound=[zmin, zmax], reffacemid=rmid, reffaceup=rup,
↪reffacelow=rdown);
mesh3 Th3 = freeyams(Th);

medit("SurfaceMesh", Th3);
```

mmg3d

Todo: mmg3d-v4.0

Mmg3d is a 3D remeshing software developed by C. Dobrzynski and P. Frey.

This software allows to remesh an initial mesh made of tetrahedra. This initial mesh is adapted to a geometric metric tensor field or to a displacement vector (moving rigid body). The metric can be obtained with *mshmet*.

Note:

- If no metric is given, an isotropic metric is computed by analyzing the size of the edges in the initial mesh.
- If a displacement is given, the vertices of the surface triangles are moved without verifying the geometrical structure of the new surface mesh.

The parameters of `mmg3d` are :

- **`options=` vector expression.** This vector contains the option parameters of `mmg3d`. It is a vector of 6 values, with the following meaning:
  - Optimization parameters : (default 1)

    0 : mesh optimization.

    1 : adaptation with metric (deletion and insertion vertices) and optimization.

    -1 : adaptation with metric (deletion and insertion vertices) without optimization.

    4 : split tetrahedra (be careful modify the surface).

    9 : moving mesh with optimization.

    -9 : moving mesh without optimization.
  - Debug mode : (default 0)

    1 : turn on debug mode.

    0 : otherwise.
  - Specify the size of bucket per dimension (default 64)
  - Swapping mode : (default 0)

    1 : no edge or face flipping.

    0 : otherwise.
  - Insert points mode : (default 0)

    1 : no edge splitting or collapsing and no insert points.

    0 : otherwise.

  5. Verbosity level (default 3)
- **`memory=` integer expression.** Set the maximum memory size of new mesh in Mbytes. By default the number of maximum vertices, tetrahedra and triangles are respectively 500 000, 3000 000, 100000 which represent approximately a memory of 100 Mo.
- **`metric=` vector expression.** This vector contains the metric given at `mmg3d`. It is a vector of size nv or 6 nv respectively for an isotropic and anisotropic metric where nv is the number of vertices in the initial mesh. The structure of `metric` vector is described in the *mshmet*.
- `displacement=` $[\Phi 1, \Phi 2, \Phi 3]$ set the displacement vector of the initial mesh $\mathbf{\Phi}(\mathbf{x}, \mathbf{y}) = [\Phi 1(x, y), \Phi 2(x, y), \Phi 3(x, y)]$.
- **`displVect=` sets the vector displacement in a vector expression.** This vector contains the displacement at each point of the initial mesh. It is a vector of size 3 nv.

Tip: mmg3d

```
load "msh3"
load "medit"
load "mmg3d"
include "Cube.idp"

// Parameters
int n = 6;
int[int] Nxyz = [12, 12, 12];
real [int, int] Bxyz = [[0., 1.], [0., 1.], [0., 1.]];
int [int, int] Lxyz = [[1, 1], [2, 2], [2, 2]];
```

(continues on next page)

(continued from previous page)

```

// Mesh
mesh3 Th = Cube(Nxyz, Bxyz, Lxyz);

real[int] isometric(Th.nv);
for (int ii = 0; ii < Th.nv; ii++)
    isometric[ii] = 0.17;

mesh3 Th3 = mmg3d(Th, memory=100, metric=isometric);

// Plot
medit("Initial", Th);
medit("Isometric", Th3);
```

Tip: Falling spheres

```
load "msh3"
load "TetGen"
load "medit"
load "mmg3d"
include "MeshSurface.idp"

// Parameters
real hs = 0.8;
int[int] N = [4/hs, 8/hs, 11.5/hs];
real [int, int] B = [[-2, 2], [-2, 6], [-10, 1.5]];
int [int, int] L = [[311, 311], [311, 311], [311, 311]];

int[int] opt = [9, 0, 64, 0, 0, 3];
real[int] vit=[0, 0, -0.3];
func zero = 0.;
func dep = vit[2];

// Mesh
mesh3 ThH = SurfaceHex(N, B, L, 1);
mesh3 ThSg = Sphere(1, hs, 300, -1);
mesh3 ThSd = Sphere(1, hs, 310, -1);
ThSd = movemesh3(ThSd, transfo=[x, 4+y, z]);
mesh3 ThHS = ThH + ThSg + ThSd;//gluing surface meshes
medit("ThHS", ThHS);

real voltet = (hs^3)/6.;
real[int] domain = [0, 0, -4, 1, voltet];
real [int] holes = [0, 0, 0, 0, 4, 0];
mesh3 Th = tetg(ThHS, switch="pqaAAYYQ", nbofregions=1, regionlist=domaine,
↪nbofholes=2, holelist=holes);
medit("Box-With-two-Ball", Th);

// Fespace
fespace Vh(Th, P1);
Vh uh,vh;

// Macro
macro Grad(u) [dx(u),dy(u),dz(u)]
```

(continues on next page)

(continued from previous page)

```

// Problem
problem Lap (uh, vh, solver=CG)
    = int3d(Th)(
        Grad(uh)' * Grad(vh)
    )
    + on(310, 300, uh=dep)
    + on(311, uh=0.)
    ;

// Falling loop
for(int it = 0; it < 29; it++){
    cout << " ITERATION " << it << endl;

    // Solve
    Lap;

    // Plot
    plot(Th, uh);

    // Sphere falling
    Th = mmg3d(Th, options=opt, displacement=[zero, zero, uh], memory=1000);
}
```

A first 3d isotrope mesh adaptation process

Tip: Adaptation 3D

```
load "msh3"
load "TetGen"
load "mshmet"
load "medit"

// Parameters
int nn = 6;
int[int] l1111 = [1, 1, 1, 1]; //labels
int[int] l01 = [0, 1];
int[int] l11 = [1, 1];

real errm = 1e-2; //level of error

// Mesh
mesh3 Th3 = buildlayers(square(nn, nn, region=0, label=l1111),
nn, zbound=[0, 1], labelmid=l11, labelup=l01, labeldown=l01);

Th3 = trunc(Th3, (x<0.5) | (y < 0.5) | (z < 0.5), label=1); //remove the ]0.5,1[^3 cube

// Fespace
fespace Vh(Th3, P1);
Vh u, v, usol, h;

// Macro
```

(continues on next page)

(continued from previous page)

```
macro Grad(u) [dx(u), dy(u), dz(u)] // EOM

// Problem
problem Poisson (u, v, solver=CG)
    = int3d(Th3)(
        Grad(u)' * Grad(v)
    )
    - int3d(Th3)(
        1*v
    )
    + on(1, u=0)
    ;

// Loop
for (int ii = 0; ii < 5; ii++){
    // Solve
    Poisson;
    cout << "u min, max = " << u[].min << " "<< u[].max << endl;

    h=0.; //for resizing h[] because the mesh change
    h[] = mshmet(Th3, u, normalization=1, aniso=0, nbregul=1, hmin=1e-3, hmax=0.3, err=errm);
    cout << "h min, max = " << h[].min << " "<< h[].max << " " << h[].n << " " << Th3.nv << endl;
    plot(u, wait=true);

    errm *= 0.8; //change the level of error
    cout << "Th3 " << Th3.nv < " " << Th3.nt << endl;
    Th3 = tetgreconstruction(Th3, switch="raAQ", sizeofvolume=h*h*h/6.); //rebuild mesh
    medit("U-adap-iso-"+ii, Th3, u, wait=true);
}
```

Build a 2d mesh from a isoline

The idea is to get the discretization of an isoline of fluid meshes, this tool can be useful to construct meshes from image. First, we give an example of the isovalue meshes 0.2 of analytical function $\sqrt{(x-1/2)^2+(y-1/2)^2}$, on unit square.

```
load "isoline"

real[int,int] xy(3, 1); //to store the isoline points
int[int] be(1); //to store the begin, end couple of lines
{
    mesh Th = square(10, 10);
    fespace Vh(Th, P1);
    Vh u = sqrt(square(x-0.5) + square(y-0.5));
    real iso = 0.2 ;
    real[int] viso = [iso];
    plot(u, viso=viso,Th);//to see the iso line

    int nbc = isoline(Th, u, xy, close=1, iso=iso, beginend=be, smoothing=0.1);
```

The `isoline` parameters are `Th` the mesh, the expression u, the bidimentionnal array `xy` to store the list coordinate of the points. The list of named parameter are :

- `iso`= value of the isoline to compute (0 is the default value)
- `close`= close the isoline with the border (default `true`), we add the part of the mesh border such the value is less than the isovalue
- **`smoothing`= number of smoothing process is the $l^r s$ where l is the length of the current line component, r the ratio, s is s**
  The smoothing default value is 0.
- `ratio`= the ratio (1 by default).
- `eps`= relative ε (default 1e-10)
- `beginend`= array to get begin, end couple of each of sub line (resize automatically)
- `file`= to save the data curve in data file for gnuplot

In the array `xy` you get the list of vertices of the isoline, each connex line go from $i = i_0^c, \ldots, i_1^c - 1$ with $i_0^c = be(2*c)$ $i_1^c = be(2*c+1)$, and where $x_i = xy(0,i), y_i = yx(1,i), l_i = xy(2,i)$.

Here l_i is the length of the line (the origin of the line is point i_0^c).

The sense of the isoline is such that the upper part is at the left size of the isoline. So here : the minimum is a point $0.5, 05$ so the curve 1 turn in the clockwise sense, the order of each component are sort such that the number of point by component is decreasing.

```
      cout << "Number of the line component = " << nbc << endl;
      cout << "Number of points = " << xy.m << endl;
      cout << "be = " << be << endl;

      // shows the lines component
      for (int c = 0; c < nbc; ++c){
         int i0 = be[2*c], i1 = be[2*c+1]-1;
         cout << "Curve " << c << endl;
         for(int i = i0; i <= i1; ++i)
            cout << "x= " << xy(0,i) << " y= " << xy(1,i) << " s= " << xy(2, i) << endl;
         plot([xy(0, i0:i1), xy(1, i0:i1)], wait=true, viso=viso, cmm=" curve "+c);
   }
}

cout << "length of last curve = " << xy(2, xy.m-1) << endl;
```

We also have a new function to easily parametrize a discrete curve defined by the couple be, xy.

```
border Curve0(t=0, 1){
    int c=0; //component 0
    int i0=be[2*c], i1=be[2*c+1]-1;
    P=Curve(xy, i0, i1, t); //Curve 0
    label=1;
}

border Curve1(t=0, 1){
    int c=1; //component 1
    int i0=be[2*c], i1=be[2*c+1]-1;
    P=Curve(xy, i0, i1, t); //Curve 1
    label=1;
}

plot(Curve1(100)); //show curve
mesh Th = buildmesh(Curve1(-100));
plot(Th, wait=true);
```

Secondly, we use this idea to build meshes from an image, we use the plugins `ppm2rnm` to read `pgm` a gray scale image and then we extract the gray contour at level 0.25.

Tip: Leman lake

```
load "ppm2rnm"
load "isoline"

// Parameters
string leman = "LemanLake.pgm";
real AreaLac = 580.03; //in km^2
real hsize = 5;
real[int, int] Curves(3, 1);
int[int] be(1);
int nc; //nb of curve
{
    real[int, int] ff1(leman); //read image
    //and set it in a rect. array
    int nx = ff1.n, ny = ff1.m;
    //build a Cartesian mesh such that the origin is in the right place.
    mesh Th = square(nx-1, ny-1, [(nx-1)*(x), (ny-1)*(1-y)]);
    //warning the numbering of the vertices (x,y) is
    //given by $i = x/nx + nx* y/ny $
    fespace Vh(Th, P1);
    Vh f1;
    f1[] = ff1; //transform array in finite element functions.
    nc = isoline(Th, f1, iso=0.25, close=1, Curves, beginend=be, smoothing=.1, ratio=0.5);
}

//The longest isoline: the lake
int ic0 = be(0), ic1 = be(1)-1;
plot([Curves(0, ic0:ic1), Curves(1, ic0:ic1)], wait=true);

int NC = Curves(2, ic1)/hsize;
real xl = Curves(0, ic0:ic1).max - 5;
real yl = Curves(1, ic0:ic1).min + 5;
border G(t=0, 1){P=Curve(Curves, ic0, ic1, t); label=1+(x>xl)*2+(y<yl);}
plot(G(-NC), wait=true);

mesh Th = buildmesh(G(-NC));
plot(Th, wait=true);

real scale = sqrt(AreaLac/Th.area);
Th = movemesh(Th, [x*scale, y*scale]);
cout << "Th.area = " << Th.area << " Km^2 " << " == " << AreaLac << " Km^2 " << endl;
plot(Th, wait=true, ps="leman.eps");
```

3.3 Finite element

As stated in *tutorials*, FEM approximates all functions w as:

$$w(x,y) \simeq w_0\phi_0(x,y) + w_1\phi_1(x,y) + \cdots + w_{M-1}\phi_{M-1}(x,y)$$

(a) The image of the Leman lake meshes

(b) The mesh of the lake

Fig. 3.29: Isoline

with finite element basis functions $\phi_k(x, y)$ and numbers w_k $(k = 0, \cdots, M-1)$. The functions $\phi_k(x, y)$ are constructed from the triangle T_{i_k}, and called *shape functions*.

In **FreeFEM**, the finite element space:

$$V_h = \{w \mid w_0\phi_0 + w_1\phi_1 + \cdots + w_{M-1}\phi_{M-1},\ w_i \in \mathbb{R}\}$$

is easily created by:

```
fespace IDspace(IDmesh,<IDFE>);
```

or with ℓ pairs of periodic boundary conditions in 2D:

```
fespace IDspace(IDmesh,<IDFE>,
    periodic=[[la1, sa1], [lb1, sb1],
              ...
              [lak, sak], [lbk, sbl]]);
```

and in 3D:

```
fespace IDspace(IDmesh,<IDFE>,
    periodic=[[la1, sa1, ta1], [lb1, sb1, tb1],
              ...
              [lak, sak, tak], [lbk, sbl, tbl]]);
```

where `IDspace` is the name of the space (e.g. `Vh`), `IDmesh` is the name of the associated mesh and `<IDFE>` is an identifier of finite element type.

In 2D we have a pair of periodic boundary conditions, if $[la_i, sa_i]$, $[lb_i, sb_i]$ is a pair of `int`, and the 2 labels la_i and lb_i refer to 2 pieces of boundary to be in equivalence.

If $[la_i, sa_i]$, $[lb_i, sb_i]$ is a pair of `real`, then sa_i and sb_i give two common abscissa on the two boundary curves, and two points are identified as one if the two abscissa are equal.

In 2D, we have a pair of periodic boundary conditions, if $[la_i, sa_i, ta_i]$, $[lb_i, sb_i, tb_i]$ is a pair of `int`, the 2 labels la_i and lb_i define the 2 pieces of boundary to be in equivalence.

If $[la_i, sa_i, ta_i]$, $[lb_i, sb_i, tb_i]$ is a pair of `real`, then sa_i, ta_i and sb_i, tb_i give two common parameters on the two boundary surfaces, and two points are identified as one if the two parameters are equal.

Note: The 2D mesh of the two identified borders must be the same, so to be sure, use the parameter `fixedborder=true` in `buildmesh` command (see *fixedborder*).

As of today, the known types of finite elements are:

- `[P0, P03d]` piecewise constant discontinuous finite element (2d, 3d), the degrees of freedom are the barycenter element value.

$$\mathbb{P}_h^0 = \left\{ v \in L^2(\Omega) \,\middle|\, \text{for all } K \in \mathcal{T}_h \;\; \text{there is } \alpha_K \in \mathbb{R} : \;\; v_{|K} = \alpha_K \right\} \tag{3.2}$$

- `[P1, P13d]` piecewise linear continuous finite element (2d, 3d), the degrees of freedom are the vertices values.

$$\mathbb{P}_h^1 = \left\{ v \in H^1(\Omega) \,\middle|\, \forall K \in \mathcal{T}_h,\; v_{|K} \in P_1 \right\} \tag{3.3}$$

- `[P1dc]` piecewise linear discontinuous finite element

$$\mathbb{P}_{dc|h}^1 = \left\{ v \in L^2(\Omega) \,\middle|\, \forall K \in \mathcal{T}_h,\; v_{|K} \in P_1 \right\} \tag{3.4}$$

Warning: Due to an interpolation problem, the degree of freedom is not the vertices but three vertices which move inside $T(X) = G + .99(X - G)$ where G is the barycenter.

- `[P1b, P1b3d]` piecewise linear continuous finite element plus bubble (2d, 3d)

  The 2D Case:

$$\mathbb{P}_{b|h}^1 = \left\{ v \in H^1(\Omega) \,\middle|\, \forall K \in \mathcal{T}_h,\; v_{|K} \in P_1 \oplus \mathrm{Span}\{\lambda_0^K \lambda_1^K \lambda_2^K\} \right\} \tag{3.5}$$

  The 3D Case:

$$\mathbb{P}_{b|h}^1 = \left\{ v \in H^1(\Omega) \,\middle|\, \forall K \in \mathcal{T}_h,\; v_{|K} \in P_1 \oplus \mathrm{Span}\{\lambda_0^K \lambda_1^K \lambda_2^K \lambda_3^K\} \right\} \tag{3.6}$$

  where $\lambda_i^K, i = 0, .., d$ are the $d + 1$ barycentric coordinate functions of the element K (triangle or tetrahedron).

- `P1bl,P1bl3d` piecewise linear continuous finite element plus linear bubble (2d, 3d).

  The bubble is built by splitting the K, a barycenter in $d + 1$ sub element. (need `load "Element_P1bl"`)

- `[P2, P23d]` piecewise P_2 continuous finite element (2d, 3d)

$$\mathbb{P}_h^2 = \left\{ v \in H^1(\Omega) \,\middle|\, \forall K \in \mathcal{T}_h,\; v_{|K} \in P_2 \right\}$$

  where P_2 is the set of polynomials of $\mathbb{R}^2$ of degrees ≤ 2.

- `[P2b]` piecewise P_2 continuous finite element plus bubble

$$\mathbb{P}_h^2 = \left\{ v \in H^1(\Omega) \,\middle|\, \forall K \in \mathcal{T}_h,\ v_{|K} \in P_2 \oplus \mathrm{Span}\{\lambda_0^K \lambda_1^K \lambda_2^K\} \right\}$$

- `[P2dc]` piecewise P_2 discontinuous finite element

$$\mathbb{P}_{dc|h}^2 = \left\{ v \in L^2(\Omega) \,\middle|\, \forall K \in \mathcal{T}_h,\ v_{|K} \in P_2 \right\}$$

> **Warning:** Due to an interpolation problem, the degree of freedom is not the six P2 nodes but six nodes which move inside $T(X) = G + .99(X - G)$ where G is the barycenter.

- `[P2h]` quadratic homogeneous continuous (without `P1`).
- `[P3]` piecewise P_3 continuous finite element (2d) (needs `load "Element_P3"`)

$$\mathbb{P}_h^3 = \left\{ v \in H^1(\Omega) \,\middle|\, \forall K \in \mathcal{T}_h,\ v_{|K} \in P_3 \right\}$$

  where P_3 is the set of polynomials of $\mathbb{R}^2$ of degrees ≤ 3.

- `[P3dc]` piecewise P_3 discontinuous finite element (2d) (needs `load "Element_P3dc"`)

$$\mathbb{P}_{dc|h}^3 = \left\{ v \in L^2(\Omega) \,\middle|\, \forall K \in \mathcal{T}_h,\ v_{|K} \in P_3 \right\}$$

  where P_3 is the set of polynomials of $\mathbb{R}^2$ of degrees ≤ 3.

- `[P4]` piecewise P_4 continuous finite element (2d) (needs `load "Element_P4"`)

$$\mathbb{P}_h^4 = \left\{ v \in H^1(\Omega) \,\middle|\, \forall K \in \mathcal{T}_h,\ v_{|K} \in P_4 \right\}$$

  where P_4 is the set of polynomials of $\mathbb{R}^2$ of degrees ≤ 4.

- `[P4dc]` piecewise P_4 discontinuous finite element (2d) (needs `load "Element_P4dc"`)

$$\mathbb{P}_{dc|h}^4 = \left\{ v \in L^2(\Omega) \,\middle|\, \forall K \in \mathcal{T}_h,\ v_{|K} \in P_3 \right\}$$

  where P_4 is the set of polynomials of $\mathbb{R}^2$ of degrees ≤ 3.

- `[P0Edge]` piecewise P_0 discontinuous finite element (2d) contained on each edge of the mesh.
- `[P1Edge]` piecewise P_1 discontinuous finite element (2d) (needs `load "Element_PkEdge"`) P_1 on each edge of the mesh.
- `[P2Edge]` piecewise P_2 discontinuous finite element (2d) (needs `load "Element_PkEdge"`) P_2 on each edge of the mesh.
- `[P3Edge]` piecewise P_3 discontinuous finite element (2d) (needs `load "Element_PkEdge"`) P_3 on each edge of the mesh.

- `[P4Edge]` piecewise P_4 discontinuous finite element (2d) (needs `load "Element_PkEdge"`) P_4 on each edge of the mesh.
- `[P5Edge]` piecewise P_5 discontinuous finite element (2d) (needs `load "Element_PkEdge"`) P_5 on each edge of the mesh.
- `[P2Morley]` piecewise P_2 non conform finite element (2d) (needs `load "Morley"`)

$$\mathbb{P}_h^2 = \left\{ v \in L^2(\Omega) \,\middle|\, \forall K \in \mathcal{T}_h,\ v_{|K} \in P_3, \left\{ \begin{array}{c} v \text{ continuous at vertices,} \\ \partial_n v \text{ continuous at middle of edge,} \end{array} \right. \right\}$$

where P_2 is the set of polynomials of $\mathbb{R}^2$ of degrees ≤ 2.

> **Warning:** To build the interplant of a function u (scalar) for this finite element, we need the function and 2 partial derivatives (u, u_x, u_y), creating this vectorial finite element with 3 components (u, u_x, u_y).

See our example for solving the BiLaplacien problem:

```
load "Morley"

// Parameters
int nn = 10;
real h = 0.01;

real f = 1;

// Mesh
mesh Th = square(nn, nn);
Th = adaptmesh(Th, h, IsMetric=1);

// Fespace
fespace Vh(Th, P2Morley); //The Morley finite element space
Vh [u, ux, uy], [v, vx, vy];

// Macro
macro bilaplacien(u, v) (dxx(u)*dxx(v) + dyy(u)*dyy(v) + 2.
↪*dxy(u)*dxy(v)) //

// Problem
solve bilap ([u, ux, uy], [v, vx, vy])
    = int2d(Th)(
        bilaplacien(u, v)
    )
    - int2d(Th)(
        f*v
    )
    + on(1, 2, 3, 4, u=0, ux=0, uy=0)
    ;

// Plot
plot(u, cmm="u");
```

- `[HCT]` P_3 C^1 conforms finite element (2d) (needs `load "Element_HCT"`) one 3 sub triangles.

Lets call $\mathcal{T}_h^{\triangle}$ the sub mesh of $\mathcal{T}_h$ where all triangles are split in 3 at the barycenter.

$$\mathbb{P}_h^{HCT} = \left\{ v \in C^1(\Omega) \,\middle|\, \forall K \in \mathcal{T}_h^{\triangle},\ v_{|K} \in P_3 \right\}$$

where P_3 is the set of polynomials of $\mathbb{R}^2$ of degrees ≤ 3.

The degrees of freedom are the values of the normal derivative at the mid-point of each edge [BERNADOU1980].

> **Warning:** To build the interplant of a function u (scalar) for this finite element, we need the function and 2 partial derivatives (u, u_x, u_y), creating this vectorial finite element with 3 components (u, u_x, u_y) like in previous finite element.

- `[P2BR]` (needs `load "BernadiRaugel"`) the Bernadi Raugel Finite Element is a Vectorial element (2d) with 2 components, see [BERNARDI1985].

  It is a 2D coupled Finite Element, where the Polynomial space is P_1^2 with 3 normal bubble edge functions (P_2). There are 9 degrees of freedom:

  - 2 components at each of the 3 vertices and
  - the 3 flux on the 3 edges.

- `[RT0, RT03d]` Raviart-Thomas finite element of degree 0.

  The 2D Case:

$$RT0_h = \left\{ \mathbf{v} \in H(\text{div}) \,\middle|\, \forall K \in \mathcal{T}_h,\ \mathbf{v}_{|K}(x,y) = \left| \begin{smallmatrix} \alpha_K^1 \\ \alpha_K^2 \end{smallmatrix} \right. + \beta_K \left| \begin{smallmatrix} x \\ y \end{smallmatrix} \right. \right\} \tag{3.7}$$

  The 3D Case:

$$RT0_h = \left\{ \mathbf{v} \in H(\text{div}) \,\middle|\, \forall K \in \mathcal{T}_h,\ \mathbf{v}_{|K}(x,y,z) = \left| \begin{smallmatrix} \alpha_K^1 \\ \alpha_K^2 \\ \alpha_K^3 \end{smallmatrix} \right. + \beta_K \left| \begin{smallmatrix} x \\ y \\ z \end{smallmatrix} \right. \right\} \tag{3.8}$$

  where by writing div $\mathbf{w} = \sum_{i=1}^d \partial w_i / \partial x_i$ with $\mathbf{w} = (w_i)_{i=1}^d$:

$$H(\text{div}) = \left\{ \mathbf{w} \in L^2(\Omega)^d \,\middle|\, \text{div } \mathbf{w} \in L^2(\Omega) \right\}$$

  and where α_K^1, α_K^2, α_K^3, β_K are real numbers.

- `[RT0Ortho]` Raviart-Thomas Orthogonal, or Nedelec finite element type I of degree 0 in dimension 2

$$RT0Orthoh = \left\{ \mathbf{v} \in H(\text{curl}) \,\middle|\, \forall K \in \mathcal{T}_h,\ \mathbf{v}_{|K}(x,y) = \left| \begin{smallmatrix} \alpha_K^1 \\ \alpha_K^2 \end{smallmatrix} \right. + \beta_K \left| \begin{smallmatrix} -y \\ x \end{smallmatrix} \right. \right\} \tag{3.9}$$

- `[Edge03d]` 3d Nedelec finite element or Edge Element of degree 0.

$$Edge0_h = \left\{ \mathbf{v} \in H(\text{Curl}) \,\middle|\, \forall K \in \mathcal{T}_h,\ \mathbf{v}_{|K}(x,y,z) = \left| \begin{smallmatrix} \alpha_K^1 \\ \alpha_K^2 \\ \alpha_K^3 \end{smallmatrix} \right. + \left| \begin{smallmatrix} \beta_K^1 \\ \beta_K^2 \\ \beta_K^3 \end{smallmatrix} \right. \times \left| \begin{smallmatrix} x \\ y \\ z \end{smallmatrix} \right. \right\} : label : eq : Edge03d$$

  where by writing curl$\mathbf{w} = \left| \begin{smallmatrix} \partial w_2/\partial x_3 - \partial w_3/\partial x_2 \\ \partial w_3/\partial x_1 - \partial w_1/\partial x_3 \\ \partial w_1/\partial x_2 - \partial w_2/\partial x_1 \end{smallmatrix} \right.$ with $\mathbf{w} = (w_i)_{i=1}^d$:

$$H(\text{curl}) = \left\{ \mathbf{w} \in L^2(\Omega)^d \,\middle|\, \text{curl } \mathbf{w} \in L^2(\Omega)^d \right\}$$

  and $\alpha_K^1, \alpha_K^2, \alpha_K^3, \beta_K^1, \beta_K^2, \beta_K^3$ are real numbers.

- `[Edge13d]` (needs `load "Element_Mixte3d"`) 3d Nedelec finite element or Edge Element of degree 1.
- `[Edge23d]` (needs `load "Element_Mixte3d"`) 3d Nedelec finite element or Edge Element of degree 2.
- `[P1nc]` piecewise linear element continuous at the mid-point of the edge only in 2D (Crouzeix-Raviart Finite Element 2D).
- `[P2pnc]` piecewise quadratic plus a P3 bubble element with the continuity of the 2 moments on each edge (needs `load "Element_P2pnc"`)
- `[RT1]` (needs `load "Element_Mixte"`)

$$RT1_h = \left\{ \mathbf{v} \in H(\text{div}) \,\middle|\, \forall K \in \mathcal{T}_h,\ \alpha_K^1, \alpha_K^2, \beta_K \in P_1^2, P_0, \mathbf{v}_{|K}(x,y) = \left|\begin{smallmatrix} \alpha_K^1 \\ \alpha_K^2 \end{smallmatrix} + \beta_K \,\middle|\, \begin{smallmatrix} x \\ y \end{smallmatrix}\right. \right\} \tag{3.10}$$

- `[RT1Ortho]` (needs `load "Element_Mixte"`)

$$RT1_h = \left\{ \mathbf{v} \in H(\text{curl}) \,\middle|\, \forall K \in \mathcal{T}_h,\ \alpha_K^1, \alpha_K^2, \beta_K \in P_1^2, P_0, \mathbf{v}_{|K}(x,y) = \left|\begin{smallmatrix} \alpha_K^1 \\ \alpha_K^2 \end{smallmatrix} + \beta_K \,\middle|\, \begin{smallmatrix} -y \\ x \end{smallmatrix}\right. \right\} \tag{3.11}$$

- `[RT2]` (needs `load "Element_Mixte"`)

$$RT2_h = \left\{ \mathbf{v} \in H(\text{div}) \,\middle|\, \forall K \in \mathcal{T}_h,\ \alpha_K^1, \alpha_K^2, \beta_K \in P_2^2, P_1, \mathbf{v}_{|K}(x,y) = \left|\begin{smallmatrix} \alpha_K^1 \\ \alpha_K^2 \end{smallmatrix} + \beta_K \,\middle|\, \begin{smallmatrix} x \\ y \end{smallmatrix}\right. \right\} \tag{3.12}$$

- `[RT2Ortho]` (needs `load "Element_Mixte"`)

$$RT2_h = \left\{ \mathbf{v} \in H(\text{curl}) \,\middle|\, \forall K \in \mathcal{T}_h,\ \alpha_K^1, \alpha_K^2, \beta_K \in P_2^2, P_1,\ \mathbf{v}_{|K}(x,y) = \left|\begin{smallmatrix} \alpha_K^1 \\ \alpha_K^2 \end{smallmatrix} + \beta_K \,\middle|\, \begin{smallmatrix} -y \\ x \end{smallmatrix}\right. \right\} \tag{3.13}$$

- `[BDM1]` (needs `load "Element_Mixte"`) the Brezzi-Douglas-Marini finite element:

$$BDM1_h = \left\{ \mathbf{v} \in H(\text{div}) \,\middle|\, \forall K \in \mathcal{T}_h,\ \mathbf{v}_{|K} \in P_1^2 \right\} \tag{3.14}$$

- `[BDM1Ortho]` (needs `load "Element_Mixte"`) the Brezzi-Douglas-Marini Orthogonal also call Nedelec of type II , finite element

$$BDM1Ortho_h = \left\{ \mathbf{v} \in H(\text{curl}) \,\middle|\, \forall K \in \mathcal{T}_h,\ \mathbf{v}_{|K} \in P_1^2 \right\} \tag{3.15}$$

- `[FEQF]` (needs `load "Element_QF"`) the finite element to store functions at default quadrature points (so the quadrature is `qf5pT` in 2D and is `qfV5` in 3d).

  For over quadrature you have the following corresponding finite element's quadrature formula.

  - `FEQF1` $\mapsto$ `qf1pT`,
  - `FEQF2` $\mapsto$ `qf2pT`,
  - `FEQF5` $\mapsto$ `qf5pT`,

- FEQF7 $\mapsto$ qf7pT,
- FEQF9 $\mapsto$ qf9pT,
- FEQF13d $\mapsto$ qfV1,
- FEQF23d $\mapsto$ qfV2,
- FEQF53d $\mapsto$ qfV5

You can use this element to optimize the storage and reuse of functions with a long formula inside an integral for non linear processes.

3.3.1 Use of freefem fespace in 2D

With the 2D finite element spaces

$$X_h = \left\{ v \in H^1(]0,1[^2) |\ \forall K \in \mathcal{T}_h \quad v_{|K} \in P_1 \right\}$$

$$X_{ph} = \{ v \in X_h |\ v\left(|\,{}^{0}_{\cdot}\,\right) = v\left(|\,{}^{1}_{\cdot}\,\right), v\left(|\,\dot{\mathrm{o}}\,\right) = v\left(|\,\mathrm{i}\,\right) \}$$

$$M_h = \left\{ v \in H^1(]0,1[^2) |\ \forall K \in \mathcal{T}_h \quad v_{|K} \in P_2 \right\}$$

$$R_h = \left\{ \mathbf{v} \in H^1(]0,1[^2)^2 |\ \forall K \in \mathcal{T}_h \quad \mathbf{v}_{|K}(x,y) = \left|\,{}^{\alpha_K}_{\beta_K}\right. + \gamma_K \left|\,{}^{x}_{y}\right. \right\}$$

when $\mathcal{T}_h$ is a mesh 10×10 of the unit square $]0,1[^2$, we only write in **FreeFEM**:

```
mesh Th = square(10, 10);
fespace Xh(Th, P1); //scalar FE
fespace Xph(Th,P1,
    periodic=[[2, y], [4, y], [1, x], [3, x]]); //bi-periodic FE
fespace Mh(Th, P2); //scalar FE
fespace Rh(Th, RT0); //vectorial FE
```

where `Xh, Mh, Rh` expresses finite element spaces (called FE spaces) X_h, M_h, R_h, respectively.

To use FE-functions $u_h, v_h \in X_h$, $p_h, q_h \in M_h$ and $U_h, V_h \in R_h$, we write:

```
Xh uh, vh;
Xph uph, vph;
Mh ph, qh;
Rh [Uxh, Uyh], [Vxh, Vyh];
Xh[int] Uh(10);          //array of 10 functions in Xh
Rh[int] [Wxh, Wyh](10); //array of 10 functions in Rh
Wxh[5](0.5,0.5);         //the 6th function at point (0.5, 0.5)
Wxh[5][];                //the array of the degree of freedom of the 6th function
```

The functions U_h, V_h have two components so we have

$$U_h = \left|\,{}^{Uxh}_{Uyh}\right. \quad \text{and} \quad V_h = \left|\,{}^{Vxh}_{Vyh}\right.$$

3.3.2 Use of fespace in 3D

With the 3D finite element spaces

$$X_h = \{ v \in H^1(]0,1[^3) |\ \forall K \in \mathcal{T}_h \quad v_{|K} \in P_1 \}$$

$$X_{ph} = \{ v \in X_h |\ v\left(|\,{}^{0}_{\cdot}\,\right) = v\left(|\,{}^{1}_{\cdot}\,\right), v\left(|\,\dot{\mathrm{o}}\,\right) = v\left(|\,\mathrm{i}\,\right) \}$$

$$M_h = \{v \in H^1(]0,1[^2) |\ \forall K \in \mathcal{T}_h \quad v_{|K} \in P_2\}$$

$$R_h = \left\{\mathbf{v} \in H^1(]0,1[^2)^2 |\ \forall K \in \mathcal{T}_h \quad \mathbf{v}_{|K}(x,y) = \left|\begin{smallmatrix}\alpha_K \\ \beta_K\end{smallmatrix}\right. + \gamma_K \left|\begin{smallmatrix}x \\ y\end{smallmatrix}\right. \right\}$$

when $\mathcal{T}_h$ is a mesh $10 \times 10 \times 10$ of the unit cubic $]0,1[^2$, we write in **FreeFEM**:

```
mesh3 Th = buildlayers(square(10, 10),10, zbound=[0,1]);
//label: 0 up, 1 down, 2 front, 3 left, 4 back, 5 right
fespace Xh(Th, P1); //scalar FE
fespace Xph(Th, P1,
    periodic=[[0, x, y], [1, x, y],
        [2, x, z], [4, x, z],
        [3, y, z], [5, y, z]]); //three-periodic FE
fespace Mh(Th, P2); //scalar FE
fespace Rh(Th, RT03d); //vectorial FE
```

where `Xh, Mh, Rh` expresses finite element spaces (called FE spaces) X_h, M_h, R_h, respectively.

To define and use FE-functions $u_h, v_h \in X_h, p_h, q_h \in M_h$ and $U_h, V_h \in R_h$, we write:

```
Xh uh, vh;
Xph uph, vph;
Mh ph, qh;
Rh [Uxh, Uyh, Uyzh], [Vxh, Vyh, Vyzh];
Xh[int] Uh(10);              //array of 10 functions in Xh
Rh[int] [Wxh,Wyh,Wzh](10);   // array of 10 functions in Rh
Wxh[5](0.5,0.5,0.5);         //the 6th function at point (0.5, 0.5, 0.5)
Wxh[5][];                    //the array of the degree of freedom of the 6th function
```

The functions U_h, V_h have three components, so we have:

$$U_h = \left|\begin{smallmatrix}Uxh \\ Uyh \\ Uzh\end{smallmatrix}\right. \quad \text{and} \quad V_h = \left|\begin{smallmatrix}Vxh \\ Vyh \\ Vzh\end{smallmatrix}\right.$$

Note: One challenge of the periodic boundary condition is that the mesh must have equivalent faces.

The `buildlayers` mesh generator splits each quadrilateral face with the diagonal passing through the vertex with maximum number, so to be sure to have the same mesh one both face periodic the 2D numbering in corresponding edges must be compatible (for example the same variation).

By Default, the numbering of square vertex is correct.

To change the mesh numbering you can use the `change` function like:

```
{
    int[int] old2new(0:Th.nv-1); //array set on 0, 1, .., nv-1
    fespace Vh2(Th, P1);
    Vh2 sorder = x+y; //choose an order increasing on 4 square borders with x or y
    sort(sorder[], old2new); //build the inverse permutation
    int[int] new2old = old2new^-1; //inverse the permutation
    Th = change(Th, renumv=new2old);
}
```

The full example is in *examples*.

3.3.3 Lagrangian Finite Elements

P0-element

For each triangle (d=2) or tetrahedron (d=3) T_k, the basis function ϕ_k in `Vh(Th, P0)` is given by:

$$\phi_k(\mathbf{x}) = \begin{cases} 1 & \text{if } (\mathbf{x}) \in T_k \\ 0 & \text{if } (\mathbf{x}) \notin T_k \end{cases}$$

If we write:

```
Vh(Th, P0);
Vh fh = f(x,y);
```

then for vertices q^{k_i}, $i = 1, 2, ..d+1$ in Fig. 3.30, f_h is built as `fh`= $f_h(x,y) = \sum_k f(\frac{\sum_i q^{k_i}}{d+1})\phi_k$

See Fig. 3.31b for the projection of $f(x,y) = \sin(\pi x)\cos(\pi y)$ on `Vh(Th, P0)` when the mesh `Th` is a 4×4-grid of $[-1,1]^2$ as in Fig. 3.31a.

P1-element

Fig. 3.30: P_1 and P_2 degrees of freedom on triangle T_k

For each vertex q^i, the basis function ϕ_i in `Vh(Th, P1)` is given by:

$$\phi_i(x,y) = a_i^k + b_i^k x + c_i^k y \text{ for } (x,y) \in T_k,$$
$$\phi_i(q^i) = 1, \quad \phi_i(q^j) = 0 \text{ if } i \neq j$$

The basis function $\phi_{k_1}(x,y)$ with the vertex q^{k_1} in Fig. 3.30 at point $p = (x,y)$ in triangle T_k simply coincide with the *barycentric coordinates* λ_1^k *(area coordinates)*:

$$\phi_{k_1}(x,y) = \lambda_1^k(x,y) = \frac{\text{area of triangle}(p, q^{k_2}, q^{k_3})}{\text{area of triangle}(q^{k_1}, q^{k_2}, q^{k_3})}$$

If we write:

(a) Test mesh `Th` for projection

(b) Projection to `Vh(Th, P0)`

Fig. 3.31: Finite element `P0`

```
Vh(Th, P1);
Vh fh = g(x.y);
```

then:

$$\texttt{fh} = f_h(x,y) = \sum_{i=1}^{n_v} f(q^i)\phi_i(x,y)$$

See Fig. 3.32a for the projection of $f(x,y) = \sin(\pi x)\cos(\pi y)$ into `Vh(Th, P1)`.

P2-element

For each vertex or mid-point q^i. The basis function ϕ_i in `Vh(Th, P2)` is given by:

$$\begin{array}{rcl} \phi_i(x,y) & = & a_i^k + b_i^k x + c_i^k y + d_i^k x^2 + e_i^k xy + f_j^f y^2 \text{ for } (x,y) \in T_k, \\ \phi_i(q^i) & = & 1, \quad \phi_i(q^j) = 0 \text{ if } i \neq j \end{array}$$

The basis function $\phi_{k_1}(x,y)$ with the vertex q^{k_1} in Fig. 3.30 is defined by the *barycentric coordinates*:

$$\phi_{k_1}(x,y) = \lambda_1^k(x,y)(2\lambda_1^k(x,y) - 1)$$

and for the mid-point q^{k_2}:

$$\phi_{k_2}(x,y) = 4\lambda_1^k(x,y)\lambda_4^k(x,y)$$

If we write:

```
Vh(Th, P2);
Vh fh = f(x.y);
```

then:

$$\texttt{fh} = f_h(x,y) = \sum_{i=1}^{M} f(q^i)\phi_i(x,y) \quad \text{(summation over all vertex or mid-point)}$$

See *Projection to Vh(Th, P2)* for the projection of $f(x,y) = \sin(\pi x)\cos(\pi y)$ into `Vh(Th, P2)`.

(a) Projection to `Vh(Th, P1)`

(b) Projection to `Vh(Th, P2)`

Fig. 3.32: Finite elements `P1, P2`

3.3.4 P1 Nonconforming Element

Refer to [THOMASSET2012] for details; briefly, we now consider non-continuous approximations so we will lose the property:

$$w_h \in V_h \subset H^1(\Omega)$$

If we write:

```
Vh(Th, P1nc);
Vh fh = f(x.y);
```

then:

$$\texttt{fh} = f_h(x,y) = \sum_{i=1}^{n_v} f(m^i)\phi_i(x,y) \quad \text{(summation over all midpoint)}$$

Here the basis function ϕ_i associated with the mid-point $m^i = (q^{k_i} + q^{k_{i+1}})/2$ where q^{k_i} is the i-th point in T_k, and we assume that $j+1 = 0$ if $j = 3$:

$$\phi_i(x,y) = a_i^k + b_i^k x + c_i^k y \text{ for } (x,y) \in T_k,$$
$$\phi_i(m^i) = 1, \quad \phi_i(m^j) = 0 \text{ if } i \neq j$$

Strictly speaking $\partial\phi_i/\partial x$, $\partial\phi_i/\partial y$ contain Dirac distribution $\rho\delta_{\partial T_k}$.

The numerical calculations will automatically *ignore* them. In [THOMASSET2012], there is a proof of the estimation

$$\left(\sum_{k=1}^{n_v} \int_{T_k} |\nabla w - \nabla w_h|^2 textdxtextdy\right)^{1/2} = O(h)$$

The basis functions ϕ_k have the following properties.

1. For the bilinear form a defined in Fig. 3.33a satisfy:

$$\begin{array}{rrcl} a(\phi_i,\phi_i) > 0, & a(\phi_i,\phi_j) & \leq & 0 \quad \text{if } i \neq j \\ & \sum_{k=1}^{n_v} a(\phi_i,\phi_k) & \geq & 0 \end{array}$$

(a) Projection to `Vh(Th, P1nc)`

(b) Projection to `Vh(Th, P1b)`

Fig. 3.33: Finite elements `P1nc, P1b`

2. $f \geq 0 \Rightarrow u_h \geq 0$
3. If $i \neq j$, the basis function ϕ_i and ϕ_j are L^2-orthogonal:

$$\int_\Omega \phi_i \phi_j \, textdxtextdy = 0 \qquad \text{if } i \neq j$$

which is false for P_1-element.

See Fig. 3.33a for the projection of $f(x,y) = \sin(\pi x)\cos(\pi y)$ into `Vh(Th, P1nc)`.

3.3.5 Other FE-space

For each triangle $T_k \in \mathcal{T}_h$, let $\lambda_{k_1}(x,y)$, $\lambda_{k_2}(x,y)$, $\lambda_{k_3}(x,y)$ be the area cordinate of the triangle (see Fig. 3.30), and put:

$$\beta_k(x,y) = 27\lambda_{k_1}(x,y)\lambda_{k_2}(x,y)\lambda_{k_3}(x,y)$$

called *bubble* function on T_k. The bubble function has the feature: 1. $\beta_k(x,y) = 0 \quad \text{if } (x,y) \in \partial T_k$.

2. $\beta_k(q^{k_b}) = 1$ where q^{k_b} is the barycenter $\frac{q^{k_1}+q^{k_2}+q^{k_3}}{3}$.

If we write:

```
Vh(Th, P1b);
Vh fh = f(x.y);
```

then:

$$\texttt{fh} = f_h(x,y) = \sum_{i=1}^{n_v} f(q^i)\phi_i(x,y) + \sum_{k=1}^{n_t} f(q^{k_b})\beta_k(x,y)$$

See Fig. 3.33b for the projection of $f(x, y) = \sin(\pi x)\cos(\pi y)$ into `Vh(Th, P1b)`.

3.3.6 Vector Valued FE-function

Functions from $\mathbb{R}^2$ to $\mathbb{R}^N$ with $N = 1$ are called scalar functions and called *vector valued* when $N > 1$. When $N = 2$

```
fespace Vh(Th, [P0, P1]) ;
```

makes the space

$$V_h = \{\mathbf{w} = (w_1, w_2)|\ w_1 \in V_h(\mathcal{T}_h, P_0),\ w_2 \in V_h(\mathcal{T}_h, P_1)\}$$

Raviart-Thomas Element

In the Raviart-Thomas finite element $RT0_h$, the degrees of freedom are the fluxes across edges e of the mesh, where the flux of the function $\mathbf{f} : \mathbb{R}^2 \longrightarrow \mathbb{R}^2$ is $\int_e \mathbf{f}.n_e$, n_e is the unit normal of edge e.

This implies an orientation of all the edges of the mesh, for example we can use the global numbering of the edge vertices and we just go from small to large numbers.

To compute the flux, we use a quadrature with one Gauss point, the mid-point of the edge.

Consider a triangle T_k with three vertices $(\mathbf{a}, \mathbf{b}, \mathbf{c})$.

Lets denote the vertices numbers by i_a, i_b, i_c, and define the three edge vectors $\mathbf{e}^1, \mathbf{e}^2, \mathbf{e}^3$ by $sgn(i_b - i_c)(\mathbf{b} - \mathbf{c})$, $sgn(i_c - i_a)(\mathbf{c} - \mathbf{a})$, $sgn(i_a - i_b)(\mathbf{a} - \mathbf{b})$.

We get three basis functions:

$$\phi_1^k = \frac{sgn(i_b - i_c)}{2|T_k|}(\mathbf{x} - \mathbf{a}), \quad \phi_2^k = \frac{sgn(i_c - i_a)}{2|T_k|}(\mathbf{x} - \mathbf{b}), \quad \phi_3^k = \frac{sgn(i_a - i_b)}{2|T_k|}(\mathbf{x} - \mathbf{c}),$$

where $|T_k|$ is the area of the triangle T_k. If we write:

```
Vh(Th, RT0);
Vh [f1h, f2h] = [f1(x, y), f2(x, y)];
```

then:

$$\texttt{fh} = \mathbf{f}_h(x, y) = \sum_{k=1}^{n_t}\sum_{l=1}^{6} n_{i_l j_l}|\mathbf{e}^{\mathbf{i_1}}|f_{j_l}(m^{i_l})\phi_{i_l j_l}$$

where $n_{i_l j_l}$ is the j_l-th component of the normal vector $\mathbf{n}_{i_l}$,

$$\{m_1, m_2, m_3\} = \left\{\frac{\mathbf{b}+\mathbf{c}}{2}, \frac{\mathbf{a}+\mathbf{c}}{2}, \frac{\mathbf{b}+\mathbf{a}}{2}\right\}$$

and $i_l = \{1, 1, 2, 2, 3, 3\}$, $j_l = \{1, 2, 1, 2, 1, 2\}$ with the order of l.

```
// Mesh
mesh Th = square(2, 2);

// Fespace
fespace Xh(Th, P1);
Xh uh = x^2 + y^2, vh;

fespace Vh(Th, RT0);
```

(continues on next page)

Fig. 3.34: Normal vectors of each edge

(continued from previous page)

```
Vh [Uxh, Uyh] = [sin(x), cos(y)]; //vectorial FE function

// Change the mesh
Th = square(5,5);

//Xh is unchanged
//Uxh = x; //error: impossible to set only 1 component
          //of a vector FE function
vh = Uxh;//ok
//and now vh use the 5x5 mesh
//but the fespace of vh is always the 2x2 mesh

// Plot
plot(uh);
uh = uh; //do a interpolation of uh (old) of 5x5 mesh
        //to get the new uh on 10x10 mesh
plot(uh);

vh([x-1/2, y]) = x^2 + y^2; //interpolate vh = ((x-1/2)^2 + y^2)
```

To get the value at a point $x = 1, y = 2$ of the FE function `uh`, or `[Uxh, Uyh]`, one writes:

```
real value;
value = uh(2,4); //get value = uh(2, 4)
value = Uxh(2, 4); //get value = Uxh(2, 4)
//OR
x = 1; y = 2;
value = uh; //get value = uh(1, 2)
value = Uxh; //get value = Uxh(1, 2)
value = Uyh; //get value = Uyh(1, 2)
```

(a) vh Iso on mesh 2×2

(b) vh Iso on mesh 5×5

To get the value of the array associated to the FE function uh, one writes

```
real value = uh[][0]; //get the value of degree of freedom 0
real maxdf = uh[].max; //maximum value of degree of freedom
int size = uh.n; //the number of degree of freedom
real[int] array(uh.n) = uh[]; //copy the array of the function uh
```

Warning: For a non-scalar finite element function `[Uxh, Uyh]` the two arrays `Uxh[]` and `Uyh[]` are the same array, because the degree of freedom can touch more than one component.

3.3.7 A Fast Finite Element Interpolator

In practice, one may discretize the variational equations by the Finite Element method. Then there will be one mesh for Ω_1 and another one for Ω_2. The computation of integrals of products of functions defined on different meshes is difficult.

Quadrature formula and interpolations from one mesh to another at quadrature points are needed. We present below the interpolation operator which we have used and which is new, to the best of our knowledge.

Let $\mathcal{T}_h^0 = \cup_k T_k^0, \mathcal{T}_h^1 = \cup_k T_k^1$ be two triangulations of a domain Ω. Let:

$$V(\mathrm{T}_h^i) = \{C^0(\Omega_h^i) \ : \ f|_{T_k^i} \in P_0\}, \quad i = 0, 1$$

be the spaces of continuous piecewise affine functions on each triangulation.

Let $f \in V(\mathcal{T}_h^0)$. The problem is to find $g \in V(\mathcal{T}_h^1)$ such that:

$$g(q) = f(q) \quad \forall q \text{ vertex of } \mathcal{T}_h^1$$

Although this is a seemingly simple problem, it is difficult to find an efficient algorithm in practice.

We propose an algorithm which is of complexity $N^1 \log N^0$, where N^i is the number of vertices of $\mathcal{T}_{\langle}^{\rangle}$, and which is very fast for most practical 2D applications.

Algorithm

The method has 5 steps.

First a quadtree is built containing all the vertices of the mesh $\mathcal{T}_h^0$ such that in each terminal cell there are at least one, and at most 4, vertices of $\mathcal{T}_h^0$.

For each q^1, vertex of $\mathcal{T}_h^1$ do:

1. Find the terminal cell of the quadtree containing q^1.
2. Find the the nearest vertex q_j^0 to q^1 in that cell.
3. Choose one triangle $T_k^0 \in \mathcal{T}_h^0$ which has q_j^0 for vertex.
4. Compute the barycentric coordinates $\{\lambda_j\}_{j=1,2,3}$ of q^1 in T_k^0.
   - if all barycentric coordinates are positive, go to Step 5
   - otherwise, if one barycentric coordinate λ_i is negative, replace T_k^0 by the adjacent triangle opposite q_i^0 and go to Step 4.
   - otherwise, if two barycentric coordinates are negative, take one of the two randomly and replace T_k^0 by the adjacent triangle as above.
5. Calculate $g(q^1)$ on T_k^0 by linear interpolation of f:

$$g(q^1) = \sum_{j=1,2,3} \lambda_j f(q_j^0)$$

Fig. 3.36: To interpolate a function at q^0, the knowledge of the triangle which contains q^0 is needed. The algorithm may start at $q^1 \in T_k^0$ and stall on the boundary (thick line) because the line q^0q^1 is not inside Ω. But if the holes are triangulated too (doted line) then the problem does not arise.

Two problems need to be solved:

- *What if :math:'q^1' is not in Ω_h^0 ?* Then Step 5 will stop with a boundary triangle.

  So we add a step which tests the distance of q^1 with the two adjacent boundary edges and selects the nearest, and so on till the distance grows.

- *What if Ω_h^0 is not convex and the marching process of Step 4 locks on a boundary?* By construction Delaunay-Voronoï's mesh generators always triangulate the convex hull of the vertices of the domain.

Therefore, we make sure that this information is not lost when $\mathcal{T}_h^0, \mathcal{T}_h^1$ are constructed and we keep the triangles which are outside the domain on a special list.

That way, in step 5 we can use that list to step over holes if needed.

Note: Sometimes, in rare cases, the interpolation process misses some points, we can change the search algorithm through a global variable `searchMethod`

```
searchMethod = 0; // default value for fast search algorithm
searchMethod = 1; // safe search algorithm, uses brute force in case of missing point
// (warning: can be very expensive in cases where a lot of points are outside of the domain)
searchMethod = 2; // always uses brute force. It is very computationally expensive.
```

Note: Step 3 requires an array of pointers such that each vertex points to one triangle of the triangulation.

Note: The operator = is the interpolation operator of **FreeFEM**, the continuous finite functions are extended by continuity to the outside of the domain.

Try the following example :

```
// Mesh
mesh Ths = square(10, 10);
mesh Thg = square(30, 30, [x*3-1, y*3-1]);
plot(Ths, Thg, wait=true);

// Fespace
fespace Ch(Ths, P2);
Ch us = (x-0.5)*(y-0.5);

fespace Dh(Ths, P2dc);
Dh vs = (x-0.5)*(y-0.5);

fespace Fh(Thg, P2dc);
Fh ug=us, vg=vs;

// Plot
plot(us, ug, wait=true);
plot(vs, vg, wait=true);
```

3.3.8 Keywords: Problem and Solve

For **FreeFEM**, a problem must be given in variational form, so we need a bilinear form $a(u, v)$, a linear form $\ell(f, v)$, and possibly a boundary condition form must be added.

```
problem P (u, v)
    = a(u,v) - l(f,v)
    + (boundary condition)
    ;
```

(a) Extension of a continuous FE-function

(b) Extension of discontinuous FE-function

Fig. 3.37: Extension of FE-function

Note: When you want to formulate the problem and solve it in the same time, you can use the keyword `solve`.

Weak Form and Boundary Condition

To present the principles of Variational Formulations, also called weak form, for the Partial Differential Equations, let's take a model problem: a Poisson equation with Dirichlet and Robin Boundary condition.

The problem: Find u a real function defined on a domain Ω of $\mathbb{R}^d$ $(d = 2, 3)$ such that:

$$\begin{array}{rcll} -\nabla \cdot (\kappa \nabla u) & = & f & \text{in } \Omega \\ au + \kappa \frac{\partial u}{\partial n} & = & b & \text{on } \Gamma_r \\ u & = & g & \text{on } \Gamma_d \end{array}$$

where:

- if $d = 2$ then $\nabla.(\kappa \nabla u) = \partial_x(\kappa \partial_x u) + \partial_y(\kappa \partial_y u)$ with $\partial_x u = \frac{\partial u}{\partial x}$ and $\partial_y u = \frac{\partial u}{\partial y}$
- if $d = 3$ then $\nabla.(\kappa \nabla u) = \partial_x(\kappa \partial_x u) + \partial_y(\kappa \partial_y u) + \partial_z(\kappa \partial_z u)$ with $\partial_x u = \frac{\partial u}{\partial x}$, $\partial_y u = \frac{\partial u}{\partial y}$ and , $\partial_z u = \frac{\partial u}{\partial z}$
- The border $\Gamma = \partial\Omega$ is split in Γ_d and Γ_n such that $\Gamma_d \cap \Gamma_n = \emptyset$ and $\Gamma_d \cup \Gamma_n = \partial\Omega$,
- κ is a given positive function, such that $\exists \kappa_0 \in \mathbb{R}, \quad 0 < \kappa_0 \leq \kappa$.
- a a given non negative function,
- b a given function.

Note: This is the well known Neumann boundary condition if $a = 0$, and if Γ_d is empty.

In this case the function appears in the problem just by its derivatives, so it is defined only up to a constant (if u is a solution then $u + c$ is also a solution).

Let v, a regular test function, null on Γ_d, by integration by parts we get:

$$-\int_\Omega \nabla \cdot (\kappa \nabla u)\, v\, d\omega = \int_\Omega \kappa \nabla v \cdot \nabla u\, d\omega - \int_\Gamma v \kappa \frac{\partial u}{\partial \mathbf{n}}\, d\gamma, = \int_\Omega f v\, d\omega$$

where if $d = 2$ the $\nabla v.\nabla u = (\frac{\partial u}{\partial x}\frac{\partial v}{\partial x} + \frac{\partial u}{\partial y}\frac{\partial v}{\partial y})$,

where if $d = 3$ the $\nabla v.\nabla u = (\frac{\partial u}{\partial x}\frac{\partial v}{\partial x} + \frac{\partial u}{\partial y}\frac{\partial v}{\partial y} + \frac{\partial u}{\partial z}\frac{\partial v}{\partial z})$,

and where $\mathbf{n}$ is the unitary outer-pointing normal of the Γ.

Now we note that $\kappa\frac{\partial u}{\partial n} = -au + b$ on Γ_r and $v = 0$ on Γ_d and $\Gamma = \Gamma_d \cup \Gamma_n$ thus:

$$-\int_\Gamma v\kappa\frac{\partial u}{\partial n} = \int_{\Gamma_r} auv - \int_{\Gamma_r} bv$$

The problem becomes:

Find $u \in V_g = \{w \in H^1(\Omega)/w = g \text{ on } \Gamma_d\}$ such that:

$$\int_\Omega \kappa\nabla v.\nabla u\, d\omega + \int_{\Gamma_r} auv\, d\gamma = \int_\Omega fv\, d\omega + \int_{\Gamma_r} bv\, d\gamma, \quad \forall v \in V_0 \tag{3.16}$$

where $V_0 = \{v \in H^1(\Omega)/v = 0 \text{ on } \Gamma_d\}$

Except in the case of Neumann conditions everywhere, the problem (3.16) is well posed when $\kappa \geq \kappa_0 > 0$.

Note: If we have only the Neumann boundary condition, linear algebra tells us that the right hand side must be orthogonal to the kernel of the operator for the solution to exist.

One way of writing the compatibility condition is:

$$\int_\Omega f\, d\omega + \int_\Gamma b\, d\gamma = 0$$

and a way to fix the constant is to solve for $u \in H^1(\Omega)$ such that:

$$\int_\Omega (\varepsilon uv \ + \ \kappa\nabla v.\nabla u)\, d\omega = \int_\Omega fv\, d\omega + \int_{\Gamma_r} bv\, d\gamma, \quad \forall v \in H^1(\Omega)$$

where ε is a small parameter ($\sim \kappa\ 10^{-10}|\Omega|^{\frac{2}{d}}$).

Remark that if the solution is of order $\frac{1}{\varepsilon}$ then the compatibility condition is unsatisfied, otherwise we get the solution such that $\int_\Omega u = 0$, you can also add a Lagrange multiplier to solve the real mathematical problem like in the *Lagrange multipliers example*.

In **FreeFEM**, the bidimensional problem (3.16) becomes:

```
problem Pw (u, v)
    = int2d(Th)( //int_{Omega} kappa nabla v . nabla u
        kappa*(dx(u)*dx(v) + dy(u)*dy(v))
    )
    + int1d(Th, gn)( //int_{Gamma_r} a u v
        a * u*v
    )
    - int2d(Th)( //int_{Omega} f v
        f*v
    )
    - int1d(Th, gn)( //int_{Gamma_r} b v
        b * v
    )
    + on(gd, u=g) //u = g on Gamma_d
    ;
```

where `Th` is a mesh of the bi-dimensional domain Ω, and `gd` and `gn` are respectively the boundary labels of boundary Γ_d and Γ_n.

And the three dimensional problem (3.16) becomes

```
macro Grad(u) [dx(u), dy(u), dz(u) ]//
problem Pw (u, v)
    = int3d(Th)( //int_{Omega} kappa nabla v . nabla u
        kappa*(Grad(u)'*Grad(v))
    )
    + int2d(Th, gn)( //int_{Gamma_r} a u v
        a * u*v
    )
    - int3d(Th)( //int_{Omega} f v
        f*v
    )
    - int2d(Th, gn)( //int_{Gamma_r} b v
        b * v
    )
    + on(gd, u=g) //u = g on Gamma_d
    ;
```

where `Th` is a mesh of the three dimensional domain Ω, and `gd` and `gn` are respectively the boundary labels of boundary Γ_d and Γ_n.

3.3.9 Parameters affecting solve and problem

The parameters are FE functions real or complex, the number n of parameters is even ($n = 2 * k$), the k first function parameters are unknown, and the k last are test functions.

Note: If the functions are a part of vectorial FE then you must give all the functions of the vectorial FE in the same order (see *Poisson problem with mixed finite element* for example).

Note: Don't mix complex and real parameters FE function.

Warning: Bug:

The mixing of multiple `fespace` with different periodic boundary conditions are not implemented.

So all the finite element spaces used for tests or unknown functions in a problem, must have the same type of periodic boundary conditions or no periodic boundary conditions.

No clean message is given and the result is unpredictable.

The parameters are:

- **solver=** `LU`, `CG`, `Crout`, `Cholesky`, `GMRES`, `sparsesolver`, `UMFPACK` ...

  The default solver is `sparsesolver` (it is equal to `UMFPACK` if no other sparse solver is defined) or is set to `LU` if no direct sparse solver is available.

  The storage mode of the matrix of the underlying linear system depends on the type of solver chosen; for `LU` the matrix is sky-line non symmetric, for `Crout` the matrix is sky-line symmetric, for `Cholesky` the matrix is sky-line symmetric positive definite, for `CG` the matrix is sparse symmetric positive, and for `GMRES`, `sparsesolver` or `UMFPACK` the matrix is just sparse.

- **eps=** a real expression.

ε sets the stopping test for the iterative methods like `CG`.

Note that if ε is negative then the stopping test is:

$$||Ax - b|| < |\varepsilon|$$

if it is positive, then the stopping test is:

$$||Ax - b|| < \frac{|\varepsilon|}{||Ax_0 - b||}$$

- **init=** boolean expression, if it is false or 0 the matrix is reconstructed.

  Note that if the mesh changes the matrix is reconstructed too.

- **precon=** name of a function (for example `P`) to set the preconditioner.

  The prototype for the function `P` must be:

```
func real[int] P(real[int] & xx);
```

- **tgv=** Huge value (10^{30}) used to implement Dirichlet boundary conditions.
- **tolpivot=** sets the tolerance of the pivot in `UMFPACK` (10^{-1}) and, `LU`, `Crout`, `Cholesky` factorisation (10^{-20}).
- **tolpivotsym=** sets the tolerance of the pivot sym in `UMFPACK`
- **strategy=** sets the integer `UMFPACK` strategy (0 by default).

3.3.10 Problem definition

Below `v` is the unknown function and `w` is the test function.

After the "=" sign, one may find sums of:

- Identifier(s); this is the name given earlier to the variational form(s) (type `varf`) for possible reuse.

  Remark, that the name in the `varf` of the unknown test function is forgotten, we use the order in the argument list to recall names as in a `C++` function,

- The terms of the bilinear form itself: if K is a given function,
- Bilinear part for 3D meshes `Th`
  - `int3d(Th)(K*v*w)` $= \sum_{T\in \mathtt{Th}} \int_T K\, v\, w$
  - `int3d(Th, 1)(K*v*w)` $= \sum_{T\in \mathtt{Th}, T\subset\Omega_1} \int_T K\, v\, w$
  - `int3d(Th, levelset=phi)(K*v*w)` $= \sum_{T\in \mathtt{Th}} \int_{T,\phi<0} K\, v\, w$
  - `int3d(Th, l, levelset=phi)(K*v*w)` $= \sum_{T\in \mathtt{Th}, T\subset\Omega_l} \int_{T,\phi<0} K\, v\, w$
  - `int2d(Th, 2, 5)(K*v*w)` $= \sum_{T\in \mathtt{Th}} \int_{(\partial T\cup\Gamma)\cap(\Gamma_2\cup\Gamma_5)} K\, v\, w$
  - `int2d(Th, 1)(K*v*w)` $= \sum_{T\in \mathtt{Th}, T\subset\Omega_1} \int_T K\, v\, w$

- `int2d(Th, 2, 5)(K*v*w)` $= \sum_{T\in\mathtt{Th}} \int_{(\partial T\cup\Gamma)\cap(\Gamma_2\cup\Gamma_5)} K\,v\,w$
- `int2d(Th, levelset=phi)(K*v*w)` $= \sum_{T\in\mathtt{Th}} \int_{T,\phi=0} K\,v\,w$
- `int2d(Th, l, levelset=phi)(K*v*w)` $= \sum_{T\in\mathtt{Th},T\subset\Omega_l} \int_{T,\phi=0} K\,v\,w$
- `intallfaces(Th)(K*v*w)` $= \sum_{T\in\mathtt{Th}} \int_{\partial T} K\,v\,w$
- `intallfaces(Th, 1)(K*v*w)` $= \sum_{T\in\mathtt{Th},T\subset\Omega_1} \int_{\partial T} K\,v\,w$
- They contribute to the sparse matrix of type `matrix` which, whether declared explicitly or not, is constructed by **FreeFEM**.

- Bilinear part for 2D meshes `Th`
  - `int2d(Th)(K*v*w)` $= \sum_{T\in\mathtt{Th}} \int_{T} K\,v\,w$
  - `int2d(Th, 1)(K*v*w)` $= \sum_{T\in\mathtt{Th},T\subset\Omega_1} \int_{T} K\,v\,w$
  - `int2d(Th, levelset=phi)(K*v*w)` $= \sum_{T\in\mathtt{Th}} \int_{T,\phi<0} K\,v\,w$
  - `int2d(Th, l, levelset=phi)(K*v*w)` $= \sum_{T\in\mathtt{Th},T\subset\Omega_l} \int_{T,\phi<0} K\,v\,w$
  - `int1d(Th, 2, 5)(K*v*w)` $= \sum_{T\in\mathtt{Th}} \int_{(\partial T\cup\Gamma)\cap(\Gamma_2\cup\Gamma_5)} K\,v\,w$
  - `int1d(Th, 1)(K*v*w)` $= \sum_{T\in\mathtt{Th},T\subset\Omega_1} \int_{T} K\,v\,w$
  - `int1d(Th, 2, 5)(K*v*w)` $= \sum_{T\in\mathtt{Th}} \int_{(\partial T\cup\Gamma)\cap(\Gamma_2\cup\Gamma_5)} K\,v\,w$
  - `int1d(Th, levelset=phi)(K*v*w)` $= \sum_{T\in\mathtt{Th}} \int_{T,\phi=0} K\,v\,w$
  - `int1d(Th, l, levelset=phi)(K*v*w)` $= \sum_{T\in\mathtt{Th},T\subset\Omega_l} \int_{T,\phi=0} K\,v\,w$
  - `intalledges(Th)(K*v*w)` $= \sum_{T\in\mathtt{Th}} \int_{\partial T} K\,v\,w$
  - `intalledges(Th, 1)(K*v*w)` $= \sum_{T\in\mathtt{Th},T\subset\Omega_1} \int_{\partial T} K\,v\,w$
  - They contribute to the sparse matrix of type `matrix` which, whether declared explicitly or not, is constructed by **FreeFEM**.

- The right hand-side of the Partial Differential Equation in 3D, the terms of the linear form: for given functions K, f:

- `int3d(Th)(K*w)` $= \sum_{T\in \mathtt{Th}} \int_T K\,w$
- `int3d(Th, l)(K*w)` $= \sum_{T\in \mathtt{Th}, T\in\Omega_l} \int_T K\,w$
- `int3d(Th, levelset=phi)(K*w)` $= \sum_{T\in \mathtt{Th}} \int_{T,\phi<0} K\,w$
- `int3d(Th, l, levelset=phi)(K*w)` $= \sum_{T\in \mathtt{Th}, T\subset\Omega_l} \int_{T,\phi<0} K\,w$
- `int2d(Th, 2, 5)(K*w)` $= \sum_{T\in \mathtt{Th}} \int_{(\partial T\cup\Gamma)\cap(\Gamma_2\cup\Gamma_5)} K\,w$
- `int2d(Th, levelset=phi)(K*w)` $= \sum_{T\in \mathtt{Th}} \int_{T,\phi=0} K\,w$
- `int2d(Th, l, levelset=phi)(K*w)` $= \sum_{T\in \mathtt{Th}, T\subset\Omega_l} \int_{T,\phi=0} K\,w$
- `intallfaces(Th)(f*w)` $= \sum_{T\in \mathtt{Th}} \int_{\partial T} f\,w$
- A vector of type `real[int]`

- The right hand-side of the Partial Differential Equation in 2D, the terms of the linear form: for given functions K, f:
  - `int2d(Th)(K*w)` $= \sum_{T\in \mathtt{Th}} \int_T K\,w$
  - `int2d(Th, l)(K*w)` $= \sum_{T\in \mathtt{Th}, T\in\Omega_l} \int_T K\,w$
  - `int2d(Th, levelset=phi)(K*w)` $= \sum_{T\in \mathtt{Th}} \int_{T,\phi<0} K\,w$
  - `int2d(Th, l, levelset=phi)(K*w)` $= \sum_{T\in \mathtt{Th}, T\subset\Omega_l} \int_{T,\phi<0} K\,w$
  - `int1d(Th, 2, 5)(K*w)` $= \sum_{T\in \mathtt{Th}} \int_{(\partial T\cup\Gamma)\cap(\Gamma_2\cup\Gamma_5)} K\,w$
  - `int1d(Th, levelset=phi)(K*w)` $= \sum_{T\in \mathtt{Th}} \int_{T,\phi=0} K\,w$
  - `int1d(Th, l, levelset=phi)(K*w)` $= \sum_{T\in \mathtt{Th}, T\subset\Omega_l} \int_{T,\phi=0} K\,w$
  - `intalledges(Th)(f*w)` $= \sum_{T\in \mathtt{Th}} \int_{\partial T} f\,w$
  - a vector of type `real[int]`

- The boundary condition terms:
  - An "on" scalar form (for Dirichlet) : `on(1, u=g)`

Used for all degrees of freedom i of the boundary referred by "1", the diagonal term of the matrix $a_{ii} = tgv$ with the *terrible giant value* `tgv` (= 10^{30} by default), and the right hand side $b[i] =$ "$(\Pi_h g)[i]$" $\times\, tgv$, where the "$(\Pi_h g)g[i]$" is the boundary node value given by the interpolation of g.

Note: if `tgv` < 0 then we put to 0 all term of the line i in the matrix, except diagonal term $a_{ii} = 1$, and $b[i] =$ "$(\Pi_h g)[i]$".

- An "on" vectorial form (for Dirichlet): `on(1, u1=g1, u2=g2)`

If you have vectorial finite element like `RT0`, the 2 components are coupled, and so you have : $b[i] =$ "$(\Pi_h(g1, g2))[i]$" $\times\, tgv$, where Π_h is the vectorial finite element interpolant.

- A linear form on Γ (for Neumann in 2d) `-int1d(Th)(f*w)` or `-int1d(Th, 3)(f*w)`
- A bilinear form on Γ or Γ_2 (for Robin in 2d) `int1d(Th)(K*v*w)` or `int1d(Th, 2)(K*v*w)`
- A linear form on Γ (for Neumann in 3d) `-int2d(Th)(f*w)` or `-int2d(Th, 3)(f*w)`
- A bilinear form on Γ or Γ_2 (for Robin in 3d) `int2d(Th)(K*v*w)` or `int2d(Th, 2)(K*v*w)`

Note:

- If needed, the different kind of terms in the sum can appear more than once.
- The integral mesh and the mesh associated to test functions or unknown functions can be different in the case of linear form.
- `N.x`, `N.y` and `N.z` are the normal's components.

Warning: It is not possible to write in the same integral the linear part and the bilinear part such as in `int1d(Th)(K*v*w - f*w)`.

3.3.11 Numerical Integration

Let D be a N-dimensional bounded domain.

For an arbitrary polynomial f of degree r, if we can find particular (quadrature) points ξ_j, $j = 1, \cdots, J$ in D and (quadrature) constants ω_j such that

$$\int_D f(\mathbf{x}) = \sum_{\ell=1}^{L} c_\ell f(\xi_\ell)$$

then we have an error estimate (see [CROUZEIX1984]), and then there exists a constant $C > 0$ such that

$$\left| \int_D f(\mathbf{x}) - \sum_{\ell=1}^{L} \omega_\ell f(\xi_\ell) \right| \leq C|D|h^{r+1}$$

for any function $r+1$ times continuously differentiable f in D, where h is the diameter of D and $|D|$ its measure (a point in the segment $[q^i q^j]$ is given as

$$\{(x, y)|\ x = (1-t)q_x^i + tq_x^j,\ y = (1-t)q_y^i + tq_y^j,\ 0 \leq t \leq 1\}$$

For a domain $\Omega_h = \sum_{k=1}^{n_t} T_k,\ \mathcal{T}_h = \{T_k\}$, we can calculate the integral over $\Gamma_h = \partial\Omega_h$ by:

$\int_{\Gamma_h} f(\mathbf{x})ds$ =`int1d(Th)(f)` =`int1d(Th, qfe=*)(f)` =`int1d(Th, qforder=*)(f)`

where * stands for the name of the quadrature formula or the precision (order) of the Gauss formula.

Quadrature formula on an edge					
L	`qfe`	`qforder`	Point in $[q^i, q^j]$	ω_ℓ	Exact on $P_k,\ k =$
1	`qf1pE`	2	$1/2$	$\|\|q^iq^j\|\|$	1
2	`qf2pE`	3	$(1 \pm \sqrt{1/3})/2$	$\|\|q^iq^j\|\|/2$	3
3	`qf3pE`	6	$(1 \pm \sqrt{3/5})/2$ $1/2$	$(5/18)\|\|q^iq^j\|\|$ $(8/18)\|\|q^iq^j\|\|$	5
4	`qf4pE`	8	$(1 \pm \frac{525+70\sqrt{30}}{35})/2$ $(1 \pm \frac{525-70\sqrt{30}}{35})/2$	$\frac{18-\sqrt{30}}{72}\|\|q^iq^j\|\|$ $\frac{18+\sqrt{30}}{72}\|\|q^iq^j\|\|$	7
5	`qf5pE`	10	$(1 \pm \frac{245+14\sqrt{70}}{21})/2$ $1/2$ $(1 \pm \frac{245-14\sqrt{70}}{21})/2$	$\frac{322-13\sqrt{70}}{1800}\|\|q^iq^j\|\|$ $\frac{64}{225}\|\|q^iq^j\|\|$ $\frac{322+13\sqrt{70}}{1800}\|\|q^iq^j\|\|$	9
2	`qf1pElump`	2	0 1	$\|\|q^iq^j\|\|/2$ $\|\|q^iq^j\|\|/2$	1

where $|q^iq^j|$ is the length of segment $\overline{q^iq^j}$.

For a part Γ_1 of Γ_h with the label "1", we can calculate the integral over Γ_1 by:

$\int_{\Gamma_1} f(x,y)ds$ =`int1d(Th, 1)(f)` =`int1d(Th, 1, qfe=qf2pE)(f)`

The integrals over $\Gamma_1,\ \Gamma_3$ are given by:

$\int_{\Gamma_1\cup\Gamma_3} f(x,y)ds$

For each triangle $T_k = [q^{k_1}q^{k_2}q^{k_3}]$, the point $P(x,y)$ in T_k is expressed by the *area coordinate* as $P(\xi,\eta)$:

$$|T_k| = \frac{1}{2}\begin{vmatrix} 1 & q_x^{k_1} & q_y^{k_1} \\ 1 & q_x^{k_2} & q_y^{k_2} \\ 1 & q_x^{k_3} & q_y^{k_3} \end{vmatrix} \quad D_1 = \begin{vmatrix} 1 & x & y \\ 1 & q_x^{k_2} & q_y^{k_2} \\ 1 & q_x^{k_3} & q_y^{k_3} \end{vmatrix} \quad D_2 = \begin{vmatrix} 1 & q_x^{k_1} & q_y^{k_1} \\ 1 & x & y \\ 1 & q_x^{k_3} & q_y^{k_3} \end{vmatrix} \quad D_3 = \begin{vmatrix} 1 & q_x^{k_1} & q_y^{k_1} \\ 1 & q_x^{k_2} & q_y^{k_2} \\ 1 & x & y \end{vmatrix}$$

$$\xi = \frac{1}{2}D_1/|T_k| \qquad \eta = \frac{1}{2}D_2/|T_k| \qquad \text{then } 1-\xi-\eta = \frac{1}{2}D_3/|T_k|$$

For a two dimensional domain or a border of three dimensional domain $\Omega_h = \sum_{k=1}^{n_t} T_k,\ \mathcal{T}_h = \{T_k\}$, we can calculate the integral over Ω_h by:

$\int_{\Omega_h} f(x,y)$ =`int2d(Th)(f)` =`int2d(Th, qft=*)(f)` =`int2d(Th, qforder=*)(f)`

where * stands for the name of quadrature formula or the order of the Gauss formula.

Quadrature formula on a triangle					
L	`qft`	`qforder`	Point in T_k	ω_ℓ	Exact on $P_k,\ k=$
1	`qf1pT`	2	$\left(\frac{1}{3},\frac{1}{3}\right)$	$\lvert T_k\rvert$	1
3	`qf2pT`	3	$\left(\frac{1}{2},\frac{1}{2}\right)$ $\left(\frac{1}{2},0\right)$ $\left(0,\frac{1}{2}\right)$	$\lvert T_k\rvert/3$ $\lvert T_k\rvert/3$ $\lvert T_k\rvert/3$	2
7	`qf5pT`	6	$\left(\frac{1}{3},\frac{1}{3}\right)$ $\left(\frac{6-\sqrt{15}}{21},\frac{6-\sqrt{15}}{21}\right)$ $\left(\frac{6-\sqrt{15}}{21},\frac{9+2\sqrt{15}}{21}\right)$ $\left(\frac{9+2\sqrt{15}}{21},\frac{6-\sqrt{15}}{21}\right)$ $\left(\frac{6+\sqrt{15}}{21},\frac{6+\sqrt{15}}{21}\right)$ $\left(\frac{6+\sqrt{15}}{21},\frac{9-2\sqrt{15}}{21}\right)$ $\left(\frac{9-2\sqrt{15}}{21},\frac{6+\sqrt{15}}{21}\right)$	$0.225\lvert T_k\rvert$ $\frac{(155-\sqrt{15})\lvert T_k\rvert}{1200}$ $\frac{(155-\sqrt{15})\lvert T_k\rvert}{1200}$ $\frac{(155-\sqrt{15})\lvert T_k\rvert}{1200}$ $\frac{(155+\sqrt{15})\lvert T_k\rvert}{1200}$ $\frac{(155+\sqrt{15})\lvert T_k\rvert}{1200}$ $\frac{(155+\sqrt{15})\lvert T_k\rvert}{1200}$	5
3	`qf1pTlump`		$(0,0)$ $(1,0)$ $(0,1)$	$\lvert T_k\rvert/3$ $\lvert T_k\rvert/3$ $\lvert T_k\rvert/3$	1
9	`qf2pT4P1`		$\left(\frac{1}{4},\frac{3}{4}\right)$ $\left(\frac{3}{4},\frac{1}{4}\right)$ $\left(0,\frac{1}{4}\right)$ $\left(0,\frac{3}{4}\right)$ $\left(\frac{1}{4},0\right)$ $\left(\frac{3}{4},0\right)$ $\left(\frac{1}{4},\frac{1}{4}\right)$ $\left(\frac{1}{4},\frac{1}{2}\right)$ $\left(\frac{1}{2},\frac{1}{4}\right)$	$\lvert T_k\rvert/12$ $\lvert T_k\rvert/12$ $\lvert T_k\rvert/12$ $\lvert T_k\rvert/12$ $\lvert T_k\rvert/12$ $\lvert T_k\rvert/12$ $\lvert T_k\rvert/6$ $\lvert T_k\rvert/6$ $\lvert T_k\rvert/6$	1
15	`qf7pT`	8	See [TAYLOR2005] for detail		7
21	`qf9pT`	10	See [TAYLOR2005] for detail		9

For a three dimensional domain $\Omega_h = \sum_{k=1}^{n_t} T_k,\ \mathcal{T}_h = \{T_k\}$, we can calculate the integral over Ω_h by:

$\int_{\Omega_h} f(x,y)$ `=int3d(Th)(f) =int3d(Th,qfV=*)(f) =int3D(Th,qforder=*)(f)`

where * stands for the name of quadrature formula or the order of the Gauss formula.

Quadrature formula on a tetrahedron					
L	`qfV`	`qforder`	Point in $T_k \in \mathbb{R}^3$	ω_ℓ	Exact on $P_k,\ k=$
1	`qfV1`	2	$\left(\frac{1}{4},\frac{1}{4},\frac{1}{4}\right)$	$\lvert T_k\rvert$	1
4	`qfV2`	3	$G4(0.58\ldots,0.13\ldots,0.13\ldots)$	$\lvert T_k\rvert/4$	2
14	`qfV5`	6	$G4(0.72\ldots,0.092\ldots,0.092\ldots)$ $G4(0.067\ldots,0.31\ldots,0.31\ldots)$ $G6(0.45\ldots,0.045\ldots,0.45\ldots)$	$0.073\ldots\lvert T_k\rvert$ $0.11\ldots\lvert T_k\rvert$ $0.042\ldots\lvert T_k\rvert$	5
4	`qfV1lump`		$G4(1,0,0)$	$\lvert T_k\rvert/4$	1

Where $G4(a,b,b)$ such that $a+3b=1$ is the set of the four point in barycentric coordinate:

$$\{(a,b,b,b),(b,a,b,b),(b,b,a,b),(b,b,b,a)\}$$

and where $G6(a,b,b)$ such that $2a+2b=1$ is the set of the six points in barycentric coordinate:

$$\{(a,a,b,b),(a,b,a,b),(a,b,b,a),(b,b,a,a),(b,a,b,a),(b,a,a,b)\}$$

Note: These tetrahedral quadrature formulae come from http://nines.cs.kuleuven.be/research/ecf/mtables.html

Note: By default, we use the formula which is exact for polynomials of degree 5 on triangles or edges (in bold in three tables).

It is possible to add an own quadrature formulae with using plugin `qf11to25` on segment, triangle or Tetrahedron.

The quadrature formulae in D dimension is a bidimentional array of size $N_q \times (D+1)$ such that the $D+1$ value of on row $i=0,...,N_p-1$ are $w^i, \hat{x}^i_1, ..., \hat{x}^i_D$ where w^i is the weight of the quadrature point, and $1-\sum_{k=1}^{D}\hat{x}^i_k, \hat{x}^i_1, ..., \hat{x}^i_D$ is the barycentric coordinate the quadrature point.

```
load "qf11to25"

// Quadrature on segment
real[int, int] qq1 = [
    [0.5, 0],
    [0.5, 1]
];

QF1 qf1(1, qq1); //def of quadrature formulae qf1 on segment
//remark:
//1 is the order of the quadrature exact for polynome of degree < 1

//Quadrature on triangle
real[int, int] qq2 = [
    [1./3., 0, 0],
    [1./3., 1, 0],
    [1./3., 0, 1]
];

QF2 qf2(1, qq2); //def of quadrature formulae qf2 on triangle
//remark:
//1 is the order of the quadrature exact for polynome of degree < 1
//so must have sum w^i = 1

// Quadrature on tetrahedron
real[int, int] qq3 = [
    [1./4., 0, 0, 0],
    [1./4., 1, 0, 0],
    [1./4., 0, 1, 0],
    [1./4., 0, 0, 1]
];

QF3 qf3(1, qq3); //def of quadrature formulae qf3 on get
//remark:
//1 is the order of the quadrature exact for polynome of degree < 1)

// Verification in 1d and 2d
mesh Th = square(10, 10);

```

(continues on next page)

(continued from previous page)

```
real I1 = int1d(Th, qfe=qf1)(x^2);
real I1l = int1d(Th, qfe=qf1pElump)(x^2);

real I2 = int2d(Th, qft=qf2)(x^2);
real I2l = int2d(Th, qft=qf1pTlump)(x^2);

cout << I1 << " == " << I1l << endl;
cout << I2 << " == " << I2l << endl;
assert( abs(I1-I1l) < 1e-10 );
assert( abs(I2-I2l) < 1e-10 );
```

The output is

```
1.67 == 1.67
0.335 == 0.335
```

3.3.12 Variational Form, Sparse Matrix, PDE Data Vector

In **FreeFEM** it is possible to define variational forms, and use them to build matrices and vectors, and store them to speed-up the script (4 times faster here).

For example let us solve the *Thermal Conduction problem*.

The variational formulation is in $L^2(0,T;H^1(\Omega))$; we shall seek u^n satisfying:

$$\forall w \in V_0; \qquad \int_\Omega \frac{u^n - u^{n-1}}{\delta t} w + \kappa \nabla u^n \nabla w) + \int_\Gamma \alpha(u^n - u_{ue}) w = 0$$

where $V_0 = \{w \in H^1(\Omega)/w_{|\Gamma_{24}} = 0\}$.

So to code the method with the matrices $A = (A_{ij})$, $M = (M_{ij})$, and the vectors $u^n, b^n, b', b", b_{cl}$ (notation if w is a vector then w_i is a component of the vector).

$$u^n = A^{-1} b^n, \qquad b' = b_0 + M u^{n-1}, \quad b" = \frac{1}{\varepsilon}\, b_{cl}, \quad b_i^n = \begin{cases} b"_i & \text{if } i \in \Gamma_{24} \\ b'_i & \text{else if } \notin \Gamma_{24} \end{cases}$$

Where with $\frac{1}{\varepsilon} = \texttt{tgv} = 10^{30}$:

$$\begin{array}{rcl} A_{ij} & = & \begin{cases} \frac{1}{\varepsilon} & \text{if } i \in \Gamma_{24}, \text{and} j = i \\ \int_\Omega w_j w_i / dt + k(\nabla w_j . \nabla w_i) + \int_{\Gamma_{13}} \alpha w_j w_i & \text{else if } i \notin \Gamma_{24}, \text{or} j \neq i \end{cases} \\ M_{ij} & = & \begin{cases} \frac{1}{\varepsilon} & \text{if } i \in \Gamma_{24}, \text{and} j = i \\ \int_\Omega w_j w_i / dt & \text{else if } i \notin \Gamma_{24}, \text{or} j \neq i \end{cases} \\ b_{0,i} & = & \int_{\Gamma_{13}} \alpha u_{ue} w_i \\ b_{cl} & = & u^0 \quad \text{the initial data} \end{array}$$

```
// Parameters
func fu0 = 10 + 90*x/6;
func k = 1.8*(y<0.5) + 0.2;
real ue = 25.;
real alpha = 0.25;
real T = 5;
real dt = 0.1 ;
```

(continues on next page)

(continued from previous page)

```

// Mesh
mesh Th = square(30, 5, [6*x, y]);

// Fespace
fespace Vh(Th, P1);
Vh u0 = fu0, u = u0;
```

Create three variational formulation, and build the matrices A,M.

```
// Problem
varf vthermic (u, v)
    = int2d(Th)(
        u*v/dt
        + k*(dx(u)*dx(v) + dy(u)*dy(v))
    )
    + int1d(Th, 1, 3)(
        alpha*u*v
    )
    + on(2, 4, u=1)
    ;

varf vthermic0 (u, v)
    = int1d(Th, 1, 3)(
        alpha*ue*v
    )
    ;

varf vMass (u, v)
    = int2d(Th)(
        u*v/dt
    )
    + on(2, 4, u=1)
    ;

real tgv = 1e30;
matrix A = vthermic(Vh, Vh, tgv=tgv, solver=CG);
matrix M = vMass(Vh, Vh);
```

Now, to build the right hand size we need 4 vectors.

```
real[int] b0 = vthermic0(0, Vh); //constant part of the RHS
real[int] bcn = vthermic(0, Vh); //tgv on Dirichlet boundary node ( !=0 )
//we have for the node i : i in Gamma_24 -> bcn[i] != 0
real[int] bcl = tgv*u0[]; //the Dirichlet boundary condition part
```

Note: The boundary condition is implemented by penalization and vector `bcn` contains the contribution of the boundary condition $u = 1$, so to change the boundary condition, we have just to multiply the vector `bc[]` by the current value `f` of the new boundary condition term by term with the operator `.*`.

Uzawa model gives a real example of using all this features.

And the new version of the algorithm is now:

```
// Time loop
ofstream ff("thermic.dat");
for(real t = 0; t < T; t += dt){
    // Update
    real[int] b = b0; //for the RHS
    b += M*u[]; //add the the time dependent part
    //lock boundary part:
    b = bcn ? bcl : b; //do forall i: b[i] = bcn[i] ? bcl[i] : b[i]

    // Solve
    u[] = A^-1*b;

    // Save
    ff << t << " " << u(3, 0.5) << endl;

    // Plot
    plot(u);
}

// Display
for(int i = 0; i < 20; i++)
    cout << dy(u)(6.0*i/20.0, 0.9) << endl;

// Plot
plot(u, fill=true, wait=true);
```

Note: The functions appearing in the variational form are formal and local to the `varf` definition, the only important thing is the order in the parameter list, like in:

```
varf vb1([u1, u2], q) = int2d(Th)((dy(u1) + dy(u2))*q) + int2d(Th)(1*q);
varf vb2([v1, v2], p) = int2d(Th)((dy(v1) + dy(v2))*p) + int2d(Th)(1*p);
```

To build matrix A from the bilinear part the variational form a of type `varf` simply write:

```
A = a(Vh, Wh , []...]);
// where
//Vh is "fespace" for the unknown fields with a correct number of component
//Wh is "fespace" for the test fields with a correct number of component
```

Possible named parameters in `, [...]` are

- `solver=` `LU`, `CG`, `Crout`, `Cholesky`, `GMRES`, `sparsesolver`, `UMFPACK` ...

  The default solver is `GMRES`.

  The storage mode of the matrix of the underlying linear system depends on the type of solver chosen; for `LU` the matrix is sky-line non symmetric, for `Crout` the matrix is sky-line symmetric, for `Cholesky` the matrix is sky-line symmetric positive definite, for `CG` the matrix is sparse symmetric positive, and for `GMRES`, `sparsesolver` or `UMFPACK` the matrix is just sparse.

- `factorize =` If true then do the matrix factorization for `LU`, `Cholesky` or `Crout`, the default value is `false`.

- `eps=` A real expression.

  ε sets the stopping test for the iterative methods like `CG`.

Note that if ε is negative then the stopping test is:

$$||Ax - b|| < |\varepsilon|$$

if it is positive then the stopping test is

$$||Ax - b|| < \frac{|\varepsilon|}{||Ax_0 - b||}$$

- `precon`= Name of a function (for example `P`) to set the preconditioner.

  The prototype for the function `P` must be:

```
func real[int] P(real[int] & xx) ;
```

- `tgv`= Huge value (10^{30}) used to implement Dirichlet boundary conditions.
- `tolpivot`= Set the tolerance of the pivot in `UMFPACK` ($10^{-}1$) and, `LU`, `Crout`, `Cholesky` factorization (10^{-20}).
- `tolpivotsym`= Set the tolerance of the pivot sym in `UMFPACK`
- `strategy`= Set the integer UMFPACK strategy (0 by default).

Note: The line of the matrix corresponding to the space `Wh` and the column of the matrix corresponding to the space `Vh`.

To build the dual vector `b` (of type `real[int]`) from the linear part of the variational form `a` do simply:

```
real b(Vh.ndof);
b = a(0, Vh);
```

A first example to compute the area of each triangle K of mesh Th, just do:

```
fespace Nh(Th, P0); //the space function constant / triangle
Nh areaK;
varf varea (unused, chiK) = int2d(Th)(chiK);
etaK[] = varea(0, Ph);
```

Effectively, the basic functions of space Nh, are the characteristic function of the element of Th, and the numbering is the numeration of the element, so by construction:

$$\texttt{etaK}[i] = \int 1_{|K_i} = \int_{K_i} 1;$$

Now, we can use this to compute error indicators like in example *Adaptation using residual error indicator*.

First to compute a continuous approximation to the function h "density mesh size" of the mesh Th.

```
fespace Vh(Th, P1);
Vh h ;
real[int] count(Th.nv);
varf vmeshsizen (u, v) = intalledges(Th, qfnbpE=1)(v);
varf vedgecount (u, v) = intalledges(Th, qfnbpE=1)(v/lenEdge);

// Computation of the mesh size
count = vedgecount(0, Vh); //number of edge / vertex
h[] = vmeshsizen(0, Vh); //sum length edge / vertex
h[] = h[]./count; //mean length edge / vertex
```

To compute error indicator for Poisson equation:

$$\eta_K = \int_K h_K^2 |(f + \Delta u_h)|^2 + \int_{\partial K} h_e |[\frac{\partial u_h}{\partial n}]|^2$$

where h_K is size of the longest edge (`hTriangle`), h_e is the size of the current edge (`lenEdge`), n the normal.

```
fespace Nh(Th, P0); // the space function constant / triangle
Nh etak;
varf vetaK (unused, chiK)
    = intalledges(Th)(
        chiK*lenEdge*square(jump(N.x*dx(u) + N.y*dy(u)))
    )
    + int2d(Th)(
        chiK*square(hTriangle*(f + dxx(u) + dyy(u)))
    )
    ;

etak[] = vetaK(0, Ph);
```

We add automatic expression optimization by default, if this optimization creates problems, it can be removed with the keyword `optimize` as in the following example:

```
varf a (u1, u2)
    = int2d(Th, optimize=0)(
        dx(u1)*dx(u2)
        + dy(u1)*dy(u2)
    )
    + on(1, 2, 4, u1=0)
    + on(3, u1=1)
    ;
```

or you can also do optimization and remove the check by setting `optimize=2`.

Remark, it is all possible to build interpolation matrix, like in the following example:

```
mesh TH = square(3, 4);
mesh th = square(2, 3);
mesh Th = square(4, 4);

fespace VH(TH, P1);
fespace Vh(th, P1);
fespace Wh(Th, P1);

matrix B = interpolate(VH, Vh); //build interpolation matrix Vh->VH
matrix BB = interpolate(Wh, Vh); //build interpolation matrix Vh->Wh
```

and after some operations on sparse matrices are available for example:

```
int N = 10;
real [int, int] A(N, N); //a full matrix
real [int] a(N), b(N);
A = 0;
for (int i = 0; i < N; i++){
    A(i, i) = 1+i;
    if (i+1 < N) A(i, i+1) = -i;
    a[i] = i;
}
```

(continues on next page)

(continued from previous page)

```
b = A*b;
matrix sparseA = A;
cout << sparseA << endl;
sparseA = 2*sparseA + sparseA';
sparseA = 4*sparseA + sparseA*5;
matrix sparseB = sparseA + sparseA + sparseA; ;
cout << "sparseB = " << sparseB(0,0) << endl;
```

3.3.13 Interpolation matrix

It is also possible to store the matrix of a linear interpolation operator from a finite element space V_h to another W_h to `interpolate`(W_h,V_h,...) a function.

Note that the continuous finite functions are extended by continuity outside of the domain.

The named parameters of function `interpolate` are:

- `inside`= set true to create zero-extension.
- `t`= set true to get the transposed matrix
- `op`= set an integer written below
  - 0 the default value and interpolate of the function
  - 1 interpolate the ∂_x
  - 2 interpolate the ∂_y
  - 3 interpolate the ∂_z
- `U2Vc`= set the which is the component of W_h come in V_h in interpolate process in a int array so the size of the array is number of component of W_h, if the put -1 then component is set to 0, like in the following example: (by default the component number is unchanged).

```
fespace V4h(Th4, [P1, P1, P1, P1]);
fespace V3h(Th, [P1, P1, P1]);
int[int] u2vc = [1, 3, -1]; //-1 -> put zero on the component
matrix IV34 = interpolate(V3h, V4h, inside=0, U2Vc=u2vc); //V3h <- V4h
V4h [a1, a2, a3, a4] = [1, 2, 3, 4];
V3h [b1, b2, b3] = [10, 20, 30];
b1[] = IV34*a1[];
```

So here we have: `freefem b1 == 2, b2 == 4, b3 == 0 ...`

Tip: Matrix interpolation

```
// Mesh
mesh Th = square(4, 4);
mesh Th4 = square(2, 2, [x*0.5, y*0.5]);
plot(Th, Th4, wait=true);

// Fespace
fespace Vh(Th, P1);
Vh v, vv;
fespace Vh4(Th4, P1);
Vh4 v4=x*y;
```

(continues on next page)

(continued from previous page)

```

fespace Wh(Th, P0);
fespace Wh4(Th4, P0);

// Interpolation
matrix IV = interpolate(Vh, Vh4); //here the function is exended by continuity
cout << "IV Vh<-Vh4 " << IV << endl;

v=v4;
vv[]= IV*v4[]; //here v == vv

real[int] diff= vv[] - v[];
cout << "|| v - vv || = " << diff.linfty << endl;
assert( diff.linfty<= 1e-6);

matrix IV0 = interpolate(Vh, Vh4, inside=1); //here the function is exended by zero
cout << "IV Vh<-Vh4 (inside=1) " << IV0 << endl;

matrix IVt0 = interpolate(Vh, Vh4, inside=1, t=1);
cout << "IV Vh<-Vh4^t (inside=1) " << IVt0 << endl;

matrix IV4t0 = interpolate(Vh4, Vh);
cout << "IV Vh4<-Vh^t " << IV4t0 << endl;

matrix IW4 = interpolate(Wh4, Wh);
cout << "IV Wh4<-Wh " << IW4 << endl;

matrix IW4V = interpolate(Wh4, Vh);
cout << "IV Wh4<-Vh " << IW4 << endl;
```

Build interpolation matrix A at a array of points $(xx[j], yy[j]),\ i = 0,\ 2$ here:

$$a_i j = dop(w_c^i(xx[j], yy[j]))$$

where w_i is the basic finite element function, c the component number, dop the type of diff operator like in op def.

```
real[int] xx = [.3, .4], yy = [.1, .4];
int c = 0, dop = 0;
matrix Ixx = interpolate(Vh, xx, yy, op=dop, composante=c);
cout << Ixx << endl;
Vh ww;
real[int] dd = [1, 2];
ww[] = Ixx*dd;
```

Tip: Schwarz

The following shows how to implement with an interpolation matrix a domain decomposition algorithm based on Schwarz method with Robin conditions.

Given a non-overlapping partition $\bar{\Omega} = \bar{\Omega}_0 \cup \bar{\Omega}_1$ with $\Omega_0 \cap \Omega_1 = \emptyset$, $\Sigma := \bar{\Omega}_0 \cap \bar{\Omega}_1$ the algorithm is:

$$-\Delta u_i = f \text{ in } \Omega_i,\ i = 0, 1,$$

$$\frac{\partial(u_1 - u_0)}{\partial n} + \alpha(u_1 - u_0) = 0 \text{ on } \Sigma.$$

The same in variational form is:

$$\int_{\Omega_i} \nabla u_i \cdot \nabla v \quad + \int_{\Sigma} \alpha u_i v = \int_{\Omega_i} f v$$

$$- \int_{\Omega_j} (\nabla u_j \cdot \nabla v - f v) + \int_{\Sigma} \alpha u_j v, \;\; \forall v \in H_0^1(\Omega), i, j = [0,1] \cup [1,0]$$

To discretized with the P^1 triangular Lagrangian finite element space V_h simply replace $H_0^1(\Omega)$ by $V_h(\Omega_0) \cup V_h(\Omega_1)$. Then difficulty is to compute $\int_{\Omega_j} \nabla u_j \cdot \nabla v$ when v is a basis function of $V_h(\Omega_i)$, $i \neq j$.

It is done as follows (with $\Gamma = \partial\Omega$):

```
// Parameters
int n = 30;
int Gamma = 1;
int Sigma = 2;

func f = 1.;
real alpha = 1.;

int Niter = 50;

// Mesh
mesh[int] Th(2);
int[int] reg(2);

border a0(t=0, 1){x=t; y=0; label=Gamma;}
border a1(t=1, 2){x=t; y=0; label=Gamma;}
border b1(t=0, 1){x=2; y=t; label=Gamma;}
border c1(t=2, 1){x=t; y=1; label=Gamma;}
border c0(t=1, 0){x=t; y=1; label=Gamma;}
border b0(t=1, 0){x=0; y=t; label=Gamma;}
border d(t=0, 1){x=1; y=t; label=Sigma;}
plot(a0(n) + a1(n) + b1(n) + c1(n) + c0(n) + b0(n) + d(n));
mesh TH = buildmesh(a0(n) + a1(n) + b1(n) + c1(n) + c0(n) + b0(n) + d(n));

reg(0) = TH(0.5, 0.5).region;
reg(1) = TH(1.5, 0.5).region;

for(int i = 0; i < 2; i++) Th[i] = trunc(TH, region==reg(i));

// Fespace
fespace Vh0(Th[0], P1);
Vh0 u0 = 0;

fespace Vh1(Th[1], P1);
Vh1 u1 = 0;

// Macro
macro grad(u) [dx(u), dy(u)] //

// Problem
int i;
varf a (u, v)
    = int2d(Th[i])(
        grad(u)'*grad(v)
    )
    + int1d(Th[i], Sigma)(
        alpha*u*v
    )
```

(continues on next page)

(continued from previous page)

```
    + on(Gamma, u=0)
    ;

varf b (u, v)
    = int2d(Th[i])(
        f*v
    )
    + on(Gamma, u=0)
    ;

varf du1dn (u, v)
    =-int2d(Th[1])(
        grad(u1)'*grad(v)
        - f*v
    )
    + int1d(Th[1], Sigma)(
        alpha*u1*v
    )
    +on(Gamma, u=0)
    ;

varf du0dn (u, v)
    =-int2d(Th[0])(
        grad(u0)'*grad(v)
        - f*v
    )
    + int1d(Th[0], Sigma)(
        alpha*u0*v
    )
    +on(Gamma, u=0)
    ;

matrix I01 = interpolate(Vh1, Vh0);
matrix I10 = interpolate(Vh0, Vh1);

matrix[int] A(2);
i = 0; A[i] = a(Vh0, Vh0);
i = 1; A[i] = a(Vh1, Vh1);

// Solving loop
for(int iter = 0; iter < Niter; iter++){
    // Solve on Th[0]
    {
        i = 0;
        real[int] b0 = b(0, Vh0);
        real[int] Du1dn = du1dn(0, Vh1);
        real[int] Tdu1dn(Vh0.ndof); Tdu1dn = I01'*Du1dn;
        b0 += Tdu1dn;
        u0[] = A[0]^-1*b0;
    }
    // Solve on Th[1]
    {
        i = 1;
        real[int] b1 = b(0, Vh1);
        real[int] Du0dn = du0dn(0, Vh0);
        real[int] Tdu0dn(Vh1.ndof); Tdu0dn = I10'*Du0dn;
        b1 += Tdu0dn;
```

(continues on next page)

(continued from previous page)

```
        u1[] = A[1]^-1*b1;
    }
    plot(u0, u1, cmm="iter="+iter);
}
```

3.3.14 Finite elements connectivity

Here, we show how to get informations on a finite element space $W_h(\mathcal{T}_n, *)$, where "*" may be `P1, P2, P1nc`, etc.

- `Wh.nt` gives the number of element of W_h
- `Wh.ndof` gives the number of degrees of freedom or unknown
- `Wh.ndofK` gives the number of degrees of freedom on one element
- `Wh(k,i)` gives the number of ith degrees of freedom of element k.

See the following example:

Tip: Finite element connectivity

```
// Mesh
mesh Th = square(5, 5);

// Fespace
fespace Wh(Th, P2);

cout << "Number of degree of freedom = " << Wh.ndof << endl;
cout << "Number of degree of freedom / ELEMENT = " << Wh.ndofK << endl;

int k = 2, kdf = Wh.ndofK; //element 2
cout << "Degree of freedom of element " << k << ":" << endl;
for (int i = 0; i < kdf; i++)
    cout << Wh(k,i) << " ";
cout << endl;
```

The output is:

```
Number of degree of freedom = 121
Number of degree of freedom / ELEMENT = 6
Degree of freedom of element 2:
78 95 83 87 79 92
```

3.4 Visualization

Results created by the finite element method can be a huge set of data, so it is very important to render them easy to grasp.

There are two ways of visualization in **FreeFEM**:

- One, the default view, which supports the drawing of meshes, isovalues of real FE-functions, and of vector fields, all by the command `plot` (see *Plot section* below). For publishing purpose, **FreeFEM** can store these plots as postscript files.
- Another method is to use external tools, for example, gnuplot (see *Gnuplot section*, *medit section*, *Paraview section*, *Matlab/Octave section*) using the command `system` to launch them and/or to save the data in text files.

3.4.1 Plot

With the command `plot`, meshes, isovalues of scalar functions, and vector fields can be displayed.

The parameters of the plot command can be meshes, real FE functions, arrays of 2 real FE functions, arrays of two double arrays, to plot respectively a mesh, a function, a vector field, or a curve defined by the two double arrays.

Note: The length of an arrow is always bound to be in [5‰, 5%] of the screen size in order to see something.

The `plot` command parameters are listed in the *Reference part*.

The keyboard shortcuts are:

- **enter** tries to show plot
- **p** previous plot (10 plots saved)
- **?** shows this help
- **+,-** zooms in/out around the cursor 3/2 times
- **=** resets the view
- **r** refreshes plot
- **up, down, left, right** special keys to tanslate
- **3** switches 3d/2d plot keys :
  - **z,Z** focal zoom and zoom out
  - **H,h** increases or decreases the Z scale of the plot
- **mouse motion**:
  - **left button** rotates
  - **right button** zooms (ctrl+button on mac)
  - **right button +alt** tanslates (alt+ctrl+button on mac)
- **a,A** increases or decreases the arrow size
- **B** switches between showing the border meshes or not
- **i,I** updates or not: the min/max bound of the functions to the window
- **n,N** decreases or increases the number of iso value arrays
- **b** switches between black and white or color plotting
- **g** switches between grey or color plotting
- **f** switches between filling iso or iso line
- **l** switches between lighting or not

- **v** switches between show or not showing the numerical value of colors
- **m** switches between show or not showing the meshes
- **w** window dump in file ffglutXXXX.ppm
- ***** keep/drop viewpoint for next plot
- **k** complex data / change view type
- **ESC** closes the graphics process before version 3.22, after no way to close
- **otherwise** does nothing

For example:

```
real[int] xx(10), yy(10);

mesh Th = square(5,5);

fespace Vh(Th, P1);

//plot scalar and vectorial FE function
Vh uh=x*x+y*y, vh=-y^2+x^2;
plot(Th, uh, [uh, vh], value=true, ps="three.eps", wait=true);

//zoom on box defined by the two corner points [0.1,0.2] and [0.5,0.6]
plot(uh, [uh, vh], bb=[[0.1, 0.2], [0.5, 0.6]],
   wait=true, grey=true, fill=true, value=true, ps="threeg.eps");

//compute a cut
for (int i = 0; i < 10; i++){
   x = i/10.;
   y = i/10.;
   xx[i] = i;
   yy[i] = uh; //value of uh at point (i/10., i/10.)
}
plot([xx, yy], ps="likegnu.eps", wait=true);
```

To change the color table and to choose the value of iso line you can do:

```
// from: \url{http://en.wikipedia.org/wiki/HSV_color_space}
// The HSV (Hue, Saturation, Value) model defines a color space
// in terms of three constituent components:
// HSV color space as a color wheel
// Hue, the color type (such as red, blue, or yellow):
// Ranges from 0-360 (but normalized to 0-100% in some applications, like here)
// Saturation, the "vibrancy" of the color: Ranges from 0-100%
// The lower the saturation of a color, the more "grayness" is present
// and the more faded the color will appear.
// Value, the brightness of the color: Ranges from 0-100%

mesh Th = square(10, 10, [2*x-1, 2*y-1]);

fespace Vh(Th, P1);
Vh uh=2-x*x-y*y;

real[int] colorhsv=[ // color hsv model
   4./6., 1 , 0.5, // dark blue
   4./6., 1 , 1, // blue
```

(continues on next page)

(a) Mesh, isovalue and vector

(b) Enlargement in grey of isovalue and vector

(c) Plots a cut of `uh`. Note that a refinement of the same can be obtained in combination with gnuplot

Fig. 3.38: Plot

(a) HSV color cylinder

(b) Isovalue with an other color table

Fig. 3.39: HSV

(continued from previous page)

```
   5./6., 1 , 1, // magenta
   1, 1. , 1, // red
   1, 0.5 , 1 // light red
   ];
 real[int] viso(31);

 for (int i = 0; i < viso.n; i++)
   viso[i] = i*0.1;

 plot(uh, viso=viso(0:viso.n-1), value=true, fill=true, wait=true, hsv=colorhsv);
```

Note: See *HSV example* for the complete script.

3.4.2 Link with gnuplot

Example *Membrane* shows how to generate a gnuplot from a **FreeFEM** file. Here is another technique which has the advantage of being online, i.e. one doesn't need to quit **FreeFEM** to generate a gnuplot.

However, this works only if gnuplot is installed, and only on an Unix-like computer.

Add to the previous example:

```
{// file for gnuplot
   ofstream gnu("plot.gp");
   for (int i = 0; i < n; i++)
      gnu << xx[i] << " " << yy[i] << endl;
}

// to call gnuplot command and wait 5 second (due to the Unix command)
// and make postscript plot
exec("echo 'plot \"plot.gp\" w l \n pause 5 \n set term postscript \n set output \
↪"gnuplot.eps\" \n replot \n quit' | gnuplot");
```

Fig. 3.40: Plots a cut of uh with gnuplot

Note: See *Plot example* for the complete script.

3.4.3 Link with medit

As said above, `medit` is a freeware display package by Pascal Frey using OpenGL. Then you may run the following example.

Now `medit` software is included in **FreeFEM** under `ffmedit` name.

The `medit` command parameters are listed in the *Reference part*.

Fig. 3.41: :freefem:medit‘ plot

With version 3.2 or later

```
load "medit"

mesh Th = square(10, 10, [2*x-1, 2*y-1]);

fespace Vh(Th, P1);
Vh u=2-x*x-y*y;

medit("u", Th, u);
```

Before:

```
mesh Th = square(10, 10, [2*x-1, 2*y-1]);

fespace Vh(Th, P1);
Vh u=2-x*x-y*y;

savemesh(Th, "u", [x, y, u*.5]); //save u.points and u.faces file
// build a u.bb file for medit
{
   ofstream file("u.bb");
   file << "2 1 1 " << u[].n << " 2 \n";
   for (int j = 0; j < u[].n; j++)
```

(continues on next page)

(continued from previous page)

```
      file << u[][j] << endl;
}
//call medit command
exec("ffmedit u");
//clean files on unix-like OS
exec("rm u.bb u.faces u.points");
```

Note: See *Medit example* for the complete script.

3.4.4 Link with Paraview

One can also export mesh or results in the `.vtk` format in order to post-process data using Paraview.

```
load "iovtk"

mesh Th = square(10, 10, [2*x-1, 2*y-1]);

fespace Vh(Th, P1);
Vh u=2-x*x-y*y;

int[int] Order = [1];
string DataName = "u";
savevtk("u.vtu", Th, u, dataname=DataName, order=Order);
```

Fig. 3.42: Paraview plot

Warning: Finite element variables saved using paraview **must be in P0 or P1**

Note: See *Paraview example* for the complete script.

3.4.5 Link with Matlab© and Octave

In order to create a plot from a **FreeFEM** simulation in Octave and Matlab the mesh, the finite element space connectivity and the simulation data must be written to files:

```
include "ffmatlib.idp"

mesh Th = square(10, 10, [2*x-1, 2*y-1]);
fespace Vh(Th, P1);
Vh u=2-x*x-y*y;

savemesh(Th,"export_mesh.msh");
ffSaveVh(Th,Vh,"export_vh.txt");
ffSaveData(u,"export_data.txt");
```

Within Matlab or Octave the files can be plot with the ffmatlib library:

```
addpath('path to ffmatlib');
[p,b,t]=ffreadmesh('export_mesh.msh');
vh=ffreaddata('export_vh.txt');
u=ffreaddata('export_data.txt');
ffpdeplot(p,b,t,'VhSeq',vh,'XYData',u,'ZStyle','continuous','Mesh','on');
grid;
```

Fig. 3.43: Matlab / Octave plot

Note: For more Matlab / Octave plot examples have a look at the tutorial section *Matlab / Octave Examples* or visit the ffmatlib library on github.

3.5 Algorithms & Optimization

3.5.1 Conjugate Gradient/GMRES

Suppose we want to solve the Euler problem (here x has nothing to do with the reserved variable for the first coordinate in **FreeFEM**):

find $x \in \mathbb{R}^n$ such that

$$\nabla J(x) = \left(\frac{\partial J}{\partial x_i}(\mathbf{x})\right) = 0 \tag{3.17}$$

where J is a function (to minimize for example) from $\mathbb{R}^n$ to $\mathbb{R}$.

If the function is convex we can use the conjugate gradient algorithm to solve the problem, and we just need the function (named `dJ` for example) which computes ∇J, so the parameters are the name of that function with prototype `func real[int] dJ(real[int] &xx);` which computes ∇J, and a vector `x` of type (of course the number 20 can be changed) `real[int] x(20);` to initialize the process and get the result.

Given an initial value $\mathbf{x}^{(0)}$, a maximum number $i_{\max}$ of iterations, and an error tolerance $0 < \epsilon < 1$:

Put $\mathbf{x} = \mathbf{x}^{(0)}$ and write

```
NLCG(dJ, x, precon=M, nbiter=imax, eps=epsilon, stop=stopfunc);
```

will give the solution of $\mathbf{x}$ of $\nabla J(\mathbf{x}) = 0$. We can omit parameters `precon, nbiter, eps, stop`. Here M is the preconditioner whose default is the identity matrix.

The stopping test is

$$\|\nabla J(\mathbf{x})\|_P \leq \epsilon \|\nabla J(\mathbf{x}^{(0)})\|_P$$

Writing the minus value in `eps=`, i.e.,

```
NLCG(dJ, x, precon=M, nbiter=imax, eps=-epsilon);
```

We can use the stopping test:

$$\|\nabla J(\mathbf{x})\|_P^2 \leq \epsilon$$

The parameters of these three functions are:

- `nbiter=` set the number of iteration (by default 100)
- `precon=` set the preconditioner function (`P` for example) by default it is the identity, note the prototype is `func real[int] P(real[int] &x)`.
- `eps=` set the value of the stop test ε ($= 10^{-6}$ by default) if positive then relative test $||\nabla J(x)||_P \leq \varepsilon ||\nabla J(x_0)||_P$, otherwise the absolute test is $||\nabla J(x)||_P^2 \leq |\varepsilon|$.
- `veps=` set and return the value of the stop test, if positive, then relative test is $||\nabla J(x)||_P \leq \varepsilon ||\nabla J(x_0)||_P$, otherwise the absolute test is $||\nabla J(x)||_P^2 \leq |\varepsilon|$. The return value is minus the real stop test (remark: it is useful in loop).
- `stop= stopfunc` add your test function to stop before the `eps` criterion. The prototype for the function `stopfunc` is

  ```
  func bool stopfunc(int iter, real[int] u, real[int] g)
  ```

  where `u` is the current solution, and `g`, the current gradient, is not preconditioned.

Tip: *Algorithms.edp*

For a given function b, let us find the minimizer u of the function

$$\begin{array}{rcl} J(u) & = & \frac{1}{2}\int_\Omega f(|\nabla u|^2) - \int_\Omega ub \\ f(x) & = & ax + x - \ln(1+x), \quad f'(x) = a + \frac{x}{1+x}, \quad f''(x) = \frac{1}{(1+x)^2} \end{array}$$

under the boundary condition $u = 0$ on $\partial\Omega$.

```
fespace Ph(Th, P0);
Ph alpha; //store df(|nabla u|^2)

// The functionn J
//J(u) = 1/2 int_Omega f(|nabla u|^2) - int_Omega u b
func real J (real[int] & u){
   Vh w;
   w[] = u;
   real r = int2d(Th)(0.5*f(dx(w)*dx(w) + dy(w)*dy(w)) - b*w);
   cout << "J(u) = " << r << " " << u.min << " " << u.max << endl;
   return r;
}

// The gradiant of J
func real[int] dJ (real[int] & u){
   Vh w;
   w[] = u;
   alpha = df(dx(w)*dx(w) + dy(w)*dy(w));
   varf au (uh, vh)
      = int2d(Th)(
            alpha*(dx(w)*dx(vh) + dy(w)*dy(vh))
         - b*vh
      )
      + on(1, 2, 3, 4, uh=0)
      ;

   u = au(0, Vh);
   return u; //warning: no return of local array
}
```

We also want to construct a preconditioner C with solving the problem:

find $u_h \in V_{0h}$ such that:

$$\forall v_h \in V_{0h}, \quad \int_\Omega \alpha \nabla u_h . \nabla v_h = \int_\Omega b v_h$$

where $\alpha = f'(|\nabla u|^2)$.

```
alpha = df(dx(u)*dx(u) + dy(u)*dy(u));
varf alap (uh, vh)
   = int2d(Th)(
         alpha*(dx(uh)*dx(vh) + dy(uh)*dy(vh))
   )
   + on(1, 2, 3, 4, uh=0)
   ;

varf amass(uh, vh)
   = int2d(Th)(
         uh*vh
   )
   + on(1, 2, 3, 4, uh=0)
   ;

matrix Amass = amass(Vh, Vh, solver=CG);
matrix Alap= alap(Vh, Vh, solver=Cholesky, factorize=1);

// Preconditionner
```

(continues on next page)

(continued from previous page)

```
func real[int] C(real[int] & u){
   real[int] w = u;
   u = Alap^-1*w;
   return u; //warning: no return of local array variable
}
```

To solve the problem, we make 10 iterations of the conjugate gradient, recompute the preconditioner and restart the conjugate gradient:

```
int conv=0;
for(int i = 0; i < 20; i++){
   conv = NLCG(dJ, u[], nbiter=10, precon=C, veps=eps, verbosity=5);
   if (conv) break;

   alpha = df(dx(u)*dx(u) + dy(u)*dy(u));
   Alap = alap(Vh, Vh, solver=Cholesky, factorize=1);
   cout << "Restart with new preconditionner " << conv << ", eps =" << eps << endl;
}

// Plot
plot (u, wait=true, cmm="solution with NLCG");
```

For a given symmetric positive matrix A, consider the quadratic form

$$J(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T A\mathbf{x} - \mathbf{b}^T\mathbf{x}$$

then $J(\mathbf{x})$ is minimized by the solution $\mathbf{x}$ of $A\mathbf{x} = \mathbf{b}$. In this case, we can use the function `AffineCG`

```
AffineCG(A, x, precon=M, nbiter=imax, eps=epsilon, stop=stp);
```

If A is not symmetric, we can use GMRES(Generalized Minimum Residual) algorithm by

```
AffineGMRES(A, x, precon=M, nbiter=imax, eps=epsilon);
```

Also, we can use the non-linear version of GMRES algorithm (the function J is just convex)

```
AffineGMRES(dJ, x, precon=M, nbiter=imax, eps=epsilon);
```

For the details of these algorithms, refer to [PIRONNEAU1998], Chapter IV, 1.3.

3.5.2 Algorithms for Unconstrained Optimization

Two algorithms of COOOL package are interfaced with the Newton Raphson method (called `Newton`) and the `BFGS` method. These two are directly available in **FreeFEM** (no dynamical link to load). Be careful with these algorithms, because their implementation uses full matrices. We also provide several optimization algorithms from the NLopt library as well as an interface for Hansen's implementation of CMAES (a MPI version of this one is also available).

Example of usage for BFGS or CMAES

Tip: BFGS

```
real[int] b(10), u(10);

//J
func real J (real[int] & u){
   real s = 0;
   for (int i = 0; i < u.n; i++)
      s += (i+1)*u[i]*u[i]*0.5 - b[i]*u[i];
   if (debugJ)
      cout << "J = " << s << ", u = " << u[0] << " " << u[1] << endl;
   return s;
}

//the gradiant of J (this is a affine version (the RHS is in)
func real[int] DJ (real[int] &u){
   for (int i = 0; i < u.n; i++)
      u[i] = (i+1)*u[i];
   if (debugdJ)
      cout << "dJ: u = " << u[0] << " " << u[1] << " " << u[2] << endl;
   u -= b;
   if (debugdJ)
      cout << "dJ-b: u = " << u[0] << " " << u[1] << " " << u[2] << endl;
   return u; //return of global variable ok
}

b=1;
u=2;
BFGS(J, DJ, u, eps=1.e-6, nbiter=20, nbiterline=20);
cout << "BFGS: J(u) = " << J(u) << ", err = " << error(u, b) << endl;
```

It is almost the same a using the CMA evolution strategy except, that since it is a derivative free optimizer, the `dJ` argument is omitted and there are some other named parameters to control the behavior of the algorithm. With the same objective function as above, an example of utilization would be (see *CMAES Variational inequality* for a complete example):

```
load "ff-cmaes"
//define J, u, ...
real min = cmaes(J, u, stopTolFun=1e-6, stopMaxIter=3000);
cout << "minimum value is " << min << " for u = " << u << endl;
```

This algorithm works with a normal multivariate distribution in the parameters space and tries to adapt its covariance matrix using the information provided by the successive function evaluations (see NLopt documentation for more details). Therefore, some specific parameters can be passed to control the starting distribution, size of the sample generations, etc… Named parameters for this are the following:

- `seed=` Seed for random number generator (`val` is an integer). No specified value will lead to a clock based seed initialization.
- `initialStdDev=` Value for the standard deviations of the initial covariance matrix (`val` is a real). If the value σ is passed, the initial covariance matrix will be set to σI. The expected initial distance between initial X and the $argmin$ should be roughly initialStdDev. Default is 0.3.
- `initialStdDevs=` Same as above except that the argument is an array allowing to set a value of the initial standard deviation for each parameter. Entries differing by several orders of magnitude should be avoided (if it can't be, try rescaling the problem).
- `stopTolFun=` Stops the algorithm if function value differences are smaller than the passed one, default is 10^{-12}.

- `stopTolFunHist=` Stops the algorithm if function value differences from the best values are smaller than the passed one, default is 0 (unused).
- `stopTolX=` Stopping criteria is triggered if step sizes in the parameters space are smaller than this real value, default is 0.
- `stopTolXFactor=` Stopping criteria is triggered when the standard deviation increases more than this value. The default value is 10^3.
- `stopMaxFunEval=` Stops the algorithm when `stopMaxFunEval` function evaluations have been done. Set to $900(n+3)^2$ by default, where n is the parameters space dimension.
- `stopMaxIter=` Integer stopping the search when `stopMaxIter` generations have been sampled. Unused by default.
- `popsize=` Integer value used to change the sample size. The default value is $4 + \lfloor 3\ln(n) \rfloor$. Increasing the population size usually improves the global search capabilities at the cost of, at most, a linear reduction of the convergence speed with respect to `popsize`.
- `paramFile=` This `string` type parameter allows the user to pass all the parameters using an extern file, as in Hansen's original code. More parameters related to the CMA-ES algorithm can be changed with this file. Note that the parameters passed to the CMAES function in the **FreeFEM** script will be ignored if an input parameters file is given.

3.5.3 IPOPT

The `ff-Ipopt` package is an interface for the IPOPT [WÄCHTER2006] optimizer. IPOPT is a software library for large scale, non-linear, constrained optimization. It implements a primal-dual interior point method along with filter method based line searches.

IPOPT needs a direct sparse symmetric linear solver. If your version of **FreeFEM** has been compiled with the `--enable-download` tag, it will automatically be linked with a sequential version of MUMPS. An alternative to MUMPS would be to download the HSL subroutines (see Compiling and Installing the Java Interface JIPOPT) and place them in the `/ipopt/Ipopt-3.10.2/ThirdParty/HSL` directory of the **FreeFEM** downloads folder before compiling.

Short description of the algorithm

In this section, we give a very brief glimpse at the underlying mathematics of IPOPT. For a deeper introduction on interior methods for nonlinear smooth optimization, one may consult [FORSGREN2002], or [WÄCHTER2006] for more IPOPT specific elements. IPOPT is designed to perform optimization for both equality and inequality constrained problems. However, nonlinear inequalities are rearranged before the beginning of the optimization process in order to restrict the panel of nonlinear constraints to those of the equality kind. Each nonlinear inequality is transformed into a pair of simple bound inequalities and nonlinear equality constraints by the introduction of as many slack variables as is needed : $c_i(x) \leq 0$ becomes $c_i(x) + s_i = 0$ and $s_i \leq 0$, where s_i is added to the initial variables of the problems x_i. Thus, for convenience, we will assume that the minimization problem does not contain any nonlinear inequality constraint. It means that, given a function $f : \mathbb{R}^n \mapsto \mathbb{R}$, we want to find:

$$\begin{aligned} x_0 &= \underset{x \in V}{\operatorname{argmin}} f(x) \\ \text{with } V &= \{x \in \mathbb{R}^n \mid c(x) = 0 \text{ and } x_l \leq x \leq x_u\} \end{aligned} \tag{3.18}$$

Where $c : \mathbb{R}^n \to \mathbb{R}^m$ and $x_l, x_u \in \mathbb{R}^n$ and inequalities hold componentwise. The f function as well as the constraints c should be twice-continuously differentiable.

As a barrier method, interior points algorithms try to find a Karush-Kuhn-Tucker point for (3.18) by solving a sequence of problems, unconstrained with respect to the inequality constraints, of the form:

$$\text{for a given } \mu > 0,\ \text{find } x_\mu = \underset{x\in\mathbb{R}^n \mid c(x)=0}{\operatorname{argmin}} B(x,\mu) \tag{3.19}$$

Where μ is a positive real number and

$$B(x,\mu) = f(x) - \mu\sum_{i=1}^{n}\ln(x_{u,i}-x_i) - \mu\sum_{i=1}^{m}\ln(x_i - x_{l,i})$$

The remaining equality constraints are handled with the usual Lagrange multipliers method. If the sequence of barrier parameters μ converge to 0, intuition suggests that the sequence of minimizers of (3.19) converge to a local constrained minimizer of (3.18). For a given μ, (3.19) is solved by finding $(x_\mu, \lambda_\mu) \in \mathbb{R}^n \times \mathbb{R}^m$ such that:

$$\begin{aligned} \nabla B(x_\mu,\mu) + \sum_{i=1}^{m}\lambda_{\mu,i}\nabla c_i(x_\mu) = \nabla B(x_\mu,\mu) + J_c(x_\mu)^T\lambda_\mu &= 0 \\ c(x_\mu) &= 0 \end{aligned} \tag{3.20}$$

The derivations for ∇B only holds for the x variables, so that:

$$\nabla B(x,\mu) = \nabla f(x) + \begin{pmatrix} \mu/(x_{u,1}-x_1) \\ \vdots \\ \mu/(x_{u,n}-x_n) \end{pmatrix} - \begin{pmatrix} \mu/(x_1-x_{l,1}) \\ \vdots \\ \mu/(x_n-x_{l,n}) \end{pmatrix}$$

If we respectively call $z_u(x,\mu) = (\mu/(x_{u,1}-x_1),\ldots,\mu/(x_{u,n}-x_n))$ and $z_l(x,\mu)$ the other vector appearing in the above equation, then the optimum (x_μ,λ_μ) satisfies:

$$\nabla f(x_\mu) + J_c(x_\mu)^T\lambda_\mu + z_u(x_\mu,\mu) - z_l(x_\mu,\mu) = 0 \quad \text{and} \quad c(x_\mu) = 0 \tag{3.21}$$

In this equation, the z_l and z_u vectors seem to play the role of Lagrange multipliers for the simple bound inequalities, and indeed, when $\mu \to 0$, they converge toward some suitable Lagrange multipliers for the KKT conditions, provided some technical assumptions are fulfilled (see [FORSGREN2002]).

Equation (3.21) is solved by performing a Newton method in order to find a solution of (3.20) for each of the decreasing values of μ. Some order 2 conditions are also taken into account to avoid convergence to local maximizers, see [FORSGREN2002] for details about them. In the most classic IP algorithms, the Newton method is directly applied to (3.20). This is in most case inefficient due to frequent computation of infeasible points. These difficulties are avoided in Primal-Dual interior point methods where (3.20) is transformed into an extended system where z_u and z_l are treated as unknowns and the barrier problems are finding $(x,\lambda,z_u,z_l) \in \mathbb{R}^n\times\mathbb{R}^m\times\mathbb{R}^n\times\mathbb{R}^n$ such that:

$$\left\{\begin{array}{rcl} \nabla f(x) + J_c(x)^T\lambda + z_u - z_l &=& 0 \\ c(x) &=& 0 \\ (X_u - X)z_u - \mu e &=& 0 \\ (X - X_l)z_l - \mu e &=& 0 \end{array}\right. \tag{3.22}$$

Where if a is a vector of $\mathbb{R}^n$, A denotes the diagonal matrix $A = (a_i\delta_{ij})_{1\le i,j\le n}$ and $e \in \mathbb{R}^n = (1,1,\ldots,1)$. Solving this nonlinear system by the Newton method is known as being the *primal-dual* interior point method. Here again, more details are available in [FORSGREN2002]. Most actual implementations introduce features in order to globalize the convergence capability of the method, essentially by adding some line-search steps to the Newton algorithm, or by using trust regions. For the purpose of IPOPT, this is achieved by a *filter line search* methods, the details of which can be found in [WÄCHTER2006].

More IPOPT specific features or implementation details can be found in [WÄCHTER2006]. We will just retain that IPOPT is a smart Newton method for solving constrained optimization problems, with global convergence capabilities due to a robust line search method (in the sense that the algorithm will converge no matter the initializer). Due to the underlying Newton method, the optimization process requires expressions of all derivatives up to the order 2 of the fitness function as well as those of the constraints. For problems whose Hessian matrices are difficult to compute or lead to high dimensional dense matrices, it is possible to use a BFGS approximation of these objects at the cost of a much slower convergence rate.

IPOPT in FreeFEM

Calling the IPOPT optimizer in a **FreeFEM** script is done with the `IPOPT` function included in the `ff-Ipopt` dynamic library. IPOPT is designed to solve constrained minimization problems in the form:

$$\begin{aligned}
&\text{find} x_0 = \underset{x\in\mathbb{R}^n}{\text{argmin}} f(x) \\
&\text{s.t.} \begin{cases} \forall i \leq n,\ x_i^{\text{lb}} \leq x_i \leq x_i^{\text{ub}} & \text{(simple bounds)} \\ \forall i \leq m,\ c_i^{\text{lb}} \leq c_i(x) \leq c_i^{\text{ub}} & \text{(constraints functions)} \end{cases}
\end{aligned}$$

Where ub and lb stand for "upper bound" and "lower bound". If for some $i, 1 \leq i \leq m$ we have $c_i^{\text{lb}} = c_i^{\text{ub}}$, it means that c_i is an equality constraint, and an inequality one if $c_i^{\text{lb}} < c_i^{\text{ub}}$.

There are different ways to pass the fitness function and constraints. The more general one is to define the functions using the keyword `func`. Any returned matrix must be a sparse one (type `matrix`, not a `real[int,int]`):

```
func real J (real[int] &X) {...} //Fitness Function, returns a scalar
func real[int] gradJ (real[int] &X) {...} //Gradient is a vector

func real[int] C (real[int] &X) {...} //Constraints
func matrix jacC (real[int] &X) {...} //Constraints Jacobian
```

Warning: In the current version of **FreeFEM**, returning a `matrix` object that is local to a function block leads to undefined results. For each sparse matrix returning function you define, an extern matrix object has to be declared, whose associated function will overwrite and return on each call. Here is an example for `jacC`:

```
matrix jacCBuffer; //just declare, no need to define yet
func matrix jacC (real[int] &X){
   ...//fill jacCBuffer
   return jacCBuffer;
}
```

Warning: IPOPT requires the structure of each matrix at the initialization of the algorithm. Some errors may occur if the matrices are not constant and are built with the `matrix A = [I,J,C]` syntax, or with an intermediary full matrix (`real[int,int]`), because any null coefficient is discarded during the construction of the sparse matrix. It is also the case when making matrices linear combinations, for which any zero coefficient will result in the suppression of the matrix from the combination. Some controls are available to avoid such problems. Check the named parameter descriptions (`checkindex`, `structhess` and `structjac` can help). We strongly advice to use `varf` as much as possible for the matrix forging.

The Hessian returning function is somewhat different because it has to be the Hessian of the Lagrangian function:

$$(x, \sigma_f, \lambda) \mapsto \sigma_f \nabla^2 f(x) + \sum_{i=1}^{m} \lambda_i \nabla^2 c_i(x) \text{ where } \lambda \in \mathbb{R}^m \text{ and } \sigma \in \mathbb{R}$$

Your Hessian function should then have the following prototype:

```
matrix hessianLBuffer; //Just to keep it in mind
func matrix hessianL (real[int] &X, real sigma, real[int] &lambda){...}
```

If the constraints functions are all affine, or if there are only simple bound constraints, or no constraint at all, the Lagrangian Hessian is equal to the fitness function Hessian, one can then omit the `sigma` and `lambda` parameters:

```
matrix hessianJBuffer;
func matrix hessianJ (real[int] &X){...} //Hessian prototype when constraints are affine
```

When these functions are defined, IPOPT is called this way:

```
real[int] Xi = ... ; //starting point
IPOPT(J, gradJ, hessianL, C, jacC, Xi, /*some named parameters*/);
```

If the Hessian is omitted, the interface will tell IPOPT to use the (L)BFGS approximation (it can also be enabled with a named parameter, see further). Simple bound or unconstrained problems do not require the constraints part, so the following expressions are valid:

```
IPOPT(J, gradJ, C, jacC, Xi, ... ); //IPOPT with BFGS
IPOPT(J, gradJ, hessianJ, Xi, ... ); //Newton IPOPT without constraints
IPOPT(J, gradJ, Xi, ... ); //BFGS, no constraints
```

Simple bounds are passed using the `lb` and `ub` named parameters, while constraint bounds are passed with the `clb` and `cub` ones. Unboundedness in some directions can be achieved by using the $1e^{19}$ and $-1e^{19}$ values that IPOPT recognizes as $+\infty$ and $-\infty$:

```
real[int] xlb(n), xub(n), clb(m), cub(m);
//fill the arrays...
IPOPT(J, gradJ, hessianL, C, jacC, Xi, lb=xlb, ub=xub, clb=clb, cub=cub, /*some other named parameters*/);
```

P2 fitness function and affine constraints function : In the case where the fitness function or constraints function can be expressed respectively in the following forms:

$$\begin{array}{rl} \forall x \in \mathbb{R}^n,\ f(x) = \frac{1}{2}\langle Ax, x\rangle + \langle b, x\rangle & (A,b) \in \mathcal{M}_{n,n}(\mathbb{R}) \times \mathbb{R}^n \\ \text{or, } C(x) = Ax + b & (A,b) \in \mathcal{M}_{n,m}(\mathbb{R}) \times \mathbb{R}^m \end{array}$$

where A and b are constant, it is possible to directly pass the (A, b) pair instead of defining 3 (or 2) functions. It also indicates to IPOPT that some objects are constant and that they have to be evaluated only once, thus avoiding multiple copies of the same matrix. The syntax is:

```
// Affine constraints with "standard" fitness function
matrix A = ... ; //linear part of the constraints
real[int] b = ... ; //constant part of constraints
IPOPT(J, gradJ, hessianJ, [A, b], Xi, /*bounds and named parameters*/);
//[b, A] would work as well.
```

Note that if you define the constraints in this way, they don't contribute to the Hessian, so the Hessian should only take one `real[int]` as an argument.

```
// Affine constraints and P2 fitness func
matrix A = ... ; //bilinear form matrix
real[int] b = ... ; //linear contribution to f
matrix Ac = ... ; //linear part of the constraints
real[int] bc = ... ; //constant part of constraints
IPOPT([A, b], [Ac, bc], Xi, /*bounds and named parameters*/);
```

If both objective and constraint functions are given this way, it automatically activates the IPOPT `mehrotra_algorithm` option (better for linear and quadratic programming according to the documentation). Otherwise, this option can only be set through the option file (see the named parameters section).

A false case is the one of defining f in this manner while using standard functions for the constraints:

```
matrix A = ... ; //bilinear form matrix
real[int] b = ... ; //linear contribution to f
func real[int] C(real[int] &X){...} //constraints
func matrix jacC(real[int] &X){...} //constraints Jacobian
IPOPT([A, b], C, jacC, Xi, /*bounds and named parameters*/);
```

Indeed, when passing `[A, b]` in order to define f, the Lagrangian Hessian is automatically built and has the constant $x \mapsto A$ function, with no way to add possible constraint contributions, leading to incorrect second order derivatives. So, a problem should be defined like that in only two cases:

1. constraints are nonlinear but you want to use the BFGS mode (then add `bfgs=1` to the named parameter),
2. constraints are affine, but in this case, compatible to pass in the same way

Here are some other valid definitions of the problem (cases when f is a pure quadratic or linear form, or C a pure linear function, etc...):

```
// Pure quadratic f - A is a matrix
IPOPT(A, /*constraints arguments*/, Xi, /*bound and named parameters*/);
// Pure linear f - b is a real[int]
IPOPT(b, /*constraints arguments*/, Xi, /*bound and named parameters*/);
// Linear constraints - Ac is a matrix
IPOPT(/*fitness function arguments*/, Ac, Xi, /*bound and named parameters*/);
```

Returned Value : The `IPOPT` function returns an error code of type `int`. A zero value is obtained when the algorithm succeeds and positive values reflect the fact that IPOPT encounters minor troubles. Negative values reveal more problematic cases. The associated IPOPT return tags are listed in the table below. The IPOPT pdf documentation provides a more accurate description of these return statuses:

Success	Failures
0 `Solve_Succeeded`	
1 `Solved_To_Acceptable_Level`	-1 `Maximum_Iterations_Exceeded`
2 `Infeasible_Problem_Detected`	-2 `Restoration_Failed`
3 `Search_Direction_Becomes_Too_Small`	-3 `Error_In_Step_Computation`
4 `Diverging_Iterates`	-4 `Maximum_CpuTime_Exceeded`
5 `User_Requested_Stop`	
6 `Feasible_Point_Found`	

Problem definition issues	Critical errors
-10 `Not_Enough_Degrees_Of_Freedom`	-100 `Unrecoverable_Exception`
-11 `Invalid_Problem_Definition`	-101 `NonIpopt_Exception_Thrown`
-12 `Invalid_Option`	-102 `Insufficient_Memory`
-13 `Invalid_Number_Detected`	-199 `Internal_Error`

Named Parameters : The available named parameters in this interface are those we thought to be the most subject to variations from one optimization to another, plus a few that are interface specific. Though, as one could see at IPOPT Linear solver, there are many parameters that can be changed within IPOPT, affecting the algorithm behavior. These parameters can still be controlled by placing an option file in the execution directory. Note that IPOPT's pdf documentation may provides more information than the previously mentioned online version for certain parameters. The in-script available parameters are:

- `lb`, `ub` : `real[int]` for lower and upper simple bounds upon the search variables must be of size n (search space dimension). If two components of the same index in these arrays are equal then the corresponding search variable is fixed. By default IPOPT will remove any fixed variable from the optimization process and always use the fixed value when calling functions. It can be changed using the `fixedvar` parameter.

- `clb`, `cub` : `real[int]` of size m (number of constraints) for lower and upper constraints bounds. Equality between two components of the same index i in `clb` and `cub` reflect an equality constraint.
- `structjacc` : To pass the greatest possible structure (indexes of non null coefficients) of the constraint Jacobians under the form `[I,J]` where `I` and `J` are two integer arrays. If not defined, the structure of the constraint Jacobians, evaluated in `Xi`, is used (no issue if the Jacobian is constant or always defined with the same `varf`, hazardous if it is with a triplet array or if a full matrix is involved).
- `structhess` : Same as above but for the Hessian function (unused if f is P2 or less and constraints are affine). Here again, keep in mind that it is the Hessian of the Lagrangian function (which is equal to the Hessian of f only if constraints are affine). If no structure is given with this parameter, the Lagrangian Hessian is evaluated on the starting point, with $\sigma = 1$ and $\lambda = (1, 1, \ldots, 1)$ (it is safe if all the constraints and fitness function Hessians are constant or build with `varf`, and here again it is less reliable if built with a triplet array or a full matrix).
- `checkindex` : A `bool` that triggers a dichotomic index search when matrices are copied from **FreeFEM** functions to IPOPT arrays. It is used to avoid wrong index matching when some null coefficients are removed from the matrices by **FreeFEM**. It will not solve the problems arising when a too small structure has been given at the initialization of the algorithm. Enabled by default (except in cases where all matrices are obviously constant).
- `warmstart` : If set to `true`, the constraints dual variables λ, and simple bound dual variables are initialized with the values of the arrays passed to `lm`, `lz` and `uz` named parameters (see below).
- `lm` : `real[int]` of size m, which is used to get the final values of the constraints dual variables λ and/or initialize them in case of a warm start (the passed array is also updated to the last dual variables values at the end of the algorithm).
- `lz`, `uz` : `real[int]` of size n to get the final values and/or initialize (in case of a warm start) the dual variables associated to simple bounds.
- `tol` : `real`, convergence tolerance for the algorithm, the default value is 10^{-8}.
- `maxiter` : `int`, maximum number of iterations with 3000 as default value.
- `maxcputime` : `real` value, maximum runtime duration. Default is 10^6 (almost 11 and a halfdays).
- `bfgs` : `bool` enabling or not the (low-storage) BFGS approximation of the Lagrangian Hessian. It is set to false by default, unless there is no way to compute the Hessian with the functions that have been passed to IPOPT.
- `derivativetest` : Used to perform a comparison of the derivatives given to IPOPT with finite differences computation. The possible `string` values are : `"none"` (default), `"first-order"`, `"second-order"` and `"only-second-order"`. The associated derivative error tolerance can be changed via the option file. One should not care about any error given by it before having tried, and failed, to perform a first optimization.
- `dth` : Perturbation parameter for the derivative test computations with finite differences. Set by default to 10^{-8}.
- `dttol` : Tolerance value for the derivative test error detection (default value unknown yet, maybe 10^{-5}).
- `optfile` : `string` parameter to specify the IPOPT option file name. IPOPT will look for a `ipopt.opt` file by default. Options set in the file will overwrite those defined in the **FreeFEM** script.
- `printlevel` : An `int` to control IPOPT output print level, set to 5 by default, the possible values are from 0 to 12. A description of the output information is available in the PDF documentation of IPOPT.
- `fixedvar` : `string` for the definition of simple bound equality constraints treatment : use `"make_parameter"` (default value) to simply remove them from the optimization process (the functions will always be evaluated with the fixed value for those variables), `"make_constraint"` to treat them as any other constraint or `"relax_bounds"` to relax fixing bound constraints.
- `mustrategy` : a `string` to choose the update strategy for the barrier parameter μ. The two possible tags are `"monotone"`, to use the monotone (Fiacco-McCormick) strategy, or `"adaptive"` (default setting).

- `muinit` : `real` positive value for the barrier parameter initialization. It is only relevant when `mustrategy` has been set to `monotone`.
- `pivtol` : `real` value to set the pivot tolerance for the linear solver. A smaller number pivots for sparsity, a larger number pivots for stability. The value has to be in the $[0, 1]$ interval and is set to 10^{-6} by default.
- `brf` : Bound relax factor: before starting the optimization, the bounds given by the user are relaxed. This option sets the factor for this relaxation. If it is set to zero, then the bound relaxation is disabled. This `real` has to be positive and its default value is 10^{-8}.
- `objvalue` : An identifier to a `real` type variable to get the last value of the objective function (best value in case of success).
- `mumin` : minimum value for the barrier parameter μ, a `real` with 10^{-11} as default value.
- `linesearch` : A boolean which disables the line search when set to `false`. The line search is activated by default. When disabled, the method becomes a standard Newton algorithm instead of a primal-dual system. The global convergence is then no longer assured, meaning that many initializers could lead to diverging iterates. But on the other hand, it can be useful when trying to catch a precise local minimum without having some out of control process making the iterate caught by some other near optimum.

3.5.4 Some short examples using IPOPT

Tip: Ipopt variational inequality A very simple example consisting of, given two functions f and g (defined on $\Omega \subset \mathbb{R}^2$), minimizing $J(u) = \frac{1}{2}\int_\Omega |\nabla u|^2 - \int_\Omega fu$, with $u \leq g$ almost everywhere:

```
// Solve
//- Delta u = f
//u < g
//u = 0 on Gamma
load "ff-Ipopt";

// Parameters
int nn = 20;
func f = 1.; //rhs function
real r = 0.03, s = 0.1;
func g = r - r/2*exp(-0.5*(square(x-0.5) + square(y-0.5))/square(s));

// Mesh
mesh Th = square(nn, nn);

// Fespace
fespace Vh(Th, P2);
Vh u = 0;
Vh lb = -1.e19;
Vh ub = g;

// Macro
macro Grad(u) [dx(u), dy(u)] //

// Problem
varf vP (u, v)
   = int2d(Th)(
        Grad(u)'*Grad(v)
   )
   - int2d(Th)(
```

(continues on next page)

(continued from previous page)

```
        f*v
    )
    ;
```

Here we build the matrix and second member associated to the function to fully and finally minimize it. The `[A,b]` syntax for the fitness function is then used to pass it to IPOPT.

```
matrix A = vP(Vh, Vh, solver=CG);
real[int] b = vP(0, Vh);
```

We use simple bounds to impose the boundary condition $u = 0$ on $\partial\Omega$, as well as the $u \leq g$ condition.

```
varf vGamma (u, v) = on(1, 2, 3, 4, u=1);
real[int] onGamma = vGamma(0, Vh);

//warning: the boundary conditions are given with lb and ub on border
ub[] = onGamma ? 0. : ub[];
lb[] = onGamma ? 0. : lb[];

// Solve
IPOPT([A, b], u[], lb=lb[], ub=ub[]);

// Plot
plot(u);
```

Tip: Ipopt variational inequality 2

Let Ω be a domain of $\mathbb{R}^2$. $f_1, f_2 \in L^2(\Omega)$ and $g_1, g_2 \in L^2(\partial\Omega)$ four given functions with $g_1 \leq g_2$ almost everywhere. We define the space:

$$V = \left\{(v_1, v_2) \in H^1(\Omega)^2; v_1|_{\partial\Omega} = g_1, v_2|_{\partial\Omega} = g_2, v_1 \leq v_2 \text{ a.e. } \right\}$$

as well as the function $J : H^1(\Omega)^2 \longrightarrow \mathbb{R}$:

$$J(v_1, v_2) = \frac{1}{2}\int_\Omega |\nabla v_1|^2 - \int_\Omega f_1 v_1 + \frac{1}{2}\int_\Omega |\nabla v_2|^2 - \int_\Omega f_2 v_2$$

The problem entails finding (numerically) two functions $(u_1, u_2) = \underset{(v_1,v_2)\in V}{\operatorname{argmin}} J(v_1, v_2)$.

```
load "ff-Ipopt";

// Parameters
int nn = 10;
func f1 = 10;//right hand side
func f2 = -15;
func g1 = -0.1;//Boundary condition functions
func g2 = 0.1;

// Mesh
mesh Th = square(nn, nn);

// Fespace
fespace Vh(Th, [P1, P1]);
Vh [uz, uz2] = [1, 1];
```

(continues on next page)

(continued from previous page)

```
Vh [lz, lz2] = [1, 1];
Vh [u1, u2] = [0, 0]; //starting point

fespace Wh(Th, [P1]);
Wh lm=1.;

// Macro
macro Grad(u) [dx(u), dy(u)] //

// Loop
int iter=0;
while (++iter){
   // Problem
   varf vP ([u1, u2], [v1, v2])
      = int2d(Th)(
           Grad(u1)'*Grad(v1)
         + Grad(u2)'*Grad(v2)
      )
      - int2d(Th)(
           f1*v1
         + f2*v2
      )
      ;

   matrix A = vP(Vh, Vh); //fitness function matrix
   real[int] b = vP(0, Vh); //and linear form

   int[int] II1 = [0], II2 = [1];//Constraints matrix
   matrix C1 = interpolate (Wh, Vh, U2Vc=II1);
   matrix C2 = interpolate (Wh, Vh, U2Vc=II2);
   matrix CC = -1*C1 + C2; // u2 - u1 > 0
   Wh cl = 0; //constraints lower bounds (no upper bounds)

   //Boundary conditions
   varf vGamma ([u1, u2], [v1, v2]) = on(1, 2, 3, 4, u1=1, u2=1);
   real[int] onGamma = vGamma(0, Vh);
   Vh [ub1, ub2] = [g1, g2];
   Vh [lb1, lb2] = [g1, g2];
   ub1[] = onGamma ? ub1[] : 1e19; //Unbounded in interior
   lb1[] = onGamma ? lb1[] : -1e19;

   Vh [uzi, uzi2] = [uz, uz2], [lzi, lzi2] = [lz, lz2];
   Wh lmi = lm;
   Vh [ui1, ui2] = [u1, u2];

   // Solve
   IPOPT([b, A], CC, ui1[], lb=lb1[], clb=cl[], ub=ub1[], warmstart=iter>1, uz=uzi[],
↪lz=lzi[], lm=lmi[]);

   // Plot
   plot(ui1, ui2, wait=true, nbiso=60, dim=3);

   if(iter > 1) break;

   // Mesh adpatation
   Th = adaptmesh(Th, [ui1, ui2], err=0.004, nbvx=100000);
   [uz, uz2] = [uzi, uzi2];
```

(continues on next page)

(a) Numerical Approximation of the Variational Inequality

(b) Numerical Approximation of the Variational Inequality

Fig. 3.44: Variational inequality

(continued from previous page)

```
    [lz, lz2] = [lzi, lzi2];
    [u1, u2] = [ui1, ui2];
    lm = lmi;
}
```

3.5.5 3D constrained minimum surface with IPOPT

Area and volume expressions

This example is aimed at numerically solving some constrained minimum surface problems with the IPOPT algorithm. We restrain to C^k ($k \geq 1$), closed, spherically parametrizable surfaces, i.e. surfaces S such that:

$$\exists \rho \in C^k([0,2\pi] \times [0,\pi]) | S = \left\{ X = \begin{pmatrix} \rho(\theta,\phi) \\ 0 \\ 0 \end{pmatrix}, (\theta,\phi) \in [0,2\pi] \times [0,\pi] \right\}$$

Where the components are expressed in the spherical coordinate system. Let's call Ω the $[0,2\pi] \times [0,\pi]$ angular parameters set. In order to exclude self crossing and opened shapes, the following assumptions upon ρ are made:

$$\rho \geq 0 \text{ and } \forall \phi, \rho(0,\phi) = \rho(2\pi,\phi)$$

For a given function ρ the first fundamental form (the metric) of the defined surface has the following matrix representation:

$$G = \begin{pmatrix} \rho^2 \sin^2(\phi) + (\partial_\theta \rho)^2 & \partial_\theta \rho \partial_\phi \rho \\ \partial_\theta \rho \partial_\phi \rho & \rho^2 + (\partial_\phi \rho)^2 \end{pmatrix} \tag{3.23}$$

This metric is used to express the area of the surface. Let $g = \det(G)$, then we have:

$$\begin{aligned} \mathcal{A}(\rho) &= \int \Omega \|\partial_\theta X \wedge \partial_\phi X\| = \int \Omega \sqrt{g} \\ &= \int \Omega \sqrt{\rho^2 (\partial_\theta \rho)^2 + \rho^4 \sin^2(\phi) + \rho^2 (\partial_\phi \rho)^2 \sin^2(\phi)} d\theta d\phi \end{aligned} \tag{3.24}$$

The volume of the space enclosed within the shape is easier to express:

$$\mathcal{V}(\rho) = \int \Omega \int_0^{\rho(\theta,\phi)} r^2 \sin(\phi) dr d\theta d\phi = \frac{1}{3} \int \Omega \rho^3 \sin(\phi) d\theta d\phi \tag{3.25}$$

Derivatives

In order to use a Newton based interior point optimization algorithm, one must be able to evaluate the derivatives of $\mathcal{A}$ and $\mathcal{V}$ with respect to rho. Concerning the area, we have the following result:

$$\forall v \in C^1(\Omega)\ ,\ \langle d\mathcal{A}(\rho), v\rangle = \int \Omega \frac{1}{2}\frac{d\bar{g}(\rho)(v)}{\sqrt{g}} d\theta d\phi$$

Where $\bar{g}$ is the application mapping the $(\theta,\phi) \mapsto g(\theta,\phi)$ scalar field to ρ. This leads to the following expression, easy to transpose in a freefem script using:

$$\begin{array}{rcl} \forall v \in C^1(\Omega) & & \\ \langle d\mathcal{A}(\rho), v\rangle & = & \int \Omega\big(2\rho^3\sin^2(\phi) + \rho(\partial_\theta\rho)^2 + \rho(\partial_\phi\rho)^2\sin^2(\phi)\big)\, v \\ & & + \int \Omega\ \rho^2\partial_\theta\rho\partial_\theta v\ +\ \rho^2\partial_\phi\rho\sin^2(\phi)\partial_\phi v \end{array} \tag{3.26}$$

With a similar approach, one can derive an expression for second order derivatives. However, comporting no specific difficulties, the detailed calculus are tedious, the result is that these derivatives can be written using a 3×3 matrix $\mathbf{B}$ whose coefficients are expressed in term of ρ and its derivatives with respect to θ and ϕ, such that:

$$\forall (w,v) \in C^1(\Omega)\ ,\ d^2\mathcal{A}(\rho)(w,v) = \int \Omega\begin{pmatrix} w & \partial_\theta w & \partial_\phi w \end{pmatrix}\mathbf{B}\begin{pmatrix} v \\ \partial_\theta v \\ \partial_\phi v \end{pmatrix} d\theta d\phi \tag{3.27}$$

Deriving the volume function derivatives is again an easier task. We immediately get the following expressions:

$$\begin{array}{rcl} \forall v\ ,\ \langle d\mathcal{V}(\rho), v\rangle & = & \int \Omega\rho^2\sin(\phi) v\ d\theta d\phi \\ \forall w, v\ , d^2\mathcal{V}(\rho)(w,v) & = & \int \Omega 2\rho\sin(\phi) wv\ d\theta d\phi \end{array} \tag{3.28}$$

The problem and its script

The whole code is available in *IPOPT minimal surface & volume example*. We propose to solve the following problem:

Tip: Given a positive function ρ_{object} piecewise continuous, and a scalar $\mathcal{V}_{\max} > \mathcal{V}(\rho_{\text{object}})$, find ρ_0 such that:

$$\rho_0 = \underset{\rho\in C^1(\Omega)}{\text{argmin}}\ \mathcal{A}(\rho)\ ,\ \text{s.t. } \rho_0 \geq \rho_{\text{object}} \text{ and } \mathcal{V}(\rho_0) \leq \mathcal{V}_{\max}$$

If ρ_{object} is the spherical parametrization of the surface of a 3-dimensional object (domain) $\mathcal{O}$, it can be interpreted as finding the surface with minimum area enclosing the object with a given maximum volume. If $\mathcal{V}_{\max}$ is close to $\mathcal{V}(\rho_{\text{object}})$, so should be ρ_0 and ρ_{object}. With higher values of $\mathcal{V}_{\max}$, ρ should be closer to the unconstrained minimum surface surrounding $\mathcal{O}$ which is obtained as soon as $\mathcal{V}_{\max} \geq \frac{4}{3}\pi\|\rho_{\text{object}}\|_\infty^3$ (sufficient but not necessary).

It also could be interesting to solve the same problem with the constraint $\mathcal{V}(\rho_0) \geq \mathcal{V}_{\min}$ which leads to a sphere when $\mathcal{V}_{\min} \geq \frac{1}{6}\pi\text{diam}(\mathcal{O})^3$ and moves toward the solution of the unconstrained problem as $\mathcal{V}_{\min}$ decreases.

We start by meshing the domain $[0,2\pi]\times\ [0,\pi]$, then a periodic P1 finite elements space is defined.

```
load "msh3";
load "medit";
load "ff-Ipopt";

// Parameters
int nadapt = 3;
real alpha = 0.9;
int np = 30;
real regtest;
```

(continues on next page)

(continued from previous page)

```
int shapeswitch = 1;
real sigma = 2*pi/40.;
real treshold = 0.1;
real e = 0.1;
real r0 = 0.25;
real rr = 2-r0;
real E = 1./(e*e);
real RR = 1./(rr*rr);

// Mesh
mesh Th = square(2*np, np, [2*pi*x, pi*y]);

// Fespace
fespace Vh(Th, P1, periodic=[[2, y], [4, y]]);
//Initial shape definition
//outside of the mesh adaptation loop to initialize with the previous optimial shape found on further iterations
Vh startshape = 5;
```

We create some finite element functions whose underlying arrays will be used to store the values of dual variables associated to all the constraints in order to reinitialize the algorithm with it in the case where we use mesh adaptation. Doing so, the algorithm will almost restart at the accuracy level it reached before mesh adaptation, thus saving many iterations.

```
Vh uz = 1., lz = 1.;
rreal[int] lm = [1];
```

Then, follows the mesh adaptation loop, and a rendering function, `Plot3D`, using 3D mesh to display the shape it is passed with `medit` (the `movemesh23` procedure often crashes when called with ragged shapes).

```
for(int kkk = 0; kkk < nadapt; ++kkk){
   int iter=0;
   func sin2 = square(sin(y));

   // A function which transform Th in 3d mesh (r=rho)
   //a point (theta,phi) of Th becomes ( r(theta,phi)*cos(theta)*sin(phi) , r(theta,phi)*sin(theta)*sin(phi) , r(theta,phi)*cos(phi) )
   //then displays the resulting mesh with medit
   func int Plot3D (real[int] &rho, string cmm, bool ffplot){
      Vh rhoo;
      rhoo[] = rho;
      //mesh sTh = square(np, np/2, [2*pi*x, pi*y]);
      //fespace sVh(sTh, P1);
      //Vh rhoplot = rhoo;
      try{
         mesh3 Sphere = movemesh23(Th, transfo=[rhoo(x,y)*cos(x)*sin(y), rhoo(x,y)*sin(x)*sin(y), rhoo(x,y)*cos(y)]);
         if(ffplot)
            plot(Sphere);
         else
            medit(cmm, Sphere);
      }
      catch(...){
         cout << "PLOT ERROR" << endl;
      }
      return 1;
```

(continues on next page)

(continued from previous page)

```
    }
}
```

Here are the functions related to the area computation and its shape derivative, according to equations (3.24) and (3.26):

```
// Surface computation
//Maybe is it possible to use movemesh23 to have the surface function less complicated
//However, it would not simplify the gradient and the hessian
func real Area (real[int] &X){
   Vh rho;
   rho[] = X;
   Vh rho2 = square(rho);
   Vh rho4 = square(rho2);
   real res = int2d(Th)(sqrt(rho4*sin2 + rho2*square(dx(rho)) +␣
↪rho2*sin2*square(dy(rho))));
   ++iter;
   if(1)
      plot(rho, value=true, fill=true, cmm="rho(theta,phi) on [0,2pi]x[0,pi] - S=
↪"+res, dim=3);
   else
      Plot3D(rho[], "shape_evolution", 1);
   return res;
}

func real[int] GradArea (real[int] &X){
   Vh rho, rho2;
   rho[] = X;
   rho2[] = square(X);
   Vh sqrtPsi, alpha;
   {
      Vh dxrho2 = dx(rho)*dx(rho), dyrho2 = dy(rho)*dy(rho);
      sqrtPsi = sqrt(rho2*rho2*sin2 + rho2*dxrho2 + rho2*dyrho2*sin2);
      alpha = 2.*rho2*rho*sin2 + rho*dxrho2 + rho*dyrho2*sin2;
   }
   varf dArea (u, v)
      = int2d(Th)(
           1./sqrtPsi * (alpha*v + rho2*dx(rho)*dx(v) + rho2*dy(rho)*sin2*dy(v))
      )
      ;

   real[int] grad = dArea(0, Vh);
   return grad;
}
```

The function returning the hessian of the area for a given shape is a bit blurry, thus we won't show here all of equation (3.27) coefficients definition, they can be found in the `edp` file.

```
matrix hessianA;
func matrix HessianArea (real[int] &X){
   Vh rho, rho2;
   rho[] = X;
   rho2 = square(rho);
   Vh sqrtPsi, sqrtPsi3, C00, C01, C02, C11, C12, C22, A;
   {
      Vh C0, C1, C2;
```

(continues on next page)

(continued from previous page)

```
      Vh dxrho2 = dx(rho)*dx(rho), dyrho2 = dy(rho)*dy(rho);
      sqrtPsi = sqrt( rho2*rho2*sin2 + rho2*dxrho2 + rho2*dyrho2*sin2);
      sqrtPsi3 = (rho2*rho2*sin2 + rho2*dxrho2 + rho2*dyrho2*sin2)*sqrtPsi;
      C0 = 2*rho2*rho*sin2 + rho*dxrho2 + rho*dyrho2*sin2;
      C1 = rho2*dx(rho);
      C2 = rho2*sin2*dy(rho);
      C00 = square(C0);
      C01 = C0*C1;
      C02 = C0*C2;
      C11 = square(C1);
      C12 = C1*C2;
      C22 = square(C2);
      A = 6.*rho2*sin2 + dxrho2 + dyrho2*sin2;
   }
   varf d2Area (w, v)
      =int2d(Th)(
           1./sqrtPsi * (
              A*w*v
            + 2*rho*dx(rho)*dx(w)*v
            + 2*rho*dx(rho)*w*dx(v)
            + 2*rho*dy(rho)*sin2*dy(w)*v
            + 2*rho*dy(rho)*sin2*w*dy(v)
            + rho2*dx(w)*dx(v)
            + rho2*sin2*dy(w)*dy(v)
         )
         + 1./sqrtPsi3 * (
              C00*w*v
            + C01*dx(w)*v
            + C01*w*dx(v)
            + C02*dy(w)*v
            + C02*w*dy(v)
            + C11*dx(w)*dx(v)
            + C12*dx(w)*dy(v)
            + C12*dy(w)*dx(v)
            + C22*dy(w)*dy(v)
         )
      )
      ;
   hessianA = d2Area(Vh, Vh);
   return hessianA;
}
```

And the volume related functions:

```
// Volume computation
func real Volume (real[int] &X){
   Vh rho;
   rho[] = X;
   Vh rho3 = rho*rho*rho;
   real res = 1./3.*int2d(Th)(rho3*sin(y));
   return res;
}

func real[int] GradVolume (real[int] &X){
   Vh rho;
   rho[] = X;
   varf dVolume(u, v) = int2d(Th)(rho*rho*sin(y)*v);
```

(continues on next page)

(continued from previous page)

```
   real[int] grad = dVolume(0, Vh);
   return grad;
}

matrix hessianV;
func matrix HessianVolume(real[int] &X){
   Vh rho;
   rho[] = X;
   varf d2Volume(w, v) = int2d(Th)(2*rho*sin(y)*v*w);
   hessianV = d2Volume(Vh, Vh);
   return hessianV;
}
```

If we want to use the volume as a constraint function we must wrap it and its derivatives in some **FreeFEM** functions returning the appropriate types. It is not done in the above functions in cases where one wants to use it as a fitness function. The lagrangian hessian also has to be wrapped since the Volume is not linear with respect to ρ, it has some non-null second order derivatives.

```
func real[int] ipVolume (real[int] &X){ real[int] vol = [Volume(X)]; return vol; }
matrix mdV;
func matrix ipGradVolume (real[int] &X) { real[int,int] dvol(1,Vh.ndof); dvol(0,:) =␣
↪GradVolume(X); mdV = dvol; return mdV; }
matrix HLagrangian;
func matrix ipHessianLag (real[int] &X, real objfact, real[int] &lambda){
   HLagrangian = objfact*HessianArea(X) + lambda[0]*HessianVolume(X);
   return HLagrangian;
}
```

The `ipGradVolume` function could pose some troubles during the optimization process because the gradient vector is transformed in a sparse matrix, so any null coefficient will be discarded. Here we create the IPOPT structure manually and use the `checkindex` named-parameter to avoid bad indexing during copies. This gradient is actually dense, there is no reason for some components to be constantly zero:

```
int[int] gvi(Vh.ndof), gvj=0:Vh.ndof-1;
gvi = 0;
```

These two arrays will be passed to IPOPT with `structjacc=[gvi,gvj]`. The last remaining things are the bound definitions. The simple lower bound must be equal to the components of the P1 projection of ρ_{object}. And we choose $\alpha \in [0,1]$ to set $\mathcal{V}_{\max}$ to $(1-\alpha)\mathcal{V}(\rho_{object}) + \alpha\frac{4}{3}\pi\|\rho_{\text{object}}\|_\infty^3$:

```
func disc1 = sqrt(1./(RR+(E-RR)*cos(y)*cos(y)))*(1+0.1*cos(7*x));
func disc2 = sqrt(1./(RR+(E-RR)*cos(x)*cos(x)*sin2));

if(1){
   lb = r0;
   for (int q = 0; q < 5; ++q){
      func f = rr*Gaussian(x, y, 2*q*pi/5., pi/3.);
      func g = rr*Gaussian(x, y, 2*q*pi/5.+pi/5., 2.*pi/3.);
      lb = max(max(lb, f), g);
   }
   lb = max(lb, rr*Gaussian(x, y, 2*pi, pi/3));
}
lb = max(lb, max(disc1, disc2));
real Vobj = Volume(lb[]);
real Vnvc = 4./3.*pi*pow(lb[].linfty,3);

```

(continues on next page)

(continued from previous page)

```
if(1)
   Plot3D(lb[], "object_inside", 1);
real[int] clb = 0., cub = [(1-alpha)*Vobj + alpha*Vnvc];
```

Calling IPOPT:

```
int res = IPOPT(Area, GradArea, ipHessianLag, ipVolume, ipGradVolume,
   rc[], ub=ub[], lb=lb[], clb=clb, cub=cub, checkindex=1, maxiter=kkk<nadapt-1 ?␣
↪40:150,
   warmstart=kkk, lm=lm, uz=uz[], lz=lz[], tol=0.00001, structjacc=[gvi,gvj]);
cout << "IPOPT: res =" << res << endl ;

// Plot
Plot3D(rc[], "Shape_at_"+kkk, 1);
Plot3D(GradArea(rc[]), "ShapeGradient", 1);
```

Finally, before closing the mesh adaptation loop, we have to perform the said adaptation. The mesh is adaptated with respect to the $X = (\rho, 0, 0)$ (in spherical coordinates) vector field, not directly with respect to ρ, otherwise the true curvature of the 3D-shape would not be well taken into account.

```
if (kkk < nadapt-1){
   Th = adaptmesh(Th, rc*cos(x)*sin(y), rc*sin(x)*sin(y), rc*cos(y),
      nbvx=50000, periodic=[[2, y], [4, y]]);
   plot(Th, wait=true);
   startshape = rc;
   uz = uz;
   lz = lz;
}
```

Here are some pictures of the resulting surfaces obtained for decreasing values of α (and a slightly more complicated object than two orthogonal discs). We return to the enclosed object when $\alpha = 0$:

3.5.6 The nlOpt optimizers

The `ff-NLopt` package provides a **FreeFEM** interface to the free/open-source library for nonlinear optimization, easing the use of several different free optimization (constrained or not) routines available online along with the PDE solver. All the algorithms are well documented in NLopt documentation, therefore no exhaustive information concerning their mathematical specificities will be found here and we will focus on the way they are used in a **FreeFEM** script. If needing detailed information about these algorithms, visit the website where a description of each of them is given, as well as many bibliographical links.

Most of the gradient based algorithms of NLopt uses a full matrix approximation of the Hessian, so if you're planning to solve a large scale problem, use the IPOPT optimizer which definitely surpass them.

All the NLopt features are identified that way:

```
load "ff-NLopt"
//define J, u, and maybe grad(J), some constraints etc...
real min = nloptXXXXXX(J, u, //Unavoidable part
   grad=<name of grad(J)>, //if needed
   lb= //Lower bounds array
   ub= //Upper bounds array
   ... //Some optional arguments:
   //Constraints functions names,
```

(continues on next page)

Medit - [Shape_at_2] #1
$0.4 \leq \alpha \leq 1$
$\alpha = 0.3$
$\alpha = 0.2$
$\alpha = 0.1$
$\alpha = 0.05$
$\alpha = 0$

(continued from previous page)

```
    //Stopping criteria,
    //Algorithm specific parameters,
    //Etc...
);
```

XXXXXX refers to the algorithm tag (not necessarily 6 characters long). u is the starting position (a real[int] type array) which will be overwritten by the algorithm, the value at the end being the found $argmin$. And as usual, J is a function taking a real[int] type array as argument and returning a real. grad, lb and ub are "half-optional" arguments, in the sense that they are obligatory for some routines but not all.

The possible optionally named parameters are the following, note that they are not used by all algorithms (some do not support constraints, or a type of constraints, some are gradient-based and others are derivative free, etc...). One can refer to the table after the parameters description to check which are the named parameters supported by a specific algorithm. Using an unsupported parameter will not stop the compiler work, seldom breaks runtime, and will just be ignored. When it is obvious you are missing a routine, you will get a warning message at runtime (for example if you pass a gradient to a derivative free algorithm, or set the population of a non-genetic one, etc...). In the following description, n stands for the dimension of the search space.

Half-optional parameters :

- grad= The name of the function which computes the gradient of the cost function (prototype should be real[int] $\rightarrow$ real[int], both argument and result should have the size n). This is needed as soon as a gradient-based method is involved, which is ignored if defined in a derivative free context.
- lb/ub = Lower and upper bounds arrays (real[int] type) of size n. Used to define the bounds within which the search variable is allowed to move. Needed for some algorithms, optional, or unsupported for others.
- subOpt : Only enabled for the Augmented Lagrangian and MLSL methods who need a sub-optimizer in order to work. Just pass the tag of the desired local algorithm with a string.

Constraints related parameters (optional - unused if not specified):

- IConst/EConst : Allows to pass the name of a function implementing some inequality (resp. equality) constraints on the search space. The function type must be real[int] $\rightarrow$ real[int] where the size of the returned array is equal to the number of constraints (of the same type - it means that all of the constraints are computed in one vectorial function). In order to mix inequality and equality constraints in a same minimization attempt, two vectorial functions have to be defined and passed. See example (3.29) for more details about how these constraints have to be implemented.
- gradIConst/gradEConst : Use to provide the inequality (resp. equality) constraints gradient. These are real[int] $\rightarrow$ real[int,int] type functions. Assuming we have defined a constraint function (either inequality or equality) with p constraints, the size of the matrix returned by its associated gradient must be $p \times n$ (the i-th line of the matrix is the gradient of the i-th constraint). It is needed in a gradient-based context as soon as an inequality or equality constraint function is passed to the optimizer and ignored in all other cases.
- tolIConst/tolEConst : Tolerance values for each constraint. This is an array of size equal to the number of inequality (resp. equality) constraints. Default value is set to 10^{-12} for each constraint of any type.

Stopping criteria :

- stopFuncValue : Makes the algorithm end when the objective function reaches this real value.
- stopRelXTol : Stops the algorithm when the relative moves in each direction of the search space is smaller than this real value.
- stopAbsXTol : Stops the algorithm when the moves in each direction of the search space is smaller than the corresponding value in this real[int] array.
- stopRelFTol : Stops the algorithm when the relative variation of the objective function is smaller than this real value.

- `stopAbsFTol` : Stops the algorithm when the variation of the objective function is smaller than this `real` value.
- `stopMaxFEval` : Stops the algorithm when the number of fitness evaluations reaches this `integer` value.
- `stopTime` : Stops the algorithm when the optimization time in seconds exceeds this `real` value. This is not a strict maximum: the time may exceed it slightly, depending upon the algorithm and on how slow your function evaluation is.

  Note that when an AUGLAG or MLSL method is used, the meta-algorithm and the sub-algorithm may have different termination criteria. Thus, for algorithms of this kind, the following named parameters has been defined (just adding the SO prefix - for Sub-Optimizer) to set the ending condition of the sub-algorithm (the meta one uses the ones above): `SOStopFuncValue`, `SOStopRelXTol`, and so on... If these are not used, the sub-optimizer will use those of the master routine.

Other named parameters :

- `popSize` : `integer` used to change the size of the sample for stochastic search methods. Default value is a peculiar heuristic to the chosen algorithm.
- `SOPopSize` : Same as above, but when the stochastic search is passed to a meta-algorithm.
- `nGradStored` : The number (`integer` type) of gradients to "remember" from previous optimization steps: increasing this increases the memory requirements but may speed convergence. It is set to a heuristic value by default. If used with AUGLAG or MLSL, it will only affect the given subsidiary algorithm.

The following table sums up the main characteristics of each algorithm, providing the more important information about which features are supported by which algorithm and what are the unavoidable arguments they need. More details can be found in NLopt documentation.

Tip: Variational inequality

Let Ω be a domain of $\mathbb{R}^2$, $f_1, f_2 \in L^2(\Omega)$ and $g_1, g_2 \in L^2(\partial\Omega)$ four given functions with $g_1 \leq g_2$ almost everywhere.

We define the space:

$$V = \left\{(v_1, v_2) \in H^1(\Omega)^2; v_1|_{\partial\Omega} = g_1, v_2|_{\partial\Omega} = g_2, v_1 \leq v_2 \text{ a.e. } \right\}$$

as well as the function $J : H^1(\Omega)^2 \longrightarrow \mathbb{R}$:

$$J(v_1, v_2) = \frac{1}{2}\int_\Omega |\nabla v_1|^2 - \int_\Omega f_1 v_1 + \frac{1}{2}\int_\Omega |\nabla v_2|^2 - \int_\Omega f_2 v_2 \tag{3.29}$$

The problem consists in finding (numerically) two functions $(u_1, u_2) = \underset{(v_1,v_2)\in V}{\operatorname{argmin}} J(v_1, v_2)$.

This can be interpreted as finding u_1, u_2 as close as possible (in a certain sense) to the solutions of the Laplace equation with respectively f_1, f_2 second members and g_1, g_2 Dirichlet boundary conditions with the $u_1 \leq u_2$ almost everywhere constraint.

Here is the corresponding script to treat this variational inequality problem with one of the NLOpt algorithms.

```
//A brief script to demonstrate how to use the freefemm interfaced nlopt routines
//The problem consist in solving a simple variational inequality using one of the
//optimization algorithm of nlopt. We restart the algorithlm a few times after
//performing some mesh adaptation to get a more precise output

load "ff-NLopt"

// Parameters
int kas = 3; //choose of the algorithm
```

(continues on next page)

Id Tag	Full Name	Bounds	Gradient -Based	Stochastic	Constraints		Sub-Opt
					Equality	Inequality	
DIRECT	Dividing rectangles	●					
DIRECTL	Locally biased dividing rectangles	●					
DIRECTLRand	Randomized locally biased dividing rectangles	●					
DIRECTNoScal	Dividing rectangles - no scaling	●					
DIRECTLNoScal	Locally biased dividing rectangles - no scaling	●					
DIRECTLRandNoScal	Randomized locally biased dividing rectangles - no scaling	●					
OrigDIRECT	Original Glabonsky's dividing rectangles	●				✔	
OrigDIRECTL	Original Glabonsky's locally biased dividing rectangles	●				✔	
StoGO	Stochastic(?) Global Optimization	●	●				
StoGORand	Randomized Stochastic(?) Global Optimization	●	●				
LBFGS	Low-storage BFGS		●				
PRAXIS	Principal AXIS	✔					
Var1	Rank-1 shifted limited-memory variable-metric		●				
Var2	Rank-2 shifted limited-memory variable-metric		●				
TNewton	Truncated Newton		●				
TNewtonRestart	Steepest descent restarting truncated Newton		●				
TNewtonPrecond	BFGS preconditionned truncated Newton		●				
TNewtonRestartPrecond	BFGS preconditionned truncated Newton with steepest descent restarting		●				
CRS2	Controled random search with local mutation	✔		●			
MMA	Method of moving asymptots	✔	●			✔	
COBYLA	Constrained optimization by linear approximations	✔			✔	✔	
NEWUOA	NEWUOA						
NEWUOABound	NEWUOA for bounded optimization	✔					
NelderMead	Nelder-Mead simplex	✔					
Sbplx	Subplex	✔					
BOBYQA	BOBYQA	✔					
ISRES	Improved stochastic ranking evolution strategy	✔		●	✔	✔	
SLSQP	Sequential least-square quadratic programming	✔	●		✔	✔	
MLSL	Multi-level single-linkage	✔	●	●			●
MLSLLDS	Low discrepancy multi-level single-linkage	✔	●	●			●
AUGLAG	Constraints augmented lagrangian	✔	●		✔	✔	●
AUGLAGEQ	Equality constraints augmented lagrangian	✔	●		✔	✔	●

Legend :

- ✔ Supported and optional
- ✔ Should be supported and optional, may lead to weird behaviour though.
- ● Intrinsic characteristic of the algorithm which then need one or more unavoidable parameter to work (for stochastic algorithm, the population size always have a default value, they will then work if it is ommited)
- ●/✔ For routines with subsidiary algorithms only, indicates that the corresponding feature will depend on the chosen sub-optimizer.

(continued from previous page)

```
int NN = 10;
func f1 = 1.;
func f2 = -1.;
func g1 = 0.;
func g2 = 0.1;
int iter = 0;
int nadapt = 2;
real starttol = 1e-6;
real bctol = 6.e-12;

// Mesh
mesh Th = square(NN, NN);

// Fespace
fespace Vh(Th, P1);
Vh oldu1, oldu2;

// Adaptation loop
for (int al = 0; al < nadapt; ++al){
   varf BVF (v, w) = int2d(Th)(0.5*dx(v)*dx(w) + 0.5*dy(v)*dy(w));
   varf LVF1 (v, w) = int2d(Th)(f1*w);
   varf LVF2 (v, w) = int2d(Th)(f2*w);
   matrix A = BVF(Vh, Vh);
   real[int] b1 = LVF1(0, Vh), b2 = LVF2(0, Vh);

   varf Vbord (v, w) = on(1, 2, 3, 4, v=1);

   Vh In, Bord;
   Bord[] = Vbord(0, Vh, tgv=1);
   In[] = Bord[] ? 0:1;
   Vh gh1 = Bord*g1, gh2 = Bord*g2;

   func real J (real[int] &X){
      Vh u1, u2;
      u1[] = X(0:Vh.ndof-1);
      u2[] = X(Vh.ndof:2*Vh.ndof-1);
      iter++;
      real[int] Au1 = A*u1[], Au2 = A*u2[];
      Au1 -= b1;
      Au2 -= b2;
      real val = u1[]'*Au1 + u2[]'*Au2;
      if (iter%10 == 9)
         plot(u1, u2, nbiso=30, fill=1, dim=3, cmm="adapt level "+al+" - iteration
↪"+iter+" - J = "+val, value=1);
      return val;
   }

   varf dBFV (v, w) = int2d(Th)(dx(v)*dx(w)+dy(v)*dy(w));
   matrix dA = dBFV(Vh, Vh);
   func real[int] dJ (real[int] &X){
      Vh u1, u2;
      u1[] = X(0:Vh.ndof-1);
      u2[] = X(Vh.ndof:2*Vh.ndof-1);

      real[int] grad1 = dA*u1[], grad2 = dA*u2[];
      grad1 -= b1;
      grad2 -= b2;
```

(continues on next page)

(continued from previous page)

```
    real[int] Grad(X.n);
    Grad(0:Vh.ndof-1) = grad1;
    Grad(Vh.ndof:2*Vh.ndof-1) = grad2;
    return Grad;
}

func real[int] IneqC (real[int] &X){
    real[int] constraints(Vh.ndof);
    for (int i = 0; i < Vh.ndof; ++i) constraints[i] = X[i] - X[i+Vh.ndof];
    return constraints;
}

func real[int,int] dIneqC (real[int] &X){
    real[int, int] dconst(Vh.ndof, 2*Vh.ndof);
    dconst = 0;
    for(int i = 0; i < Vh.ndof; ++i){
        dconst(i, i) = 1.;
        dconst(i, i+Vh.ndof) = -1.;
    }
    return dconst;
}

real[int] BordIndex(Th.nbe); //Indexes of border d.f.
{
    int k = 0;
    for (int i = 0; i < Bord.n; ++i) if (Bord[][i]){ BordIndex[k] = i; ++k; }
}

func real[int] BC (real[int] &X){
    real[int] bc(2*Th.nbe);
    for (int i = 0; i < Th.nbe; ++i){
        int I = BordIndex[i];
        bc[i] = X[I] - gh1[][I];
        bc[i+Th.nbe] = X[I+Th.nv] - gh2[][I];
    }
    return bc;
}

func real[int, int] dBC(real[int] &X){
    real[int, int] dbc(2*Th.nbe,2*Th.nv);
    dbc = 0.;
    for (int i = 0; i < Th.nbe; ++i){
        int I = BordIndex[i];
        dbc(i, I) = 1.;
        dbc(i+Th.nbe, I+Th.nv) = 1.;
    }
    return dbc;
}

real[int] start(2*Vh.ndof), up(2*Vh.ndof), lo(2*Vh.ndof);

if (al == 0){
    start(0:Vh.ndof-1) = 0.;
    start(Vh.ndof:2*Vh.ndof-1) = 0.01;
}
else{
    start(0:Vh.ndof-1) = oldu1[];
```

(continues on next page)

(continued from previous page)

```
      start(Vh.ndof:2*Vh.ndof-1) = oldu2[];
   }

   up = 1000000;
   lo = -1000000;
   for (int i = 0; i < Vh.ndof; ++i){
      if (Bord[][i]){
         up[i] = gh1[][i] + bctol;
         lo[i] = gh1[][i] - bctol;
         up[i+Vh.ndof] = gh2[][i] + bctol;
         lo[i+Vh.ndof] = gh2[][i] - bctol;
      }
   }

   real mini = 1e100;
   if (kas == 1)
      mini = nloptAUGLAG(J, start, grad=dJ, lb=lo,
         ub=up, IConst=IneqC, gradIConst=dIneqC,
         subOpt="LBFGS", stopMaxFEval=10000, stopAbsFTol=starttol);
   else if (kas == 2)
      mini = nloptMMA(J, start, grad=dJ, lb=lo, ub=up, stopMaxFEval=10000, stopAbsFTol=starttol);
   else if (kas == 3)
      mini = nloptAUGLAG(J, start, grad=dJ, IConst=IneqC,
         gradIConst=dIneqC, EConst=BC, gradEConst=dBC,
         subOpt="LBFGS", stopMaxFEval=200, stopRelXTol=1e-2);
   else if (kas == 4)
      mini = nloptSLSQP(J, start, grad=dJ, IConst=IneqC,
         gradIConst=dIneqC, EConst=BC, gradEConst=dBC,
         stopMaxFEval=10000, stopAbsFTol=starttol);
   Vh best1, best2;
   best1[] = start(0:Vh.ndof-1);
   best2[] = start(Vh.ndof:2*Vh.ndof-1);

   Th = adaptmesh(Th, best1, best2);
   oldu1 = best1;
   oldu2 = best2;
}
```

3.5.7 Optimization with MPI

The only quick way to use the previously presented algorithms on a parallel architecture lies in parallelizing the used cost function (which is in most real life cases, the expensive part of the algorithm). Somehow, we provide a parallel version of the CMA-ES algorithm. The parallelization principle is the trivial one of evolving/genetic algorithms: at each iteration the cost function has to be evaluated N times without any dependence at all, these N calculus are then equally distributed to each process. Calling the MPI version of CMA-ES is nearly the same as calling its sequential version (a complete example of use can be found in the *CMAES MPI variational inequality example*):

```
load "mpi-cmaes"
... // Define J, u and all here
real min = cmaesMPI(J, u, stopTolFun=1e-6, stopMaxIter=3000);
cout << "minimum value is " << min << " for u = " << u << endl;
```

If the population size is not changed using the `popsize` parameter, it will use the heuristic value slightly changed to be equal to the closest greatest multiple of the size of the communicator used by the optimizer. The **FreeFEM** `mpicommworld` is used by default. The user can specify his own MPI communicator with the named parameter `comm=`, see the MPI section of this manual for more information about communicators in **FreeFEM**.

3.6 Parallelization

A first attempt of parallelization of **FreeFEM** is made here with **MPI**. An extended interface with MPI has been added to **FreeFEM** version 3.5, (see the MPI documentation for the functionality of the language).

3.6.1 MPI

MPI Keywords

The following keywords and concepts are used:

- `mpiComm` to defined a *communication world*
- `mpiGroup` to defined a group of *processors* in the communication world
- `mpiRequest` to defined a request to wait for the end of the communication

MPI Constants

- `mpisize` The total number of *processes*,
- `mpirank` the id-number of my current process in `{0, ..., mpisize-1}`,
- `mpiUndefined` The `MPI_Undefined` constant,
- `mpiAnySource` The `MPI_ANY_SOURCE` constant,
- `mpiCommWorld` The `MPI_COMM_WORLD` constant,
- `[ ... ]` and all the keywords of `MPI_Op` for the *reduce* operator: `mpiMAX`, `mpiMIN`, `mpiSUM`, `mpiPROD`, `mpiLAND`, `mpiLOR`, `mpiLXOR`, `mpiBAND`, `mpiBXOR`.

MPI Constructor

```
// Parameters
int[int] proc1 = [1, 2], proc2 = [0, 3];
int color = 1;
int key = 1;

// MPI ranks
cout << "MPI rank = " << mpirank << endl;

// MPI
mpiComm comm(mpiCommWorld, 0, 0); //set a MPI_Comm to MPI_COMM_WORLD

mpiGroup grp(proc1); //set MPI_Group to proc 1,2 in MPI_COMM_WORLD
mpiGroup grp1(comm, proc1); //set MPI_Group to proc 1,2 in comm

mpiComm ncomm1(mpiCommWorld, grp); //set the MPI_Comm form grp
```

(continues on next page)

(continued from previous page)

```

mpiComm ncomm2(comm, color, key); //MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *ncomm)

mpiRequest rq; //defined an MPI_Request
mpiRequest[int] arq(10); //defined an array of 10 MPI_Request
```

MPI Functions

```
mpiComm Comm(mpiCommWorld, 0, 0);

int MPICommSize = mpiSize(Comm);
int MPIRank = mpiRank(Comm);

if (MPIRank == 0) cout << "MPI Comm size = " << MPICommSize << endl;
cout << "MPI rank in Comm = " << mpiRank(Comm) << endl;

mpiRequest Req;
mpiRequest[int] ReqArray(10);

for (int i = 0; i < MPICommSize; i++){
     //return processor i with no Resquest in MPI_COMM_WORLD
    processor(i);
    //return processor any source with no Resquest in MPI_COMM_WORLD
    processor(mpiAnySource);
    //return processor i with no Resquest in Comm
    processor(i, Comm);
    //return processor i with no Resquest in Comm
    processor(Comm, i);
    //return processor i with Resquest rq in Comm
    /* processor(i, Req, Comm);
    //return processor i with Resquest rq in MPI_COMM_WORLD
    processor(i, Req); */
    //return processor i in MPI_COMM_WORLD in block mode for synchronously communication
    processorblock(i);
    //return processor any source in MPI_COMM_WORLD in block mode for synchronously communication
    processorblock(mpiAnySource);
    //return processor i in in Comm in block mode
    processorblock(i, Comm);
}

mpiBarrier(Comm); //do a MPI_Barrier on communicator Comm
mpiWaitAny(ReqArray); //wait add of Request array,
mpiWait(Req); //wait on a Request
real t = mpiWtime(); //return MPIWtime in second
real tick = mpiWtick(); //return MPIWTick in second
```

where a `processor` is just a integer rank, pointer to a `MPI_comm` and pointer to a `MPI_Request`, and `processorblock` with a special `MPI_Request`.

MPI Communicator operator

```
int status; //to get the MPI status of send / recv
real a, b;

mpiComm comm(mpiCommWorld, 0, 0);
mpiRequest req;

//send a,b asynchronously to the process 1
processor(1) << a << b;
//receive a,b synchronously from the process 10
processor(10) >> a >> b;

//broadcast from processor of comm to other comm processor
// broadcast(processor(10, comm), a);
//send synchronously to the process 10 the data a
status = Send(processor(10, comm), a);
//receive synchronously from the process 10 the data a
status = Recv(processor(10, comm), a);

//send asynchronously to the process 10 the data a without request
status = Isend(processor(10, comm), a);
//send asynchronously to the process 10 the data a with request
status = Isend(processor(10, comm, req), a);
//receive asynchronously from the process 10 the data a
status = Irecv(processor(10, req), a);
//Error asynchronously without request.
// status = Irecv(processor(10), a);
```

where the data type of `a` can be of type of `int`, `real`, `complex`, `int[int]`, `real[int]`, `complex[int]`, `int[int,int]`, `double[int,int]`, `complex[int,int]`, `mesh`, `mesh3`, `mesh[int]`, `mesh3[int]`, `matrix`, `matrix<complex>`

```
//send asynchronously to the process 10 the data a with request
processor(10, req) << a ;
//receive asynchronously from the process 10 the data a with request
processor(10, req) >> a ;
```

If `a, b` are arrays or full matrices of `int`, `real`, or `complex`, we can use the following MPI functions:

```
mpiAlltoall(a, b, [comm]);
mpiAllgather(a, b, [comm]);
mpiGather(a, b, processor(..) );
mpiScatter(a, b, processor(..));
mpiReduce(a, b, processor(..), mpiMAX);
mpiAllReduce(a, b, comm, mpiMAX);
```

Thank you to Guy-Antoine Atenekeng Kahou for his help to code this interface.

Schwarz example in parallel

This example is a rewritting of example *Schwarz overlapping*.

```
ff-mpirun -np 2 SchwarzParallel.edp
# OR
mpirun -np 2 FreeFem++-mpi SchwarzParallel.edp
```

```
if (mpisize != 2){
    cout << " sorry, number of processors !=2 " << endl;
    exit(1);
}

// Parameters
verbosity = 0;
int interior = 2;
int exterior = 1;
int n = 4;

// Mesh
border a(t=1, 2){x=t; y=0; label=exterior;}
border b(t=0, 1){x=2; y=t; label=exterior;}
border c(t=2, 0){x=t; y=1; label=exterior;}
border d(t=1, 0){x=1-t; y=t; label=interior;}
border e(t=0, pi/2){x=cos(t); y=sin(t); label=interior;}
border e1(t=pi/2, 2*pi){x=cos(t); y=sin(t); label=exterior;}
mesh[int] Th(mpisize);
if (mpirank == 0)
    Th[0] = buildmesh(a(5*n) + b(5*n) + c(10*n) + d(5*n));
else
    Th[1] = buildmesh(e(5*n) + e1(25*n));

broadcast(processor(0), Th[0]);
broadcast(processor(1), Th[1]);

// Fespace
fespace Vh(Th[mpirank], P1);
Vh u = 0, v;

fespace Vhother(Th[1-mpirank], P1);
Vhother U = 0;

//Problem
int i = 0;
problem pb (u, v, init=i, solver=Cholesky)
    = int2d(Th[mpirank])(
          dx(u)*dx(v)
        + dy(u)*dy(v)
    )
    - int2d(Th[mpirank])(
          v
    )
    + on(interior, u=U)
    + on(exterior, u= 0 )
    ;

// Loop
for (i = 0; i < 20; i++){
    cout << mpirank << " - Loop " << i << endl;

    // Solve
    pb;
    //send u to the other proc, receive in U
    processor(1-mpirank) << u[];
    processor(1-mpirank) >> U[];
```

(continues on next page)

(continued from previous page)

```

    // Error
    real err0, err1;
    err0 = int1d(Th[mpirank],interior)(square(U - u));
    // send err0 to the other proc, receive in err1
    processor(1-mpirank) << err0;
    processor(1-mpirank) >> err1;
    real err = sqrt(err0 + err1);
    cout << " err = " << err << " - err0 = " << err0 << " - err1 = " << err1 << endl;
    if (err < 1e-3) break;
}
if (mpirank == 0)
    plot(u, U);
```

Todo: script freeze in the loop

True parallel Schwarz example

Thank you to F. Nataf

This is a explanation of the two examples *MPI-GMRES 2D* and *MPI-GMRES 3D*, a Schwarz parallel with a complexity almost independent of the number of process (with a coarse grid preconditioner).

To solve the following Poisson problem on domain Ω with boundary Γ in $L^2(\Omega)$:

$$\begin{array}{rcll} -\Delta u & = & f & \text{in } \Omega \\ u & = & g & \text{on } \Gamma \end{array}$$

where f and g are two given functions of $L^2(\Omega)$ and of $H^{\frac{1}{2}}(\Gamma)$,

Lets introduce $(\pi_i)_{i=1,..,N_p}$ a regular partition of the unity of Ω, q-e-d:

$$\pi_i \in \mathcal{C}^0(\Omega): \quad \pi_i \geq 0 \text{ and } \sum_{i=1}^{N_p} \pi_i = 1.$$

Denote Ω_i the sub domain which is the support of π_i function and also denote Γ_i the boundary of Ω_i.

The parallel Schwarz method is:

Let $\ell = 0$ the iterator and a initial guest u^0 respecting the boundary condition (i.e. $u^0_{|\Gamma} = g$).

$$\begin{array}{rcll} \forall i = 1.., N_p: & & & \\ -\Delta u_i^\ell & = & f & \text{in } \Omega_i \\ u_i^\ell & = & u^\ell & \text{on } \Gamma_i \setminus \Gamma \\ u_i^\ell & = & g & \text{on } \Gamma_i \cap \Gamma \end{array}$$

$$u^{\ell+1} = \sum_{i=1}^{N_p} \pi_i u_i^\ell \tag{3.30}$$

After discretization with the Lagrange finite element method, with a compatible mesh $\mathcal{T}_{h\,i}$ of Ω_i, i. e., the exist a global mesh $\mathcal{T}_h$ such that $\mathcal{T}_{h\,i}$ is include in $\mathcal{T}_h$.

Let us denote:

- $V_{h\,i}$ the finite element space corresponding to domain Ω_i,

- $\mathcal{N}_{hi}$ is the set of the degree of freedom σ_i^k,
- $\mathcal{N}_{hi}^{\Gamma_i}$ is the set of the degree of freedom of V_{hi} on the boundary Γ_i of Ω_i,
- $\sigma_i^k(v_h)$ is the value the degree of freedom k,
- $V_{0hi} = \{v_h \in V_{hi} : \forall k \in \mathcal{N}_{hi}^{\Gamma_i}, \quad \sigma_i^k(v_h) = 0\}$,
- The conditional expression $a\ ?\ b : c$ is defined like in :c'C' of C++ language by

$$a?b : c \equiv \begin{cases} \text{if } a \text{ is true then return b} \\ \text{else return } c \end{cases} .$$

Note: We never use finite element space associated to the full domain Ω because it is too expensive.

We have to defined to operator to build the previous algorithm:

We denote $u_{h|i}^\ell$ the restriction of u_h^ℓ on V_{hi}, so the discrete problem on Ω_i of problem (3.30) is find $u_{hi}^\ell \in V_{hi}$ such that:

$$\forall v_{hi} \in V_{0i} : \int_{\Omega_i} \nabla v_{hi} \cdot \nabla {u_{hi}}^\ell = \int_{\Omega_i} f v_{hi}, \quad \forall k \in \mathcal{N}_{hi}^{\Gamma_i} \ : \ \sigma_i^k({u_{hi}}^\ell) = (k \in \Gamma)\ ?\ g_i^k : \sigma_i^k({u_{h|i}}^\ell)$$

where g_i^k is the value of g associated to the degree of freedom $k \in \mathcal{N}_{hi}^{\Gamma_i}$.

In **FreeFEM**, it can be written has with `U` is the vector corresponding to $u_{h|i}^\ell$ and the vector `U1` is the vector corresponding to u_{hi}^ℓ is the solution of:

```
real[int] U1(Ui.n);
real[int] b = onG .* U;
b = onG ? b : Bi ;
U1 = Ai^-1*b;
```

where `onG`$[i] = (i \in \Gamma_i \setminus \Gamma)?1 : 0$, and `Bi` the right of side of the problem, are defined by

```
// Fespace
fespace Whi(Thi, P2);

// Problem
varf vPb (U, V)
    = int3d(Thi)(
          grad(U)'*grad(V)
    )
    + int3d(Thi)(
          F*V
    )
    + on(1, U=g)
    + on(10, U=G)
    ;

varf vPbon (U, V) = on(10, U=1) + on(1, U=0);

matrix Ai = vPb (Whi, Whi, solver=sparsesolver);
real[int] onG = vPbon(0, Whi);
real[int] Bi=vPb(0, Whi);
```

where the **FreeFEM** label of Γ is 1 and the label of $\Gamma_i \setminus \Gamma$ is 10.

To build the transfer/update part corresponding to (3.30) equation on process i, let us call `njpart` the number the neighborhood of domain of Ω_i (i.e: π_j is none 0 of Ω_i), we store in an array `jpart` of size `njpart` all this neighborhood.

Let us introduce two array of matrix, `Smj[j]` to defined the vector to send from i to j a neighborhood process, and the matrix $rMj[j]$ to after to reduce owith neighborhood j domain.

So the tranfert and update part compute $v_i = \pi_i u_i + \sum_{j \in J_i} \pi_j u_j$ and can be write the **FreeFEM** function Update:

```
func bool Update (real[int] &ui, real[int] &vi){
    int n = jpart.n;
    for (int j = 0; j < njpart; ++j) Usend[j][] = sMj[j]*ui;
    mpiRequest[int] rq(n*2);
    for (int j = 0; j < n; ++j) Irecv(processor(jpart[j], comm,rq[j]), Ri[j][]);
    for (int j = 0; j < n; ++j) Isend(processor(jpart[j], comm, rq[j+n]), Si[j][]);
    for (int j = 0; j < n*2; ++j) int k = mpiWaitAny(rq);
    // apply the unity local partition
    vi = Pii*ui; //set to pi_i u_i
    for (int j = 0; j < njpart; ++j) vi += rMj[j]*Vrecv[j][]; //add pi_j u_j
    return true;
}
```

where the buffer are defined by:

```
InitU(njpart, Whij, Thij, aThij, Usend) //defined the send buffer
InitU(njpart, Whij, Thij, aThij, Vrecv) //defined the revc buffer
```

with the following macro definition:

```
macro InitU(n, Vh, Th, aTh, U) Vh[int] U(n); for (int j = 0; j < n; ++j){Th = aTh[j];␣
↪U[j] = 0;}
```

First GMRES algorithm: you can easily accelerate the fixed point algorithm by using a parallel GMRES algorithm after the introduction the following affine $\mathcal{A}_i$ operator sub domain Ω_i.

```
func real[int] DJ0 (real[int]& U){
    real[int] V(U.n), b = onG .* U;
    b = onG ? b : Bi ;
    V = Ai^-1*b;
    Update(V, U);
    V -= U;
    return V;
}
```

Where the parallel `MPIGMRES` or `MPICG` algorithm is just a simple way to solve in parallel the following $A_i x_i = b_i, i = 1, .., N_p$ by just changing the dot product by reduce the local dot product of all process with the following MPI code:

```
template<class R> R ReduceSum1(R s, MPI_Comm *comm){
    R r = 0;
    MPI_Allreduce(&s, &r, 1, MPI_TYPE<R>::TYPE(), MPI_SUM, *comm );
    return r;
}
```

This is done in `MPIGC` dynamics library tool.

Second GMRES algorithm: Use scharwz algorithm as a preconditioner of basic GMRES method to solving the parallel problem.

```
func real[int] DJ (real[int]& U){ //the original problem
    ++kiter;
    real[int] V(U.n);
    V = Ai*U;
    V = onGi ? 0.: V; //remove boundary term
    return V;
}

func real[int] PDJ (real[int]& U){ //the preconditioner
    real[int] V(U.n);
    real[int] b = onG ? 0. : U;
    V = Ai^-1*b;
    Update(V, U);
    return U;
}
```

Third GMRES algorithm: Add a coarse solver to the previous algorithm

First build a coarse grid on processor 0, and the

```
matrix AC, Rci, Pci;
if (mpiRank(comm) == 0)
    AC = vPbC(VhC, VhC, solver=sparsesolver); //the coarse problem

Pci = interpolate(Whi, VhC); //the projection on coarse grid
Rci = Pci'*Pii; //the restriction on Process i grid with the partition pi_i

func bool CoarseSolve (real[int]& V, real[int]& U, mpiComm& comm){
    // solving the coarse problem
    real[int] Uc(Rci.n), Bc(Uc.n);
    Uc = Rci*U;
    mpiReduce(Uc, Bc, processor(0, comm), mpiSUM);
    if (mpiRank(comm) == 0)
    Uc = AC^-1*Bc;
    broadcast(processor(0, comm), Uc);
    V = Pci*Uc;
}
```

The New preconditionner

```
func real[int] PDJC (real[int]& U){
    // Idea: F. Nataf.
    // 0 ~ (I C1A)(I-C2A) => I ~ - C1AC2A +C1A +C2A
    // New Prec P= C1+C2 - C1AC2 = C1(I- A C2) +C2
    // ( C1(I- A C2) +C2 ) Uo
    // V = - C2*Uo
    // ....
    real[int] V(U.n);
    CoarseSolve(V, U, comm);
    V = -V; //-C2*Uo
    U += Ai*V; //U = (I-A C2) Uo
    real[int] b = onG ? 0. : U;
    U = Ai^-1*b; //C1( I -A C2) Uo
    V = U - V;
    Update(V, U);
    return U;
}
```

The code of the 4 algorithms:

```
real epss = 1e-6;
int rgmres = 0;
if (gmres == 1){
    rgmres = MPIAffineGMRES(DJ0, u[], veps=epss, nbiter=300,
        comm=comm, dimKrylov=100, verbosity=ipart?0: 50);
    real[int] b = onG .* u[];
    b = onG ? b : Bi ;
    v[] = Ai^-1*b;
    Update(v[], u[]);
}
else if (gmres == 2)
    rgmres = MPILinearGMRES(DJ, precon=PDJ, u[], Bi, veps=epss,
        nbiter=300, comm=comm, dimKrylov=100, verbosity=ipart?0: 50);
else if (gmres == 3)
    rgmres = MPILinearGMRES(DJ, precon=PDJC, u[], Bi, veps=epss,
        nbiter=300, comm=comm, dimKrylov=100, verbosity=ipart?0: 50);
else //algo Shwarz for demo
    for(int iter = 0; iter < 10; ++iter)
        ...
```

We have all ingredient to solve in parallel if we have et the partitions of the unity. To build this partition we do:

The initial step on process 1 to build a coarse mesh, ${\mathcal{T}_h}^*$ of the full domain, and build the partition π function constant equal to i on each sub domain $\mathcal{O}_i, i = 1, .., N_p$, of the grid with the `metis` graph partitioner [KARYPIS1995] and on each process i in $1.., N_p$ do

1. Broadcast from process 1, the mesh ${\mathcal{T}_h}^*$ (call `Thii` in **FreeFEM** script), and π function,
2. remark that the characteristic function $\mathbb{1}_{\mathcal{O}_i}$ of domain $\mathcal{O}_i$, is defined by $(\pi = i)?1 : 0$,
3. Let us call Π_P^2 (resp. Π_V^2) the L^2 on P_h^* the space of the constant finite element function per element on ${\mathcal{T}_h}^*$ (resp. V_h^* the space of the affine continuous finite element per element on ${\mathcal{T}_h}^*$) and build in parallel the π_i and Ω_i, such that $\mathcal{O}_i \subset \Omega_i$ where $\mathcal{O}_i = supp((\Pi_V^2 \Pi_C^2)^m \mathbb{1}_{O_i})$, and m is a the overlaps size on the coarse mesh (generally one), (this is done in function `AddLayers(Thii,suppii[],nlayer,phii[]);` We choose a function $\pi_i^* = (\Pi_1^2 \Pi_0^2)^m \mathbb{1}_{\mathcal{O}_i}$ so the partition of the unity is simply defined by

$$\pi_i = \frac{\pi_i^*}{\sum_{j=1}^{N_p} \pi_j^*}$$

The set J_i of neighborhood of the domain Ω_i, and the local version on V_{hi} can be defined the array `jpart` and `njpart` with:

```
Vhi pii = piistar;
Vhi[int] pij(npij); //local partition of 1 = pii + sum_j pij[j]
int[int] jpart(npart);
int njpart = 0;
Vhi sumphi = piistar;
for (int i = 0; i < npart; ++i)
    if (i != ipart){
        if (int3d(Thi)(pijstar,j) > 0){
            pij[njpart] = pijstar;
            sumphi[] += pij[njpart][];
            jpart[njpart++] = i;
        }
    }
```

(continues on next page)

(continued from previous page)

```
pii[] = pii[] ./ sumphi[];
for (int j = 0; j < njpart; ++j)
pij[j][] = pij[j][] ./ sumphi[];
jpart.resize(njpart);
```

4. We call $\mathcal{T}_{h_{ij}}^*$ the sub mesh part of $\mathcal{T}_{h_i}$ where π_j are none zero. And thanks to the function `trunc` to build this array,

```
for(int jp = 0; jp < njpart; ++jp)
    aThij[jp] = trunc(Thi, pij[jp] > 1e-10, label=10);
```

5. At this step we have all on the coarse mesh, so we can build the fine final mesh by splitting all meshes: `Thi, Thij[j], Thij[j]` with **FreeFEM** `trunc` mesh function which do restriction and slipping.

6. The construction of the send/recv matrices `sMj` and *freefem:'rMj*: can done with this code:

```
mesh3 Thij = Thi;
fespace Whij(Thij, Pk);
matrix Pii; Whi wpii = pii; Pii = wpii[]; //Diagonal matrix corresponding␣
↪X pi_i
matrix[int] sMj(njpart), rMj(njpart); //M send/recive case
for (int jp = 0; jp < njpart; ++jp){
    int j = jpart[jp];
    Thij = aThij[jp]; //change mesh to change Whij, Whij
    matrix I = interpolate(Whij, Whi); //Whij <- Whi
    sMj[jp] = I*Pii; //Whi -> s Whij
    rMj[jp] = interpolate(Whij, Whi, t=1); //Whij -> Whi
}
```

To buil a not too bad application, all variables come from parameters value with the following code

```
include "getARGV.idp"
verbosity = getARGV("-vv", 0);
int vdebug = getARGV("-d", 1);
int ksplit = getARGV("-k", 10);
int nloc = getARGV("-n", 25);
string sff = getARGV("-p, ", "");
int gmres = getARGV("-gmres", 3);
bool dplot = getARGV("-dp", 0);
int nC = getARGV("-N", max(nloc/10, 4));
```

And small include to make graphic in parallel of distribute solution of vector u on mesh T_h with the following interface:

```
include "MPIplot.idp"
func bool plotMPIall(mesh &Th, real[int] &u, string cm){
    PLOTMPIALL(mesh, Pk, Th, u, {cmm=cm, nbiso=20, fill=1, dim=3, value=1});
    return 1;
}
```

Note: The `cmm=cm, ...` in the macro argument is a way to quote macro argument so the argument is `cmm=cm, ....`

3.6.2 Parallel sparse solvers

Parallel sparse solvers use several processors to solve linear systems of equation. Like sequential, parallel linear solvers can be direct or iterative. In **FreeFEM** both are available.

Using parallel sparse solvers in FreeFEM

We recall that the `solver` parameters are defined in the following commands: `solve`, `problem`, `set` (setting parameter of a matrix) and in the construction of the matrix corresponding to a bilinear form. In these commands, the parameter `solver` must be set to `sparsesolver` for parallel sparse solver. We have added specify parameters to these command lines for parallel sparse solvers. These are:

- `lparams` : vector of integer parameters (`l` is for the `C++` type `long`)
- `dparams` : vector of real parameters
- `sparams` : string parameters
- `datafilename` : name of the file which contains solver parameters

The following four parameters are only for direct solvers and are vectors. These parameters allow the user to preprocess the matrix (see the section on *sparse direct solver* for more information).

- `permr` : row permutation (integer vector)
- `permc` : column permutation or inverse row permutation (integer vector)
- `scaler` : row scaling (real vector)
- `scalec` : column scaling (real vector)

There are two possibilities to control solver parameters. The first method defines parameters with `lparams`, `dparams` and `sparams` in `.edp` file.

The second one reads the solver parameters from a data file. The name of this file is specified by `datafilename`. If `lparams`, `dparams`, `sparams` or `datafilename` is not provided by the user, the solver's default values are used.

To use parallel solver in **FreeFEM**, we need to load the dynamic library corresponding to this solver. For example to use MUMPS solver as parallel solver in **FreeFEM**, write in the `.edp` file `load "MUMPS_FreeFem"`.

If the libraries are not loaded, the default sparse solver will be loaded (default sparse solver is `UMFPACK`). The Table 3.2 gives this new value for the different libraries.

Table 3.2: Default sparse solver for real and complex arithmetics when we load a parallel sparse solver library

Libraries	Default sparse solver	
	real	complex
MUMPS_FreeFem	mumps	mumps
real_SuperLU_DIST_FreeFem	SuperLU_DIST	previous solver
complex_SuperLU_DIST_FreeFem	previous solver	SuperLU_DIST
real_pastix_FreeFem	PaStiX	previous solver
complex_pastix_FreeFem	previous solver	PaStiX
hips_FreeFem	hips	previous solver
hypre_FreeFem	hypre	previous solver
parms_FreeFem	parms	previous solver

We also add functions (see Table 3.3) with no parameter to change the default sparse solver in the `.edp` file. To use these functions, we need to load the library corresponding to the solver. An example of using different parallel sparse solvers for the same problem is given in *Direct solvers example*.

Table 3.3: Functions that allow to change the default sparse solver for real and complex arithmetics and the result of these functions

Function	default sparse solver	
	real	complex
defaulttoMUMPS()	mumps	mumps
realdefaulttoSuperLUdist()	SuperLU_DIST	previous solver
complexdefaulttoSuperLUdist()	previous solver	SuperLU_DIST
realdefaultopastix()	pastix	previous solver
complexdefaulttopastix()	previous solver	pastix
defaulttohips()	hips	previous solver
defaulttohypre()	hypre	previous solver
defaulttoparms()	parms	previous solver

Tip: Test direct solvers

```
load "MUMPS_FreeFem"
//default solver: real-> MUMPS, complex -> MUMPS
load "real_SuperLU_DIST_FreeFem"
//default solver: real-> SuperLU_DIST,
complex -> MUMPS load "real_pastix_FreeFem"
//default solver: real-> pastix, complex -> MUMPS

// Solving with pastix
{
    matrix A =
        [[1, 2, 2, 1, 1],
        [ 2, 12, 0, 10, 10],
        [ 2, 0, 1, 0, 2],
        [ 1, 10, 0, 22, 0.],
        [ 1, 10, 2, 0., 22]];

    real[int] xx = [1, 32, 45, 7, 2], x(5), b(5), di(5);
    b = A*xx;
    cout << "b =" << b << endl; cout << "xx =" << xx << endl;

    set(A, solver=sparsesolver, datafilename="ffpastix_iparm_dparm.txt");
    cout << "solve" << endl;
    x = A^-1*b;
    cout << "b =" << b << endl;
    cout << "x =" << endl;
    cout << x << endl;
    di = xx - x;
    if (mpirank == 0){
        cout << "x-xx =" << endl;
        cout << "Linf =" << di.linfty << ", L2 =" << di.l2 << endl;
    }
}

// Solving with SuperLU_DIST
realdefaulttoSuperLUdist();
//default solver: real-> SuperLU_DIST, complex -> MUMPS
{
    matrix A =
        [[1, 2, 2, 1, 1],
```

(continues on next page)

(continued from previous page)

```
        [ 2, 12, 0, 10, 10],
        [ 2, 0, 1, 0, 2],
        [ 1, 10, 0, 22, 0.],
        [ 1, 10, 2, 0., 22]];

    real[int] xx = [1, 32, 45, 7, 2], x(5), b(5), di(5);
    b = A*xx;
    cout << "b =" << b << endl;
    cout << "xx =" << xx << endl;

    set(A, solver=sparsesolver, datafilename="ffsuperlu_dist_fileparam.txt");
    cout << "solve" << endl;
    x = A^-1*b;
    cout << "b =" << b << endl;
    cout << "x =" << endl;
    cout << x << endl;
    di = xx - x;
    if (mpirank == 0){
        cout << "x-xx =" << endl;
        cout << "Linf =" << di.linfty << ", L2 =" << di.l2 << endl;
    }
}

// Solving with MUMPS
defaulttoMUMPS();
//default solver: real-> MUMPS, complex -> MUMPS
{
    matrix A =
        [[1, 2, 2, 1, 1],
        [ 2, 12, 0, 10, 10],
        [ 2, 0, 1, 0, 2],
        [ 1, 10, 0, 22, 0.],
        [ 1, 10, 2, 0., 22]];

    real[int] xx = [1, 32, 45, 7, 2], x(5), b(5), di(5);
    b = A*xx;
    cout << "b =" << b << endl;
    cout << "xx =" << xx << endl;

    set(A, solver=sparsesolver, datafilename="ffmumps_fileparam.txt");
    cout << "solving solution" << endl;
    x = A^-1*b;
    cout << "b =" << b << endl;
    cout << "x =" << endl;
    cout << x << endl;
    di = xx - x;
    if (mpirank == 0){
        cout << "x-xx =" << endl;
        cout << "Linf =" << di.linfty << ", L2" << di.l2 << endl;
    }
}
```

Sparse direct solver

In this section, we present the sparse direct solvers interfaced with **FreeFEM**.

MUMPS solver

MUltifrontal Massively Parallel Solver (MUMPS) is an open-source library.

This package solves linear system of the form $A\,x = b$ where A is a square sparse matrix with a direct method. The square matrix considered in MUMPS can be either unsymmetric, symmetric positive definite or general symmetric.

The method implemented in MUMPS is a direct method based on a multifrontal approach. It constructs a direct factorization $A = L\,U$, $A = L^t\,D\,L$ depending of the symmetry of the matrix A.

MUMPS uses the following libraries :

- BLAS,
- BLACS,
- ScaLAPACK.

> **Warning:** MUMPS does not solve linear system with a rectangular matrix.

MUMPS parameters:

There are four input parameters in MUMPS. Two integers `SYM` and `PAR`, a vector of integer of size 40 `INCTL` and a vector of real of size 15 `CNTL`.

The first parameter gives the type of the matrix: 0 for unsymmetric matrix, 1 for symmetric positive matrix and 2 for general symmetric.

The second parameter defined if the host processor work during the factorization and solves steps : `PAR=1` host processor working and `PAR=0` host processor not working.

The parameter `INCTL` and `CNTL` is the control parameter of MUMPS. The vectors `ICNTL` and `CNTL` in MUMPS becomes with index 1 like vector in `Fortran`. For more details see the MUMPS user's guide.

We describe now some elements of the main parameters of `ICNTL` for MUMPS.

- **Input matrix parameter The input matrix is controlled by parameters `ICNTL(5)` and `ICNTL(18)`.**
  The matrix format (resp. matrix pattern and matrix entries) are controlled by `INCTL(5)` (resp. `INCTL(18)`).

  The different values of `ICNTL(5)` are 0 for assembled format and 1 for element format. In the current release of **FreeFEM**, we consider that FE matrix or matrix is storage in assembled format. Therefore, `INCTL(5)` is treated as 0 value.

  The main option for `ICNTL(18)`: `INCLTL(18)=0` centrally on the host processor, `ICNTL(18)=3` distributed the input matrix pattern and the entries (recommended option for distributed matrix by developer of MUMPS). For other values of `ICNTL(18)` see the MUMPS user's guide. These values can be used also in **FreeFEM**.

  The default option implemented in **FreeFEM** are `ICNTL(5)=0` and `ICNTL(18)=0`.

- **Preprocessing parameter** The preprocessed matrix A_p that will be effectively factored is defined by

$$A_p = P\,D_r\,A\,Q_c\,D_c P^t$$

  where P is the permutation matrix, Q_c is the column permutation, D_r and D_c are diagonal matrix for respectively row and column scaling.

  The ordering strategy to obtain P is controlled by parameter `ICNTL(7)`. The permutation of zero free diagonal Q_c is controlled by parameter `ICNTL(6)`. The row and column scaling is controlled

by parameter `ICNTL(18)`. These option are connected and also strongly related with `ICNTL(12)` (see the MUMPS user's guide for more details).

The parameters `permr`, `scaler`, and `scalec` in **FreeFEM** allow to give permutation matrix(P), row scaling (D_r) and column scaling (D_c) of the user respectively.

Calling MUMPS in FreeFEM

To call MUMPS in **FreeFEM**, we need to load the dynamic library `MUMPS_freefem.dylib` (MacOSX), `MUMPS_freefem.so` (Unix) or `MUMPS_freefem.dll` (Windows).

This is done in typing `load "MUMPS_FreeFem"` in the `.edp` file. We give now the two methods to give the option of MUMPS solver in **FreeFEM**.

- **Solver parameters is defined in .edp file: In this method, we need to give the parameters `lparams` and `dparams`.** These parameters are defined for MUMPS by :
  - `lparams[0] = SYM`, `lparams[1] = PAR`,
  - $\forall i = 1,\ldots,40$, `lparams[i+1] = ICNTL(i)`
  - $\forall i = 1,\ldots,15$, `dparams[i-1] = CNTL(i)`
- **Reading solver parameters on a file:**

  The structure of data file for MUMPS in **FreeFEM** is : first line parameter `SYM` and second line parameter `PAR` and in the following line the different value of vectors `ICNTL` and `CNTL`. An example of this parameter file is given in `ffmumpsfileparam.txt`.

```
0 /* SYM :: 0 for non symmetric matrix, 1 for symmetric definite positive matrix and 2 general symmetric matrix*/
1 /* PAR :: 0 host not working during factorization and solves steps, 1 host working during factorization and solves steps*/
-1 /* ICNTL(1) :: output stream for error message */
-1 /* ICNTL(2) :: output for diagnostic printing, statics and warning message */
-1 /* ICNTL(3) :: for global information */
0 /* ICNTL(4) :: Level of printing for error, warning and diagnostic message */
0 /* ICNTL(5) :: matrix format : 0 assembled format, 1 elemental format. */
7 /* ICNTL(6) :: control option for permuting and/or scaling the matrix in analysis phase */
3 /* ICNTL(7) :: pivot order strategy : AMD, AMF, metis, pord scotch*/
77 /* ICNTL(8) :: Row and Column scaling strategy */
1 /* ICNTL(9) :: 0 solve Ax = b, 1 solve the transposed system A^t x = b : parameter is not considered in the current release of FreeFEM*/
0 /* ICNTL(10) :: number of steps of iterative refinement */
0 /* ICNTL(11) :: statics related to linear system depending on ICNTL(9) */
1 /* ICNTL(12) :: constrained ordering strategy for general symmetric matrix */
0 /* ICNTL(13) :: method to control splitting of the root frontal matrix */
20 /* ICNTL(14) :: percentage increase in the estimated working space (default 20\%)*/
0 /* ICNTL(15) :: not used in this release of MUMPS */
0 /* ICNTL(16) :: not used in this release of MUMPS */
0 /* ICNTL(17) :: not used in this release of MUMPS */
3 /* ICNTL(18) :: method for given : matrix pattern and matrix entries : */
```

(continues on next page)

(continued from previous page)

```
0 /* ICNTL(19) :: method to return the Schur complement matrix */
0 /* ICNTL(20) :: right hand side form ( 0 dense form, 1 sparse form) : parameter will be set to 0 for FreeFEM */
0 /* ICNTL(21) :: 0, 1 kept distributed solution : parameter is not considered in the current release of FreeFEM */
0 /* ICNTL(22) :: controls the in-core/out-of-core (OOC) facility */
0 /* ICNTL(23) :: maximum size of the working memory in Megabyte than MUMPS can allocate per working processor */
0 /* ICNTL(24) :: control the detection of null pivot */
0 /* ICNTL(25) :: control the computation of a null space basis */
0 /* ICNTL(26) :: This parameter is only significant with Schur option (ICNTL(19) not zero). : parameter is not considered in the current release of FreeFEM */
-8 /* ICNTL(27) (Experimental parameter subject to change in next release of MUMPS) :: control the blocking factor for multiple righthand side during the solution phase : parameter is not considered in the current release of FreeFEM */
0 /* ICNTL(28) :: not used in this release of MUMPS*/
0 /* ICNTL(29) :: not used in this release of MUMPS*/
0 /* ICNTL(30) :: not used in this release of MUMPS*/
0 /* ICNTL(31) :: not used in this release of MUMPS*/
0 /* ICNTL(32) :: not used in this release of MUMPS*/
0 /* ICNTL(33) :: not used in this release of MUMPS*/
0 /* ICNTL(34) :: not used in this release of MUMPS*/
0 /* ICNTL(35) :: not used in this release of MUMPS*/
0 /* ICNTL(36) :: not used in this release of MUMPS*/
0 /* ICNTL(37) :: not used in this release of MUMPS*/
0 /* ICNTL(38) :: not used in this release of MUMPS*/
1 /* ICNTL(39) :: not used in this release of MUMPS*/
0 /* ICNTL(40) :: not used in this release of MUMPS*/
0.01 /* CNTL(1) :: relative threshold for numerical pivoting */
1e-8 /* CNTL(2) :: stopping criteria for iterative refinement */
-1 /* CNTL(3) :: threshold for null pivot detection */
-1 /* CNTL(4) :: determine the threshold for partial pivoting */
0.0 /* CNTL(5) :: fixation for null pivots */
0 /* CNTL(6) :: not used in this release of MUMPS */
0 /* CNTL(7) :: not used in this release of MUMPS */
0 /* CNTL(8) :: not used in this release of MUMPS */
0 /* CNTL(9) :: not used in this release of MUMPS */
0 /* CNTL(10) :: not used in this release of MUMPS */
0 /* CNTL(11) :: not used in this release of MUMPS */
0 /* CNTL(12) :: not used in this release of MUMPS */
0 /* CNTL(13) :: not used in this release of MUMPS */
0 /* CNTL(14) :: not used in this release of MUMPS */
0 /* CNTL(15) :: not used in this release of MUMPS */
```

If no solver parameter is given, we used default option of MUMPS solver.

Tip: MUMPS example

A simple example of calling MUMPS in **FreeFEM** with this two methods is given in the *Test solver MUMPS example*.

SuperLU distributed solver

The package SuperLU_DIST solves linear systems using LU factorization. It is a free scientific library

This library provides functions to handle square or rectangular matrix in real and complex arithmetics. The method implemented in SuperLU_DIST is a supernodal method. New release of this package includes a parallel symbolic factorization. This scientific library is written in C and MPI for communications.

SuperLU_DIST parameters:

We describe now some parameters of SuperLU_DIST. The SuperLU_DIST library use a 2D-logical process group. This process grid is specified by $nprow$ (process row) and $npcol$ (process column) such that $N_p = nprow\ npcol$ where N_p is the number of all process allocated for SuperLU_DIST.

The input matrix parameters is controlled by "matrix=" in `sparams` for internal parameter or in the third line of parameters file. The different value are

- `matrix=assembled` global matrix are available on all process
- `matrix=distributedglobal` The global matrix is distributed among all the process
- `matrix=distributed` The input matrix is distributed (not yet implemented)

The option arguments of SuperLU_DIST are described in the section Users-callable routine of the SuperLU users' guide.

The parameter `Fact` and `TRANS` are specified in **FreeFEM** interfaces to SuperLU_DIST during the different steps. For this reason, the value given by the user for this option is not considered.

The factorization LU is calculated in SuperLU_DIST on the matrix A_p.

$$A_p = P_c\ P_r\ D_r\ A\ D_c\ P_c^t$$

where P_c and P_r is the row and column permutation matrix respectively, D_r and D_c are diagonal matrix for respectively row and column scaling.

The option argument `RowPerm` (resp. `ColPerm`) control the row (resp. column) permutation matrix. D_r and D_c is controlled by the parameter `DiagScale`.

The parameter `permr`, `permc`, `scaler`, and `scalec` in **FreeFEM** is provided to give row permutation, column permutation, row scaling and column scaling of the user respectively.

The other parameters for LU factorization are `ParSymFact` and `ReplaceTinyPivot`. The parallel symbolic factorization works only on a power of two processes and need the `ParMetis` ordering. The default option argument of SuperLU_DIST are given in the file `ffsuperlu_dist_fileparam.txt`.

Calling SuperLU_DIST in FreeFEM

To call SuperLU_DIST in **FreeFEM**, we need to load the library dynamic correspond to interface. This done by the following line `load "real_superlu _DIST_FreeFem"` (resp. `load "complex_superlu_DIST_FreeFem"`) for real (resp. complex) arithmetics in the file `.edp`.

Solver parameters is defined in .edp file:

To call SuperLU_DIST with internal parameter, we used the parameters `sparams`. The value of parameters of SuperLU_DIST in `sparams` are defined by :

- `nprow=1`,
- `npcol=1`,
- `matrix= distributedgloba`,
- `Fact= DOFACT`,

- `Equil=NO`,
- `ParSymbFact=NO`,
- `ColPerm= MMD_AT_PLUS_A`,
- `RowPerm= LargeDiag`,
- `DiagPivotThresh=1.0`,
- `IterRefine=DOUBLE`,
- `Trans=NOTRANS`,
- `ReplaceTinyPivot=NO`,
- `SolveInitialized=NO`,
- `PrintStat=NO`,
- `DiagScale=NOEQUIL`

This value correspond to the parameter in the file `ffsuperlu_dist_fileparam.txt`. If one parameter is not specified by the user, we take the default value of SuperLU_DIST.

Reading solver parameters on a file: The structure of data file for SuperLU_DIST in **FreeFEM** is given in the file `ffsuperlu_dist_fileparam.txt` (default value of the **FreeFEM** interface).

```
1 /* nprow : integer value */
1 /* npcol : integer value */
distributedglobal /* matrix input : assembled, distributedglobal, distributed */
DOFACT /* Fact : DOFACT, SamePattern, SamePattern_SameRowPerm, FACTORED */
NO /* Equil : NO, YES */
NO /* ParSymbFact : NO, YES */
MMD_AT_PLUS_A /* ColPerm : NATURAL, MMD_AT_PLUS_A, MMD_ATA, METIS_AT_PLUS_A, PARMETIS,
↪ MY_PERMC */
LargeDiag /* RowPerm : NOROWPERM, LargeDiag, MY_PERMR */
1.0 /* DiagPivotThresh : real value */
DOUBLE /* IterRefine : NOREFINE, SINGLE, DOUBLE, EXTRA */
NOTRANS /* Trans : NOTRANS, TRANS, CONJ*/
NO /* ReplaceTinyPivot : NO, YES*/
NO /* SolveInitialized : NO, YES*/
NO /* RefineInitialized : NO, YES*/
NO /* PrintStat : NO, YES*/
NOEQUIL /* DiagScale : NOEQUIL, ROW, COL, BOTH*/
```

If no solver parameter is given, we used default option of SuperLU_DIST solver.

Tip: A simple example of calling SuperLU_DIST in **FreeFEM** with this two methods is given in the *Solver superLU_DIST example*.

PaStiX solver

PaStiX (Parallel Sparse matrix package) is a free scientific library under CECILL-C license. This package solves sparse linear system with a direct and block ILU(k) iterative methods. his solver can be applied to a real or complex matrix with a symmetric pattern.

PaStiX parameters:

The input `matrix` parameter of **FreeFEM** depend on PaStiX interface. `matrix = assembled` for non distributed matrix. It is the same parameter for SuperLU_DIST.

There are four parameters in PaStiX : `iparm`, `dparm`, `perm` and `invp`. These parameters are respectively the integer parameters (vector of size 64), real parameters (vector of size 64), permutation matrix and inverse permutation matrix respectively. `iparm` and `dparm` vectors are described in PaStiX RefCard.

The parameters `permr` and `permc` in **FreeFEM** are provided to give permutation matrix and inverse permutation matrix of the user respectively.

Solver parameters defined in .edp file:

To call PaStiX in **FreeFEM** in this case, we need to specify the parameters `lparams` and `dparams`. These parameters are defined by :

$\forall i = 0,\ldots,63$, `lparams[i] = iparm[i]`.

$\forall i = 0,\ldots,63$, `dparams[i] = dparm[i]`.

Reading solver parameters on a file:

The structure of data file for PaStiX parameters in **FreeFEM** is: first line structure parameters of the matrix and in the following line the value of vectors `iparm` and `dparm` in this order.

```
assembled /* matrix input :: assembled, distributed global and distributed */
iparm[0]
iparm[1]
...
...
iparm[63]
dparm[0]
dparm[1]
...
...
dparm[63]
```

An example of this file parameter is given in `ffpastix_iparm_dparm.txt` with a description of these parameters. This file is obtained with the example file `iparm.txt` and `dparm.txt` including in the PaStiX package.

If no solver parameter is given, we use the default option of PaStiX solver.

Tip: A simple example of calling PaStiX in **FreeFEM** with this two methods is given in the *Solver PaStiX example*.

In Table 3.4, we recall the different matrix considering in the different direct solvers.

Table 3.4: Type of matrix used by the different direct sparse solver

direct solver	square matrix			rectangular matrix		
	sym	sym pattern	unsym	sym	sym pattern	unsym
SuperLU_DIST	yes	yes	yes	yes	yes	yes
MUMPS	yes	yes	yes	no	no	no
Pastix	yes	yes	no	no	no	no

Parallel sparse iterative solver

Concerning iterative solvers, we have chosen pARMS, HIPS and Hypre.

Each software implements a different type of parallel preconditioner.

So, pARMS implements algebraic domain decomposition preconditioner type such as additive Schwartz [CAI1989] and interface method; while HIPS implement hierarchical incomplete factorization and finally HYPRE implements multilevel preconditioner are AMG(Algebraic MultiGrid) and parallel approximated inverse.

To use one of these programs in **FreeFEM**, you have to install it independently of **FreeFEM**. It is also necessary to install the MPI communication library which is essential for communication between the processors and, in some cases, software partitioning graphs like METIS or Scotch.

All this preconditioners are used with Krylov subspace methods accelerators.

Krylov subspace methods are iterative methods which consist in finding a solution x of linear system $Ax = b$ inside the affine space $x_0 + K_m$ by imposing that $b - Ax \perp \mathcal{L}_m$, where K_m is Krylov subspace of dimension m defined by $K_m = \{r_0, Ar_0, A^2r_0, ..., A^{m-1}r_0\}$ and $\mathcal{L}_m$ is another subspace of dimension m which depends on type of Krylov subspace. For example in GMRES, $\mathcal{L}_m = AK_m$.

We realized an interface which is easy to use, so that the call of these different softwares in **FreeFEM** is done in the same way. You just have to load the solver and then specify the parameters to apply to the specific solvers. In the rest of this chapter, when we talk about Krylov subspace methods we mean one among GMRES, CG and BICGSTAB.

pARMS solver

pARMS (parallel Algebraic Multilevel Solver) is a software developed by Youssef Saad and al at University of Minnesota.

This software is specialized in the resolution of large sparse non symmetric linear systems of equation. Solvers developed in pARMS are of type "Krylov's subspace".

It consists of variants of GMRES like FGMRES (Flexible GMRES), DGMRES (Deflated GMRES) [SAAD2003] and BICGSTAB. pARMS also implements parallel preconditioner like RAS (Restricted Additive Schwarz) [CAI1989] and Schur Complement type preconditioner.

All these parallel preconditioners are based on the principle of domain decomposition. Thus, the matrix A is partitioned into sub matrices $A_i (i = 1, ..., p)$ where p represents the number of partitions one needs. The union of A_i forms the original matrix. The solution of the overall system is obtained by solving the local systems on A_i (see [SMITH1996]). Therefore, a distinction is made between iterations on A and the local iterations on A_i.

To solve the local problem on A_i there are several preconditioners as **ilut** (Incomplete LU with threshold), **iluk** (Incomplete LU with level of fill in) and **ARMS** (Algebraic Recursive Multilevel Solver).

Tip: Default parameters

```
load "parms_FreeFem" //Tell FreeFem that you will use pARMS

// Mesh
border C(t=0, 2*pi){x=cos(t); y=sin(t); label=1;}
mesh Th = buildmesh (C(50));

// Fespace
fespace Vh(Th, P2); Vh u, v;

// Function
func f= x*y;

// Problem
problem Poisson (u, v, solver=sparsesolver)
    = int2d(Th)(
          dx(u)*dx(v)
```

(continues on next page)

(continued from previous page)

```
17          + dy(u)*dy(v) )
18      + int2d(Th)(
19          - f*v
20      )
21      + on(1, u=0) ;
22
23  // Solve
24  real cpu = clock();
25  Poisson;
26  cout << " CPU time = " << clock()-cpu << endl;
27
28  // Plot
29  plot(u);
```

In line 1, the pARMS dynamic library is loaded with interface **FreeFEM**. After this, in line 15 we specify that the bilinear form will be solved by the last sparse linear solver load in memory which, in this case, is pARMS.

The parameters used in pARMS in this case are the default one since the user does not have to provide any parameter.

Note: In order to see the plot of a parallel script, run the command `FreeFem++-mpi -glut ffglut script.edp`

Here are some default parameters:

- `solver=FGMRES`,
- `Krylov dimension=30`,
- `Maximum of Krylov=1000`,
- `Tolerance for convergence=1e-08` (see book [SAAD2003] to understand all this parameters),
- `preconditionner=Restricted Additif Schwarz` [CAI1989],
- `Inner Krylov dimension=5`,
- `Maximum of inner Krylov dimension=5`,
- `Inner preconditionner=ILUK`.

To specify the parameters to apply to the solver, the user can either give an integer vector for **integer parameters** and real vectors for **real parameters** or provide a **file** which contains those parameters.

Tip: User specifies parameters inside two vectors

Lets us consider Navier-Stokes example. In this example we solve linear systems coming from discretization of Navier-Stokes equations with pARMS. Parameters of solver is specified by user.

```
1  load "parms_FreeFem"
2
3  // Parameters
4  real nu = 1.;
5  int[int] iparm(16);
6  real[int] dparm(6);
7  for (int ii = 0; ii < 16; ii++)
8      iparm[ii] = -1;
9  for (int ii = 0; ii < 6; ii++)
```

(continues on next page)

(continued from previous page)

```
10      dparm[ii] = -1.0; iparm[0]=0;
11
12  // Mesh
13  mesh Th = square(10, 10);
14  int[int] wall = [1, 3];
15  int inlet = 4;
16
17  // Fespace
18  fespace Vh(Th, [P2, P2, P1]);
19
20  // Function
21  func uc = 1.;
22
23  // Problem
24  varf Stokes ([u, v, p], [ush, vsh, psh], solver=sparsesolver)
25      = int2d(Th)(
26            nu*(
27              dx(u)*dx(ush)
28              + dy(u)*dy(ush)
29              + dx(v)*dx(vsh)
30              + dy(v)*dy(vsh)
31          )
32          - p*psh*(1.e-6)
33          - p*(dx(ush) + dy(vsh))
34          - (dx(u) + dy(v))*psh
35      )
36      + on(wall, wall, u=0., v=0.)
37      + on(inlet, u=uc, v=0) ;
38
39  matrix AA = Stokes(Vh, Vh);
40  set(AA, solver=sparsesolver, lparams=iparm, dparams=dparm); //set pARMS as linear
    ↪solver
41  real[int] bb = Stokes(0, Vh);
42  real[int] sol(AA.n);
43  sol = AA^-1 * bb;
```

We need two vectors to specify the parameters of the linear solver. In line 5-6 of the example, we have declared these vectors(`int[int] iparm(16); real[int] dparm(6);`). In line 7-10 we have initialized these vectors by negative values.

We do this because all parameters values in pARMS are positive and if you do not change the negative values of one entry of this vector, the default value will be set.

In Table 3.7 and Table 3.8, we have the meaning of different entries of these vectors.

We run this example on a cluster paradent of Grid5000 and report results in Table 3.5.

Table 3.5: Convergence and time for solving linear system

$n = 471281$ $nnz = 13 \times 10^6$ $Te = 571.29$				
np	add(iluk)		shur(iluk)	
	nit	time	nit	time
4	230	637.57	21	557.8
8	240	364.12	22	302.25
16	247	212.07	24	167.5
32	261	111.16	25	81.5

Table 3.6: Legend of Table 3.5

`n`	matrix size
`nnz`	number of non null entries inside matrix
`nit`	number of iteration for convergence
`time`	Time for convergence
`Te`	Time for constructing finite element matrix
`np`	number of processor

In this example, we fix the matrix size (in term of finite element, we fix the mesh) and increase the number of processors used to solve the linear system. We saw that, when the number of processors increases, the time for solving the linear equation decreases, even if the number of iteration increases. This proves that, using pARMS as solver of linear systems coming from discretization of partial differential equation in **FreeFEM** can decrease drastically the total time of simulation.

Table 3.7: Meaning of `lparams` corresponding variables

Entries of `iparm`	Significations of each entries
`iparm[0]`	Krylov subspace methods Different values for this parameters are specify on Table 3.9
`iparm[1]`	Preconditionner Different preconditionners for this parameters are specify on Table 3.10
`iparm[2]`	Krylov subspace dimension in outer iteration: default value 30
`iparm[3]`	Maximum of iterations in outer iteration: default value 1000
`iparm[4]`	Number of level in arms when used
`iparm[5]`	Krylov subspace dimension in inner iteration: default value 3
`iparm[6]`	Maximum of iterations in inner iteration: default value 3
`iparm[7]`	Symmetric(=1 for symmetric) or unsymmetric matrix: default value 0(unsymmetric matrix)
`iparm[8]`	Overlap size between different subdomain: default value 0(no overlap)
`iparm[9]`	Scale the input matrix or not: Default value 1 (Matrix should be scaled)
`iparm[10]`	Block size in arms when used: default value 20
`iparm[11]`	lfil0 (ilut, iluk, and arms) : default value 20
`iparm[12]`	lfil for Schur complement const : default value 20
`iparm[13]`	lfil for Schur complement const : default value 20
`iparm[14]`	Multicoloring or not in ILU when used : default value 1
`iparm[15]`	Inner iteration : default value 0
`iparm[16]`	Print message when solving: default 0 (no message print) • 0: no message is print, • 1: Convergence informations like number of iteration and residual, • 2: Timing for a different step like preconditioner, • 3 : Print all informations

Table 3.8: Significations of `dparams` corresponding variables

Entries of `dparm`	Significations of each entries
`dparm[0]`	precision for outer iteration : default value 1e-08
`dparm[1]`	precision for inner iteration: default value 1e-2
`dparm[2]`	tolerance used for diagonal domain: : default value 0.1
`dparm[3]`	drop tolerance droptol0 (ilut, iluk, and arms) : default value 1e-2
`dparm[4]`	droptol for Schur complement const: default value 1e-2
`dparm[5]`	droptol for Schur complement const: default value 1e-2

Table 3.9: Krylov Solvers in pARMS

Values of `iparm[0]`	Krylov subspace methods
0	FGMRES (Flexible GMRES)
1	DGMRES (Deflated GMRES)
2	BICGSTAB

Table 3.10: Preconditionners in pARMS

Values of `iparm[1]`	Preconditionners type
0	additive Schwartz preconditioner with ilu0 as local preconditioner
1	additive Schwartz preconditioner with iluk as local preconditioner
2	additive Schwartz preconditioner with ilut as local preconditioner
3	additive Schwartz preconditioner with arms as local preconditioner
4	Left Schur complement preconditioner with ilu0 as local preconditioner
5	Left Schur complement preconditioner with ilut as local preconditioner
6	Left Schur complement preconditioner with iluk as local preconditioner
7	Left Schur complement preconditioner with arms as local preconditioner
8	Right Schur complement preconditioner with ilu0 as local preconditioner
9	Right Schur complement preconditioner with ilut as local preconditioner
10	Right Schur complement preconditioner with iluk as local preconditioner
11	Right Schur complement preconditioner with arms as local preconditioner
12	sch_gilu0, Schur complement preconditioner with global ilu0
13	SchurSymmetric GS preconditioner

Interfacing with HIPS

HIPS (*Hierarchical Iterative Parallel Solver*) is a scientific library that provides an efficient parallel iterative solver for very large sparse linear systems. HIPS is available as free software under the CeCILL-C licence.

HIPS implements two solver classes which are the iteratives class (GMRES, PCG) and the Direct class. Concerning preconditionners, HIPS implements a type of multilevel ILU. For further informations on those preconditionners see the HIPS documentation.

Tip: Laplacian 3D solved with HIPS

Let us consider the 3D Laplacian example inside **FreeFEM** package where after discretization we want to solve the linear equation with HIPS.

The following example is a Laplacian 3D using Hips as linear solver. We first load Hips solver at line 2. From line 7 to 18 we specify the parameters for the Hips solver and in line 82 we set these parameters in the linear solver.

In Table 3.11 results of running on Cluster Paradent of Grid5000 are reported. We can see in this running example the efficiency of parallelism.

```
load "msh3"
load "hips_FreeFem" //load Hips library

// Parameters
int nn = 10;
real zmin = 0, zmax = 1;
int[int] iparm(14);
real[int] dparm(6);
for (int iii = 0; iii < 14; iii++)
    iparm[iii] = -1;
for (int iii = 0; iii < 6; iii++)
    dparm[iii] = -1;
iparm[0] = 0; //use iterative solver
iparm[1] = 1; //PCG as Krylov method
iparm[4] = 0; //Matrix are symmetric
iparm[5] = 1; //Pattern are also symmetric
iparm[9] = 1; //Scale matrix
dparm[0] = 1e-13; //Tolerance to convergence
dparm[1] = 5e-4; //Threshold in ILUT
dparm[2] = 5e-4; //Threshold for Schur preconditionner

// Functions
func ue = 2*x*x + 3*y*y + 4*z*z + 5*x*y + 6*x*z + 1;
func uex = 4*x + 5*y + 6*z;
func uey = 6*y + 5*x;
func uez = 8*z + 6*x;
func f = -18.;

// Mesh
mesh Th2 = square(nn, nn);

int[int] rup = [0,2], rdown=[0, 1];
int[int] rmid=[1, 1, 2, 1, 3, 1, 4, 1];

mesh3 Th=buildlayers(Th2, nn, zbound=[zmin, zmax], reffacemid=rmid,
    reffaceup = rup, reffacelow = rdown);

// Fespace
fespace Vh2(Th2, P2);
Vh2 ux, uz, p2;

fespace Vh(Th, P2);
Vh uhe = ue;
cout << "uhe min =" << uhe[].min << ", max =" << uhe[].max << endl;
Vh u, v;
Vh F;

// Macro
macro Grad3(u) [dx(u), dy(u), dz(u)] //

// Problem
varf va (u, v)
    = int3d(Th)(
          Grad3(v)' * Grad3(u)
    )
```

(continues on next page)

(continued from previous page)

```
    + int2d(Th, 2)(
          u*v
    )
    - int3d(Th)(
          f*v
    )
    - int2d(Th, 2)(
          ue*v + (uex*N.x + uey*N.y + uez*N.z)*v
    )
    + on(1, u=ue);

varf l (unused, v) = int3d(Th)(f*v);

real cpu=clock();
matrix Aa = va(Vh, Vh);

F[] = va(0, Vh);

if (mpirank == 0){
    cout << "Size of A =" << Aa.n << endl;
    cout << "Non zero coefficients =" << Aa.nbcoef << endl;
    cout << "CPU TIME FOR FORMING MATRIX =" << clock()-cpu << endl;
}

set(Aa, solver=sparsesolver, dparams=dparm, lparams=iparm); //Set hips as linear solver

// Solve
u[] = Aa^-1*F[];

// Plot
plot(u);
```

Table 3.11: Legend of this table are give in Table 3.6

$n = 4 \times 10^6$ $nnz = 118 \times 10^6$ $Te = 221.34$		
np	nit	time
8	190	120.34
16	189	61.08
32	186	31.70
64	183	23.44

Tip:

Table 3.12: Significations of `lparams` corresponding to HIPS interface

Entries of `iparm`	Significations of each entries
`iparm[0]`	Strategy use for solving (Iterative=0 or Hybrid=1 or Direct=2). Defaults values are : Iterative
`iparm[1]`	Krylov methods. If iparm[0]=0, give type of Krylov methods: 0 for GMRES, 1 for PCG
`iparm[2]`	Maximum of iterations in outer iteration: default value 1000
`iparm[3]`	Krylov subspace dimension in outer iteration: default value 40
`iparm[4]`	Symmetric(=0 for symmetric) and 1 for unsymmetricmatrix: default value 1 (unsymmetric matrix)
`iparm[5]`	Pattern of matrix are symmetric or not: default value 0
`iparm[6]`	Partition type of input matrix: default value 0
`iparm[7]`	Number of level that use the HIPS locally consistentfill-in: Default value 2
`iparm[8]`	Numbering in indices array will start at 0 or 1: Default value 0
`iparm[9]`	Scale matrix. Default value 1
`iparm[10]`	Reordering use inside subdomains for reducingfill-in: Only use for iterative. Default value 1
`iparm[11]`	Number of unknowns per node in the matrix non-zeropattern graph: Default value 1
`iparm[12]`	This value is used to set the number of time the normalization is applied to the matrix: Default 2.
`iparm[13]`	Level of informations printed during solving: Default 5.
`iparm[14]`	HIPS_DOMSIZE Subdomain size

Table 3.13: Significations of `dparams` corresponding to HIPS interface

`dparm[0]`	HIPS_PREC: Relative residual norm: Default=1e-9
`dparm[1]`	HIPS_DROPTOL0: Numerical threshold in ILUT for interior domain (important : set 0.0 in HYBRID: Default=0.005)
`dparm[2]`	HIPS_DROPTOL1 : Numerical threshold in ILUT for Schur preconditioner: Default=0.005
`dparm[3]`	HIPS_DROPTOLE : Numerical threshold for coupling between the interior level and Schur: Default 0.005
`dparm[4]`	HIPS_AMALG : Numerical threshold for coupling between the interior level and Schur: Default=0.005
`dparm[5]`	HIPS_DROPSCHUR : Numerical threshold for coupling between the interior level and Schur: Default=0.005

Interfacing with HYPRE

Hypre (High Level Preconditioner) is a suite of parallel preconditioner developed at Lawrence Livermore National Lab.

There are two main classes of preconditioners developed in HYPRE: AMG (Algebraic MultiGrid) and Parasails (Parallel Sparse Approximate Inverse).

Now, suppose we want to solve $Ax = b$.

At the heart of AMG there is a series of progressively coarser (smaller) representations of the matrix A. Given an approximation $\hat{x}$ to the solution x, consider solving the residual equation $Ae = r$ to find the error e, where $r = b - A\hat{x}$. A fundamental principle of AMG is that it is an algebraically smooth error. To reduce the algebraically smooth errors

further, they need to be represented by a smaller defect equation (coarse grid residual equation) $A_c e_c = r_c$, which is cheaper to solve. After solving this coarse equation, the solution is then interpolated in fine grid represented here by matrix A. The quality of AMG depends on the choice of coarsening and interpolating operators.

The *sparse approximate inverse* approximates the inverse of a matrix A by a sparse matrix M. A technical idea to construct matrix M is to minimize the Frobenuis norm of the residual matrix $I - MA$. For more details on this preconditioner technics see [CHOW1997].

HYPRE implement three Krylov subspace solvers: GMRES, PCG and BiCGStab.

Tip: Laplacian 3D solved with HYPRE

Let us consider again the 3D Laplacian example inside **FreeFEM** package where after discretization we want to solve the linear equation with Hypre. The following example is a Laplacian 3D using Hypre as linear solver. This is the same example as Hips one, so we just show here the lines where we set some Hypre parameters.

We first load the Hypre solver at line 2. From line 6 to 18 we specifies the parameters to set to Hypre solver and in line 22 we set parameters to Hypre solver.

It should be noted that the meaning of the entries of these vectors is different from those of Hips. In the case of HYPRE, the meaning of differents entries of vectors `iparm` and `dparm` are given in Table 3.14 to Table 3.18.

In Table 3.19 the results of running on Cluster Paradent of Grid5000 are reported. We can see in this running example the efficiency of parallelism, in particular when AMG are use as preconditioner.

```
1  load "msh3"
2  load "hipre_FreeFem" //Load Hipre librairy
3
4  // Parameters
5  int nn = 10;
6  int[int] iparm(20);
7  real[int] dparm(6);
8  for (int iii = 0; iii < 20; iii++)
9      iparm[iii] = -1;
10 for (int iii = 0; iii < 6; iii++)
11     dparm[iii] = -1;
12 iparm[0] = 2; //PCG as krylov method
13 iparm[1] = 0; //AMG as preconditionner 2: if ParaSails
14 iparm[7] = 7; //Interpolation
15 iparm[9] = 6; //AMG Coarsen type
16 iparm[10] = 1; //Measure type
17 iparm[16] = 2; //Additive schwarz as smoother
18 dparm[0] = 1e-13; //Tolerance to convergence
19 dparm[1] = 5e-4; //Threshold
20 dparm[2] = 5e-4; //Truncation factor
21
22 ...
23
24 set(Aa, solver=sparsesolver, dparams=dparm, lparams=iparm);
```

Table 3.14: Definitions of common entries of `iparms` and `dparms` vectors for every preconditioner in HYPRE

`iparms[0]`	Solver identification: 0: BiCGStab, 1: GMRES, 2: PCG. Default=1
`iparms[1]`	Preconditioner identification: 0: BOOMER AMG, 1: PILUT, 2: Parasails, 3: Schwartz Default=0
`iparms[2]`	Maximum of iteration: Default=1000
`iparms[3]`	Krylov subspace dim: Default= 40
`iparms[4]`	Solver print info level: Default=2
`iparms[5]`	Solver log: Default=1
`iparms[6]`	Solver stopping criteria only for BiCGStab : Default=1
`dparms[0]`	Tolerance for convergence: Default=:math:*1.0e-11*

Table 3.15: Definitions of other entries of `iparms` and `dparms` if preconditioner is BOOMER AMG

`iparms[7]`	AMG interpolation type: Default=6
`iparms[8]`	Specifies the use of GSMG - geometrically smooth coarsening and interpolation: Default=1
`iparms[9]`	AMG coarsen type: Default=6
`iparms[10]`	Defines whether local or global measures are used: Default=1
`iparms[11]`	AMG cycle type: Default=1
`iparms[12]`	AMG Smoother type: Default=1
`iparms[13]`	AMG number of levels for smoothers: Default=3
`iparms[14]`	AMG number of sweeps for smoothers: Default=2
`iparms[15]`	Maximum number of multigrid levels: Default=25
`iparms[16]`	Defines which variant of the Schwartz method isused: 0: hybrid multiplicative Schwartz method (no overlap across processor boundaries) 1: hybrid additive Schwartz method (no overlap across processor boundaries) 2: additive Schwartz method 3: hybrid multiplicative Schwartz method (with overlap across processor boundaries) Default=1
`iparms[17]`	Size of the system of PDEs: Default=1
`iparms[18]`	Overlap for the Schwarz method: Default=1
`iparms[19]`	Type of domain used for the Schwarz method 0: each point is a domain 1: each node is a domain (only of interest in "systems" AMG) 2: each domain is generated by agglomeration (default)
`dparms[1]`	AMG strength threshold: Default=0.25
`dparms[2]`	Truncation factor for the interpolation: Default=1e-2
`dparms[3]`	Sets a parameter to modify the definition of strength for diagonal dominant portions of the matrix: Default=0.9
`dparms[5]`	Defines a smoothing parameter for the additive Schwartz method. Default=1

Table 3.16: Definitions of other entries of `iparms` and `dparms` if preconditioner is PILUT

`iparms[7]`	Row size in Parallel ILUT: Default=1000
`iparms[8]`	Set maximum number of iterations: Default=30
`dparms[1]`	Drop tolerance in Parallel ILUT: Default=1e-5

Table 3.17: Definitions of other entries of `iparms` and `dparms` if preconditioner is ParaSails

`iparms[7]`	Number of levels in Parallel Sparse Approximate inverse: Default=1
`iparms[8]`	Symmetric parameter for the ParaSails preconditioner: 0: nonsymmetric and/or indefinite problem, and nonsymmetric preconditioner 1: SPD problem, and SPD (factored) preconditioner 2: nonsymmetric, definite problem, and SPD (factored) preconditioner Default=0
`dparms[1]`	Filters parameters. The filter parameter is used to drop small nonzeros in the preconditioner, to reduce the cost of applying the preconditioner: Default=0.1
`dparms[2]`	Threshold parameter: Default=0.1

Table 3.18: Definitions of other entries of `iparms` and `dparms` if preconditionner is Schwartz

`iparms[7]`	Defines which variant of the Schwartz method isused: 0: hybrid multiplicative Schwartz method (no overlap across processor boundaries) 1: hybrid additive Schwartz method (no overlap across processor boundaries) 2: additive Schwartz method 3: hybrid multiplicative Schwartz method (with overlap across processor boundaries) Default=1
`iparms[8]`	Overlap for the Schwartz method: Default=1
`iparms[9]`	Type of domain used for the Schwartz method 0: each point is a domain 1: each node is a domain (only of interest in "systems" AMG) 2: each domain is generated by agglomeration (default)

Table 3.19: Convergence and time for solving linear system

$n = 4 \times 10^6$	$nnz = 13 \times 10^6$	$Te = 571.29$
np	AMG	
	nit	*time*
8	6	1491.83
16	5	708.49
32	4	296.22
64	4	145.64

Conclusion

With the different runs presented here, we wanted to illustrate the gain in time when we increase the number of processors used for the simulations. We saw that in every case the time for the construction of the finite element matrix is constant. This is normal because until now this phase is sequential in **FreeFEM**. In contrast, phases for solving the linear system are parallel. We saw on several examples presented here that when we increase the number of processors, in general we decrease the time used for solving the linear systems. But this is not true in every case. In several case, when we increase the number of processors the time to convergence also increases. There are two main reasons for this. First, the increase of processors can lead to the increase of volume of exchanged data across processors consequently increasing the time for solving the linear systems.

Furthermore, in decomposition domain type preconditioners, the number of processors generally corresponds to the number of sub domains. In subdomain methods, generally when we increase the number of subdomains we decrease convergence quality of the preconditioner. This can increase the time used for solving linear equations.

To end this, we should note that good use of the preconditioners interfaced in **FreeFEM** is empiric, because it is difficult to know what is a good preconditioner for some type of problems. Although, the efficiency of preconditioners sometimes depends on how its parameters are set. For this reason we advise the user to pay attention to the meaning of the parameters in the user guide of the iterative solvers interfaced in **FreeFEM**.

Domain decomposition

In the previous section, we saw that the phases to construct a matrix are sequential. One strategy to construct the matrix in parallel is to divide geometrically the domain into subdomains. In every subdomain we construct a local submatrix and after that we assemble every submatrix to form the global matrix.

We can use this technique to solve PDE directly in domain Ω. In this case, in every subdomains you have to define artificial boundary conditions to form consistent equations in every subdomains. After this, you solve equation in every subdomains and define a strategy to obtain the global solution.

In terms of parallel programming for **FreeFEM**, with MPI, this means that the user must be able to divide processors avaible for computation into subgroups of processors and also must be able to realize different type of communications in **FreeFEM** script. Here is a wrapper of some MPI functions.

Communicators and groups

Groups

`mpiGroup grpe(mpiGroup gp, KN_<long>)`: Create MPI_Group from existing group `gp` by given vector.

Communicators

Communicators is an abstract MPI object which allows MPI user to communicate across group of processors. Communicators can be Intra-communicators(involves a single group) or Inter-communicators (involves two groups). When we not specify type of communicator it will be Intra-communicators

mpiComm cc(mpiComm comm, mpiGroup gp): Creates a new communicator.

`comm` communicator(handle), `gp` group which is a subset of the group of `comm` (handle). Return new communicator

mpiComm cc(mpiGroup gp): Same as previous constructor but default `comm` here is `MPI_COMM_WORLD`.

mpiComm cc(mpiComm comm, int color, int key): Creates new communicators based on `colors` and `key`. This constructor is based on MPI_Comm_split routine of MPI.

mpiComm cc(MPIrank p, int key): Same constructor than the last one.

Here `colors` and `comm` is defined in `MPIrank`. This constructor is based on `MPI_Comm_split` routine of MPI.

Tip: Split communicator

```
mpiComm comm(mpiCommWorld, 0, 0);
int color = mpiRank(comm)%2;
mpiComm ccc(processor(color, comm), 0);
mpiComm qpp(comm, 0, 0);
mpiComm cp(ccc, color, 0);
```

mpiComm cc(mpiComm comm, int high): Creates an intracommunicator from an intercommunicator. `comm` intercommunicator, `high`.

Used to order the groups within `comm` (logical) when creating the new communicator. This constructor is based on `MPI_Intercomm_merge` routine of MPI.

mpiComm cc(MPIrank p1, MPIrank p2, int tag): This constructor creates an intercommuncator from two intracommunicators. `p1` defined local (intra)communicator and rank in `local_comm` of leader (often 0) while `p2` defined remote communicator and rank in `peer_comm` of remote leader (often 0). `tag` Message tag to use in constructing intercommunicator. This constructor is based on `MPI_Intercomm_create`.

Tip: Merge

```
mpiComm comm, cc;
int color = mpiRank(comm)%2;
int rk = mpiRank(comm);
int size = mpiSize(comm);
cout << "Color values: " << color << endl;
mpiComm ccc(processor((rk<size/2), comm), rk);
mpiComm cp(cc, color, 0);
int rleader;
if (rk == 0){ rleader = size/2; }
else if (rk == size/2){ rleader = 0; }
else{ rleader = 3; }
mpiComm qqp(processor(0, ccc), processor(rleader, comm), 12345);
int aaa = mpiSize(ccc);
cout << "Number of processor: " << aaa << endl;
```

Process

In **FreeFEM** we wrap MPI process by function call `processor` which create internal **FreeFEM** object call `MPIrank`. This mean that do not use `MPIrank` in **FreeFEM** script.

`processor(int rk)`: Keep process rank inside object `MPIrank`. Rank is inside `MPI_COMM_WORLD`.

`processor(int rk, mpiComm cc)` and `processor(mpiComm cc, int rk)` process rank inside communicator cc.

`processor(int rk, mpiComm cc)` and `processor(mpiComm cc, int rk)` process rank inside communicator cc.

`processorblock(int rk)`: This function is exactlly the same than `processor(int rk)` but is use in case of blocking communication.

`processorblock(int rk, mpiComm cc)`: This function is exactly the same as `processor(int rk, mpiComm cc)` but uses a synchronization point.

Points to Points communicators

In **FreeFEM** you can call MPI points to points communications functions.

`Send(processor(int rk, mpiComm cc), Data D)` : Blocking send of `Data D` to processor of `rank rk` inside communicator `cc`. Note that `Data D` can be: `int`, `real`, `complex`, `int[int]`, `real[int]`, `complex[int]`, `Mesh`, `Mesh3`, `Matrix`.

`Recv(processor(int rk, mpiComm cc), Data D)`: Receive `Data D` from process of rank `rk` in communicator `cc`.

Note that `Data D` can be: `int`, `real`, `complex`, `int[int]`, `real[int]`, `complex[int]`, `Mesh`, `Mesh3`, `Matrix` and should be the same type than corresponding send.

`Isend(processor(int rk, mpiComm cc), Data D)` : Non blocking send of `Data D` to processor of `rank rk` inside communicator `cc`.

Note that `Data D` can be: `int`, `real`, `complex`, `int[int]`, `real[int]`, `complex[int]`, `mesh`, `mesh3`, `matrix`.

`Recv(processor(int rk, mpiComm cc), Data D)`: Receive corresponding to send.

Global operations

In **FreeFEM** you can call MPI global communication functions.

`broadcast(processor(int rk, mpiComm cc), Data D)`: Process `rk` Broadcast `Data D` to all process inside `communicator cc`. Note that `Data D` can be: `int`, `real`, `complex`, `int[int]`, `real[int]`, `complex[int]`, `Mesh`, `Mesh3`, `Matrix`.

`broadcast(processor(int rk), Data D)`: Process `rk` Broadcast `Data D` to all process inside `MPI_COMM_WORLD`. Note that `Data D` can be: `int`, `real`, `complex`, `int[int]`, `real[int]`, `complex[int]`, `Mesh`, `Mesh3`, `Matrix`.

`mpiAlltoall(Data a, Data b)`: Sends `data a` from all to all processes. Receive buffer is `Data b`. This is done inside communicator `MPI_COMM_WORLD`.

`mpiAlltoall(Data a, Data b, mpiComm cc)`: Sends `data a` from all to all processes. Receive buffer is `Data b`. This is done inside communicator `cc`.

`mpiGather(Data a, Data b, processor(mpiComm, int rk)`: Gathers together values `Data a` from a group of processes. Process of rank `rk` get data on communicator `rk`. This function is like `MPI_Gather`.

`mpiAllgather(Data a, Data b)`: Gathers `Data a` from all processes and distribute it to all in `Data b`. This is done inside communicator `MPI_COMM_WORLD`. This function is like `MPI_Allgather`.

`mpiAllgather(Data a, Data b, mpiComm cc)`: Gathers `Data a` from all processes and distribute it to all in `Data b`. This is done inside `communicator cc`. This function is like `MPI_Allgather`.

`mpiScatter(Data a,Data b,processor(int rk, mpiComm cc))`: Sends `Data a` from one process whith rank `rk` to all other processes in group represented by communicator `mpiComm cc`.

`mpiReduce(Data a, Data b, processor(int rk, mpiComm cc), MPI_Op op)` Reduces values `Data a` on all processes to a single value `Data b` on process of rank `rk` and communicator `cc`.

Operation use in reduce is: `MPI_Op op` which can be: `mpiMAX`, `mpiMIN`, `mpiSUM`, `mpiPROD`, `mpiLAND`, `mpiLOR`, `mpiLXOR`, `mpiBAND`, `mpiBXOR`, `mpiMAXLOC`, `mpiMINLOC`.

Note that, for all global operations, only `int[int]` and `real[int]` are data type take in account in **FreeFEM**.

HPDDM solvers

Real valued problems (diffusion, heat, elasticity and Stokes) and complex valued problems (Maxwell and Helmholtz) are given in both 2D and 3D. We detail here the 3D elasticity problem and the 3D time-dependent heat problem.

Tip: Elasticity 3D

A three dimensional elasticity problem is defined. The solver is a domain decomposition method. Domain decomposition methods are a natural framework for parallel computers. The scripts run on multicores computers (from 2 to tens of thousands of cores). Recall that like in any MPI code the number of MPI processes, `mpisize`, is given in the command line via the option `-np`. We focus on the script `Elasticity3D.edp` but the other scripts have the same structure. The command line to run the example on four processes with `ffglut` visualization is: `ff-mpirun -np 4 Elasticity3D.edp -glut ffglut`

```
load "hpddm" //load HPDDM plugin
macro partitioner()metis//metis, scotch, or parmetis
macro dimension()3//2D or 3D
```

(continues on next page)

(continued from previous page)

```
macro vectorialfe()P1//
include "macro_ddm.idp" //additional DDM functions

// Macro
macro def(i)[i, i#B, i#C] //vector field definition
macro init(i)[i, i, i] //vector field initialization

real Sqrt = sqrt(2.0);
macro epsilon(u) [dx(u), dy(u#B), dz(u#C),
    (dz(u#B) + dy(u#C)) / Sqrt,
    (dz(u) + dx(u#C)) / Sqrt,
    (dy(u) + dx(u#B)) / Sqrt] //
macro div(u) (dx(u) + dy(u#B) + dz(u#C)) //

// Parameters
real f = -9000.0;
real strain = 100.0;
real Young = 2.0e11; // steel
real poisson = 0.35;

func Pk = [vectorialfe, vectorialfe, vectorialfe];

string deflation = getARGV("-deflation", "geneo"); //coarse space construction
int overlap = getARGV("-overlap", 1); //geometric overlap between subdomains
int fakeInterface = getARGV("-interface", 10); //interface between subdomains
int s = getARGV("-split", 1); //refinement factor
int p = getARGV("-hpddm_master_p", 1);

mpiComm comm;
bool excluded = splitComm(mpiCommWorld, p, comm, topology = getARGV("-hpddm_master_
↪topology", 0), exclude = (usedARGV("-hpddm_master_exclude") != -1));

// Display
if (verbosity > 0 && mpirank == 0){
    cout << " --- " << mpirank << "/" << mpisize;
    cout << " - Elasticity3D.edp - input parameters: refinement factor = " << s << " -
↪ overlap = " << overlap << endl;
}

// Mesh
int[int] LL = [2, 3, 2, 1, 2, 2];
meshN ThBorder, Th = cube(1, 1, 1, [x, y, z]);
fespace Wh(Th, Pk); //local finite element space

int[int] arrayIntersection; //ranks of neighboring subdomains
int[int][int] restrictionIntersection(0); //local-to-neighbors renumbering
real[int] D; //partition of unity
{
    meshN ThGlobal = cube(10*getARGV("-global", 5), getARGV("-global", 5), getARGV("-
↪global", 5), [10*x, y, z], label=LL); //global mesh
    build(Th, ThBorder, ThGlobal, fakeInterface, s, overlap, D, arrayIntersection,␣
↪restrictionIntersection, Wh, Pk, comm, excluded, 3)
}

// Problem
real tmp = 1.0 + poisson;
real mu = Young / (2.0 * tmp);
```

(continues on next page)

(continued from previous page)

```
real lambda = Young * poisson / (tmp * (1.0 - 2.0 * poisson));
real[int] rhs; //local right-hand side
matrix<real> Mat; //local operator
{ //local weak form
    meshN ThAugmented = Th + ThBorder;
    varf vPb (def(u), def(v))
        = intN(ThAugmented)(
              lambda * div(u) * div(v)
            + 2.0 * mu * (epsilon(u)' * epsilon(v))
        )
        + intN(ThAugmented)(
              f * vC
        )
        + on(1, u=0.0, uB=0.0, uC=0.0)
        ;

    fespace WhAugmented(ThAugmented, Pk);
    Mat = vPb(WhAugmented, WhAugmented, tgv=-1);
    real[int] rhsFull = vPb(0, WhAugmented, tgv=-1);
    matrix R = interpolate(Wh, WhAugmented);
    renumbering(Mat, R, rhsFull, rhs);
}
ThBorder = cube(1, 1, 1, [x, y, z]);

dschwarz A(Mat, arrayIntersection, restrictionIntersection, scaling = D);

set(A, sparams = "-hpddm_schwarz_method ras -hpddm_schwarz_coarse_correction balanced -hpddm_variant right -hpddm_verbosity 1 -hpddm_geneo_nu 10");

matrix<real> Opt; //local operator with optimized boundary conditions
dpair ret;
{
    int solver = getOption("schwarz_method");
    if (solver == 1 || solver == 2 || solver == 4){ //optimized Schwarz methods
        fespace Ph(Th, P0);
        real kZero = getARGV("-kZero", 10.0);
        Ph transmission = 2 * kZero * mu * (2 * mu + lambda) / (lambda + 3 * mu);
        varf vOptimized (def(u), def(v))
            = intN(Th)(
                  lambda * div(u) * div(v)
                + 2.0 * mu * (epsilon(u)' * epsilon(v))
            )
            + intN1(Th, fakeInterface)(
                  transmission * (def(u)' * def(v))
            )
            + on(1, u=0.0, uB=0.0, uC=0.0)
            ;
        Opt = vOptimized(Wh, Wh, tgv=-1);
    }
    if (mpisize > 1 && isSetOption("schwarz_coarse_correction")){ //two-level Schwarz methods
        if(excluded)
            attachCoarseOperator(mpiCommWorld, A);
        else {
            varf vPbNoPen (def(u), def(v))
                = intN(Th)(
                      lambda * div(u) * div(v)
```

(continues on next page)

(continued from previous page)

```
                    + 2.0 * mu * (epsilon(u)' * epsilon(v))
                )
                + on(1, u=0.0, uB=0.0, uC=0.0)
                ;
            matrix<real> noPen = vPbNoPen(Wh, Wh, solver=CG);
            if(deflation == "geneo") //standard GenEO, no need for RHS -> deduced from LHS (Neumann matrix)
                attachCoarseOperator(mpiCommWorld, A, A=noPen, ret=ret);
            else if(deflation == "dtn"){
                varf vMass (def(u), def(v)) = intN1(Th, fakeInterface)(u * v);
                matrix<real> massMatrix = vMass(Wh, Wh, solver=CG);
                attachCoarseOperator(mpiCommWorld, A, A=noPen, B=massMatrix, pattern=Opt, ret=ret);
            }
            else if(deflation == "geneo-2") //GenEO-2 for optimized Schwarz methods, need for RHS (LHS is still Neumann matrix)
                attachCoarseOperator(mpiCommWorld, A, A=noPen, B=Opt, pattern=Opt, ret=ret);
        }
    }
}


// Solve
Wh<real> def(u); //local solution

if(Opt.n > 0) //optimized Schwarz methods
    DDM(A, u[], rhs, excluded=excluded, ret=ret, O=Opt);
else
    u[] = A^-1 * rhs;

// Error
real[int] err(u[].n);
err = A * u[]; //global matrix-vector product
err -= rhs;

// Plot
plotMPI(Th, u[], "Global solution", Pk, def, real, 3, 1)
plotMPI(Th, err, "Global residual", Pk, def, real, 3, 1)
real alpha = 2000.0;
meshN ThMoved = movemesh3(Th, transfo = [x + alpha*u, y + alpha*uB, z + alpha*uC]);
u[] = mpirank;
plotMPI(ThMoved, u[], "Global moved solution", Pk, def, real, 3, 1)
```

The macro `build` is of particular interest since it handles the data distribution among the `mpisize` MPI processes with the following steps:

- The initial mesh `ThGlobal` is partitioned by process 0 into `mpisize` submeshes
- **The partition is broadcasted to every process i for $0 < i <$ `mpisize`.** From then on, all tasks are parallel.
- **Each process creates the local submesh `Th` (if the refinement factor `s` defined via the option `-split` is larger than 1, each**
  The number of extra layers added to the initial partition is monitored by the option `overlap`.
- Connectivity structures are created
- `D` is the diagonal of the local partition of unity (see *Distributed vectors in HPDDM*)
- `arrayIntersection` is the list of neighbors of the current subdomain

- For `j` in `arrayIntersection`, `restrictionIntersection[j]` is the list of the degrees of freedom that belong to the intersection of the current subdomain with its neighbor `j`.

Then, the variational formulation `vPb` of a three dimensional elasticity problem is used to assemble a local matrix `Mat`. This matrix along with `D`, `arrayIntersection` and `restrictionIntersection` are arguments for the constructor of the distributed matrix `A`. This is enough to solve the problem with a one-level additive Schwarz method which can be either ASM or RAS.

For some problems it is interesting to use optimized interface conditions. When there are many subdomains, it is usually profitable to add a second level to the solver. Options are set in the sequel of the script:

```
set(A, sparams="-hpddm_schwarz_method ras -hpddm_schwarz_coarse_correction balanced -
↪hpddm_variant right -hpddm_verbosity 1 -hpddm_geneo_nu 10");
```

In the above line, the first option selects the one-level preconditioner `ras` (possible choices are `ras`, `oras`, `soras`, `asm`, `osm` or `none`), the second option selects the correction formula for the second level here `balanced` (possible options are `deflated`, `additive` or `balanced`), the third option selects right preconditioning, the fourth one is verbosity level of HPDDM (different from the one of **FreeFEM**), the fifth one prints all possible options of HPPDM and the last one specifies the number of coarse degrees of freedom per subdomain of the GENEO coarse space. All other options of HPDDM library can be selected via the **FreeFEM** function `set`.

In the last part of the script, the global linear system is solved by the domain decomposition method defined above.

```
// Solve
Wh<real> def(u); //local solution

if(Opt.n > 0) //optimized Schwarz methods
DDM(A, u[], rhs, excluded=excluded, ret=ret, O=Opt);
else
u[] = A^-1 * rhs;
```

Time dependent problem

Tip: Heat 3D

A three dimensional heat problem

$$\frac{\partial u}{\partial t} - \Delta u = 1, \quad u(0,\cdot) := 0 \text{ in } \Omega\,.$$

is discretized by an implicit Euler scheme. At each time step n, we shall seek $u^n(x,y,z)$ satisfying for all $w \in H^1(\Omega)$:

$$\int_\Omega \frac{u^n - u^{n-1}}{\delta t}\, w + \nabla u^n \nabla w = \int_\Omega w, \quad u^0 := 0 \text{ in } \Omega\,.$$

so that at each time step a linear system:

$$(M + dt * K)u^n[] = M * u^{n-1}[] + \delta t * F$$

is solved by a domain decomposition method where M is the mass matrix and K is the rigidity matrix. In order to save computational efforts, the domain decomposition method preconditioner is built only once and then reused for all subsequent solves with matrix $A := M + dt * K$. The distributed matrix vector product with matrix M is made through the call to the function `dmv` using the partition of unity associated to matrix A.

```
load "hpddm" //load HPDDM plugin
macro partitioner()metis//metis, scotch, or parmetis
macro dimension()3//2D or 3D
include "macro_ddm.idp" //additional DDM functions

// Macro
macro def(i)i //scalar field definition
macro init(i)i //scalar field initialization
macro grad(u) [dx(u), dy(u), dz(u)] //three-dimensional gradient

// Parameters
func Pk = P2; //finite element space

string deflation = getARGV("-deflation", "geneo"); //coarse space construction
int overlap = getARGV("-overlap", 1); //geometric overlap between subdomains
int fakeInterface = getARGV("-interface", 10); //interface between subdomains
int s = getARGV("-split", 1); //refinement factor
real dt = getARGV("-dt", 0.01); //time step
int iMax = getARGV("-iMax", 10); //number of iterations

mpiComm comm;
int p = getARGV("-hpddm_master_p", 1);
bool excluded = splitComm(mpiCommWorld, p, comm, topology = getARGV("-hpddm_master_
↪topology", 0), exclude = (usedARGV("-hpddm_master_exclude") != -1));

// Display
if (verbosity > 0 && mpirank == 0){
    cout << " --- " << mpirank << "/" << mpisize;
    cout << " - Heat3D.edp - input parameters: refinement factor = " << s << " -␣
↪overlap = " << overlap << endl;
}

// Mesh
int[int] LL = [1, 2, 1, 1, 1, 1];
meshN ThBorder, Th = cube(1, 1, 1, [x, y, z]);
fespace Wh(Th, Pk); //local finite element space
int[int] arrayIntersection; //ranks of neighboring subdomains
int[int][int] restrictionIntersection(0); //local-to-neighbors renumbering
real[int] D; //partition of unity
{
    meshN ThGlobal = cube(getARGV("-global", 10), getARGV("-global", 10), getARGV("-
↪global", 10), [x, y, z], label=LL); //global mesh
    build(Th, ThBorder, ThGlobal, fakeInterface, s, overlap, D, arrayIntersection,␣
↪restrictionIntersection, Wh, Pk, comm, excluded)
}

// Problem
real[int] rhs; // local right-hand side
matrix<real> Mat; //local operator
matrix<real> M; //local mass matrix
{ //local weak form
    meshN ThAugmented = Th + ThBorder;
    varf vPb (u, v)
        = intN(ThAugmented)(
              u * v
            + dt * (grad(u)' * grad(v))
        )
```

(continues on next page)

(continued from previous page)

```
        + intN(ThAugmented)(
              dt * v
        )
        + on(1, u=0.0)
        ;
    fespace WhAugmented(ThAugmented, Pk);
    Mat = vPb(WhAugmented, WhAugmented, tgv=-1);
    real[int] rhsFull = vPb(0, WhAugmented, tgv=-1);
    matrix R = interpolate(Wh, WhAugmented);
    varf vPbM (u, v) = intN(ThAugmented)(u * v);
    M = vPbM(WhAugmented, WhAugmented);
    renumbering(M, R, rhsFull, rhs);
    renumbering(Mat, R, rhsFull, rhs);
}
ThBorder = cube(1, 1, 1, [x, y, z]);

dschwarz A(Mat, arrayIntersection, restrictionIntersection, scaling=D);

matrix<real> Opt; //local operator with optimized boundary conditions
dpair ret;
{
    int solver = getOption("schwarz_method");
    if (solver == 1 || solver == 2 || solver == 4){ //optimized Schwarz methods
        fespace Ph(Th, P0);
        real kZero = getARGV("-kZero", 10.0);
        Ph transmission = kZero;
        varf vOptimized (u, v)
            = intN(Th)(
                  u * v
                + dt * (grad(u)' * grad(v))
            )
            + intN1(Th, fakeInterface)(
                  transmission * (u * v)
            )
            + on(1, u=0.0)
            ;
        Opt = vOptimized(Wh, Wh, tgv=-1);
    }
    if (mpisize > 1 && isSetOption("schwarz_coarse_correction")){ //two-level Schwarz methods
        if(excluded)
            attachCoarseOperator(mpiCommWorld, A);
        else {
            varf vPbNoPen (u, v)
                = intN(Th)(
                      u * v
                    + dt * (grad(u)' * grad(v))
                )
                + on(1, u=0.0)
                ;
            matrix<real> noPen = vPbNoPen(Wh, Wh, solver=CG);
            if(deflation == "geneo") //standard GenEO, no need for RHS -> deduced from LHS (Neumann matrix)
                attachCoarseOperator(mpiCommWorld, A, A=noPen, ret = ret);
            else if(deflation == "dtn") {
                varf vMass (def(u), def(v)) = intN1(Th, fakeInterface)(u * v);
                matrix<real> massMatrix = vMass(Wh, Wh, solver=CG);
```

(continues on next page)

(continued from previous page)

```
                attachCoarseOperator(mpiCommWorld, A, A=noPen, B=massMatrix, pattern=Opt, ret=ret);
            }
            else if(deflation == "geneo-2") //GenEO-2 for optimized Schwarz methods, need for RHS (LHS is still Neumann matrix)
                attachCoarseOperator(mpiCommWorld, A, A=noPen, B=Opt, pattern=Opt, ret=ret);
        }
    }
}

// Solve
set(A, sparams="-hpddm_reuse_preconditioner=1");
Wh<real> def(u) = init(0.0); //local solution
for (int i = 0; i < iMax; ++i){
    real[int] newRhs(rhs.n);
    dmv(A, M, u[], newRhs); //newRhs = M * u[]
    newRhs += rhs;

    if (Opt.n > 0) //optimized Schwarz methods
    DDM(A, u[], newRhs, excluded=excluded, ret=ret, O=Opt);
    else
    u[] = A^-1 * newRhs;

    plotMPI(Th, u[], "Global solution", Pk, def, real, 3, 0)
}
```

Distributed vectors in HPDDM

We give here some hints on the way vectors are distributed among np processes when using **FreeFEM** interfaced with HPDDM. The set of degrees of freedom $\mathcal{N}$ is decomposed into np overlapping sets $(\mathcal{N}_i)_{1\leq i\leq np}$.

A MPI-process is in charge of each subset. Let $n := \#\mathcal{N}$ be the number of degrees of freedom of the global finite element space. Let R_i denote the restriction operator from $\mathbb{R}^n$ onto $\mathbb{R}^{\#\mathcal{N}_i}$. We have also defined local diagonal matrices $D_i \in \mathbb{R}^{\#\mathcal{N}_i} \times \mathbb{R}^{\#\mathcal{N}_i}$ so that we have a partition of unity at the algebraic level:

$$\mathbf{U} = \sum_{i=1}^{np} R_i^T \, D_i \, R_i \, \mathbf{U} \quad \forall \, \mathbf{U} \in \mathbb{R}^n \, . \tag{3.31}$$

A global vector $\mathbf{U} \in \mathbb{R}^n$ is actually not stored. Rather, it is stored in a distributed way. Each process i, $1 \leq i \leq N$, stores the local vector $\mathbf{U}_i := R_i \mathbf{U} \in \mathbb{R}^{\#\mathcal{N}_i}$.

It is important to ensure that the result of all linear algebra operators applied to this representation are coherent.

As an example, consider the scalar product of two distributed vectors $\mathbf{U}, \mathbf{V} \in \mathbb{R}^n$. Using the partition of unity (3.31), we have:

$$\begin{aligned}(\mathbf{U}, \mathbf{V}) = \left(\mathbf{U}, \sum_{i=1}^{np} R_i^T D_i R_i \mathbf{V}\right) &= \sum_{i=1}^{np} (R_i \mathbf{U}, D_i R_i \mathbf{V}) \\ &= \sum_{i=1}^{np} (\mathbf{U}_i, D_i \mathbf{V}_i) \, .\end{aligned}$$

Thus, the formula for the scalar product is:

$$(\mathbf{U}, \mathbf{V}) = \sum_{i=1}^{np} (R_i \mathbf{U}, D_i R_i \mathbf{V}) \,.$$

Local scalar products are performed concurrently. Thus, the implementation is parallel except for the sum which corresponds to a `MPI_Reduce` call across the np MPI processes.

Note also that the implementation relies on the knowledge of a partition of unity so that the **FreeFEM** syntax is `dscalprod(D, u, v)`.

A `axpy` procedure $y \leftarrow \alpha\, x + y$ for $x, y \in \mathbb{R}^n$ and $\alpha \in \mathbb{R}$ is easily implemented concurrently for distributed vectors in the form:

$$y_i \leftarrow \alpha\, x_i + y_i \,, \forall\, 1 \leq i \leq np \,.$$

The matrix vector product is more involved and details are given in the SIAM book An Introduction to Domain Decomposition Methods: algorithms, theory and parallel implementation and even more details are given in P. Jolivet's PhD manuscrit.

3.7 Plugins

3.7.1 gsl

The interface with `gsl` spline is available in **FreeFEM**, the seven kind of spline are

0. `gslinterpcspline`: default type of spline
1. `gslinterpakima`
2. `gslinterpsteffen`
3. `gslinterplinear`
4. `gslinterppolynomial`
5. `gslinterpcsplineperiodic`
6. `gslinterpakimaperiodic`

A brief wing example given all the syntax:

```
load "gsl"

// Parameters
int n = 10;
real[int, int] dspline(2,n+1); //data points to define the spline
for(int i = 0; i <= n; ++i){ //set data points
   real xx = square(real(i)/n);
   real yy = sin(xx*pi*2);
   dspline(0, i) = xx;
   dspline(1, i) = yy;
}

// GSL splines
gslspline spline1(gslinterpcspline, dspline); //define the spline1
gslspline spline11(dspline); //define the spline11
gslspline spline2(gslinterpsteffen, dspline); //define the spline2
```

(continues on next page)

(continued from previous page)

```
gslspline spline3(gslinterpcspline, dspline(0, :), dspline(1, :));
gslspline spline33(dspline(0, :), dspline(1, :)); //define the spline3
spline1 = spline2; //copy spline2 in spline1

real t = 1.;
real s1 = spline1(t); //evaluate the function spline1 at t
cout << "spline1(t) = " << s1 << endl;
real ds1 = spline1.d(t); //evaluate the derivative of function spline1 at t
cout << "spline1.d(t) = " << ds1 << endl;
real dds1 = spline1.dd(t); //evaluate the second derivative of function spline1 at t
cout << "spline1.dd(t) = " << dds1 << endl;
```

This can be usefull to build function from data value.

The list of all `gsl` functions and the **FreeFEM** equivalent is available in the *Language references* (same names without _).

3.7.2 ffrandom

Plugin to linux `random` functions.

The range of the random generator is from 0 to $(2^{31}) - 1$.

```
load "ffrandom"

srandomdev(); //set a true random seed
//warning: under window this command
//change the seed by randinit(random()) so all
//FreeFEM random function are changed

int maxrang = 2^31 - 1;
cout << " max range " << maxrang << endl;

cout << random() << endl;
cout << random() << endl;
cout << random() << endl;

srandom(10);
cout << random() << endl;
cout << random() << endl;
cout << random() << endl;
```

3.7.3 mmap / semaphore

The idea is just try to use Interprocess communication using POSIX Shared Memory in Linux.

We build a small library `libff-mmap-semaphore.c` and `libff-mmap-semaphore.h` to easily interface.

- mmap - allocate memory, or map files or devices into memory
- semaphore - allow processes and threads to synchronize their actions

  A semaphore is an integer whose value is never allowed to fall below zero. Two operations can be performed on semaphores: increment the semaphore value by one (`sem_post`); and decrement the semaphore value by one (`sem_wait`).

If the value of a semaphore is currently zero, then a `sem_wait` operation will block until the value becomes greater than zero.

The functions of library

First the `semaphore` interface to make synchronization:

- `typedef struct FF_P_sem *ff_Psem;` the pointer to data structure
- `ff_Psem ffsem_malloc();` malloc an empty data structure
- `void ffsem_del(ff_Psem sem);` clean and free the pointer
- `void ffsem_destroy(ff_Psem sem);` clean, close the data structure
- `void ffsem_init0(ff_Psem sem);` make a correct empty of the data structure
- `void ffsem_init(ff_Psem sem,const char *nmm, int crea);` create or use a new semaphore
- `long ffsem_post(ff_Psem sem);` `nlocked`, the value of the semaphore is incremented, and all threads which are waiting on the semaphore are awakened
- `long ffsem_wait(ff_Psem sem);` the semaphore referenced by `sem` is locked. When calling `sem_wait()`, if the semaphore's value is zero, the calling thread will block until the lock is acquired or until the call is interrupted by a signal.

  Alternatively, the `sem_trywait()` function will fail if the semaphore is already locked, rather than blocking on the semaphore
- `long ffsem_trywait(ff_Psem p);`

Secondly, the `mmap` functions:

- `typedef struct FF_P_mmap *ff_Pmmap;` the pointer to data structure
- `ff_Psem ffmmap_malloc();` malloc an empty data structure
- `void ffmmap_del(ff_Pmmap p);` clean and free the pointer
- `void ffmmap_destroy(ff_Pmmap p);` clean, close the data structure
- `void ffmmap_init0(ff_Pmmap p);` make a correct empty of the data structure
- `long ffmmap_msync(ff_Pmmap p, long off, long ln);` call writes modified whole pages back to the filesystem and updates the file modification time. Only those pages containing `addr` and `len-1` succeeding locations will be examined.
- `void ffmmap_init(ff_Pmmap p, const char *nmm, long len);` allocate memory, or map files or devices into memory.
- `long ffmmap_read(ff_Pmmap p, void *t, size_t n, size_t off);` read `n` bytes from the `mmap` at memory `off` in pointer `t`.
- `long ffmmap_write(ff_Pmmap p, void *t, size_t n, size_t off);` write `n` bytes to the `mmap` at memory `off` in pointer `t`.

The **FreeFEM** corresponding functions:

- `Pmmap sharedata(filename, 1024);` new type to store the `mmap` informations of name store in string `filename` with 1024 is the size the `sharedata` zone and file.
- `Psemaphore smff("ff-slave", creat);` new type to store the semaphore of name `ff-slave` where `creat` is a boolean to create or use a existing semaphore.

- `Wait(sem)` the semaphore referenced by `sem` is locked. When calling `Wait(sem)`, if the semaphore's value is zero, the calling thread will block until the lock is acquired or until the call is interrupted by a signal. Alternatively, the `trywait(sem)` function will fail if the semaphore is already locked, rather than blocking on the semaphore.
- `Post(sem)` the semaphore referenced by `sem` is unlocked, the value of the semaphore is incremented, and all threads which are waiting on the semaphore are awakened.
- `Read(sharedata ,offset, data);` read the variable `data` from the place `offset` in `sharedata` mmap.
- `Write(sharedata, offset, data);` write the variable `data` at the place `offset` in `sharedata` mmap.

The full example:

The `FFMaster.c` file:

```
#include "libff-mmap-semaphore.h"
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
ff_Psem sem_ff, sem_c; //the semaphore for mutex

int main(int argc, const char ** argv)
{
   int debug = 0;
   ff_Pmmap shd;
   double cff, rff;
   long status;
   int i;
   if (argc > 1) debug = atoi(argv[1]);
   ff_mmap_sem_verb = debug;

   sem_ff = ffsem_malloc();
   sem_c = ffsem_malloc();
   shd = ffmmap_malloc();

   ffsem_init(sem_ff, "ff-slave1", 1);
   ffsem_init(sem_c, "ff-master1", 1);
   ffmmap_init(shd, "shared-data", 1024);

   status = 1;
   ffmmap_write(shd, &status, sizeof(status), 8);
   ffmmap_msync(shd, 0, 32);

   char ff[1024];
   sprintf(ff, "FreeFem++ FFSlave.edp -nw -ns -v %d&", debug);
   system(ff); //lauch FF++ in batch no graphics
   if(debug) printf("cc: before wait\n");

   if(debug) printf("cc: before wait 0 ff\n");
   ffsem_wait(sem_ff);

   for (i = 0; i < 10; ++i){
      printf(" iter : %d \n", i);
      cff = 10+i;
      ffmmap_write(shd, &cff, sizeof(cff), 0);
      ffsem_post(sem_c);
```

(continues on next page)

(continued from previous page)

```

      if(debug) printf(" cc: before wait 2\n");
      ffsem_wait(sem_ff);
      ffmmap_read(shd, &rff, sizeof(rff), 16);
      printf(" iter = %d rff= %f\n", i, rff);
   }

   status = 0; //end
   ffmmap_write(shd, &status, sizeof(status), 8);
   ffsem_post(sem_c);
   printf("End Master \n");
   ffsem_wait(sem_ff);
   ffsem_del(sem_ff);
   ffsem_del(sem_c);
   ffmmap_del(shd);
   return 0;
}
```

The `FFSlave.edp` file:

```
load "ff-mmap-semaphore"

Psemaphore smff("ff-slave1", 0);
Psemaphore smc("ff-master1", 0);
Pmmap sharedata("shared-data", 1024);
if (verbosity < 4) verbosity = 0;

// Mesh
mesh Th = square(10, 10);
int[int] Lab = [1, 2, 3, 4];

// Fespace
fespace Vh(Th, P1);
Vh u, v;

// Macro
macro grad(u) [dx(u), dy(u)] //

int status = 1;
cout << " FF status = " << status << endl;
real cff, rff;

// Problem
problem Pb (u, v)
   = int2d(Th)(
        grad(u)'*grad(v)
   )
   - int2d(Th)(
        cff*v
   )
   + on(Lab, u=0)
   ;

if (verbosity > 9) cout << " FF: before FF post\n";
Post(smff); //unlock master end init

while (1){
```

(continues on next page)

(continued from previous page)

```
    if (verbosity > 9) cout << " FF: before FF wait \n";
    Wait(smc); //wait from cint write ok
    Read(sharedata, 0, cff);
    Read(sharedata, 8, status);

    cout << " After wait .. FF " << cff << " " << status << endl;
    if(status <= 0) break;

    // Solve
    Pb;
    rff = int2d(Th)(u*u);
    cout << " ** FF " << cff << " " << rff << endl;

    // Write
    Write(sharedata, 16, rff);
    Post(smff); //unlock cc
}

Post(smff); //wait from cint
cout << " End FreeFEM " << endl;
```

To test this example of coupling C program and **FreeFEM** script:

```
cc -c libff-mmap-semaphore.c
cc FFMaster.c -o FFMaster libff-mmap-semaphore.o -g -pthread
ff-c++ -auto ff-mmap-semaphore.cpp
./FFMaster
```

The output:

```
len 1024 size 0
len 1024 size 1024
FF status = 1
iter : 0
After wait .. FF 10 1
** FF 10 0.161797
iter = 0 rff= 0.161797
iter : 1
After wait .. FF 11 1
** FF 11 0.195774
iter = 1 rff= 0.195774
iter : 2
After wait .. FF 12 1
** FF 12 0.232987
iter = 2 rff= 0.232987
iter : 3
After wait .. FF 13 1
** FF 13 0.273436
iter = 3 rff= 0.273436
iter : 4
After wait .. FF 14 1
** FF 14 0.317121
iter = 4 rff= 0.317121
iter : 5
After wait .. FF 15 1
** FF 15 0.364042
iter = 5 rff= 0.364042
```

(continues on next page)

(continued from previous page)

```
iter : 6
After wait .. FF 16 1
** FF 16 0.414199
iter = 6 rff= 0.414199
iter : 7
After wait .. FF 17 1
** FF 17 0.467592
iter = 7 rff= 0.467592
iter : 8
After wait .. FF 18 1
** FF 18 0.524221
iter = 8 rff= 0.524221
iter : 9
After wait .. FF 19 1
** FF 19 0.584086
iter = 9 rff= 0.584086
End Master
After wait .. FF 19 0
```

3.8 Developers

3.8.1 File formats

Mesh file data structure

The mesh data structure, output of a mesh generation algorithm, refers to the geometric data structure and in some case to another mesh data structure.

In this case, the fields are

```
MeshVersionFormatted 0

Dimension [DIM](int)

Vertices
[Number of vertices](int)
X_1(double) Y_1(double) (Z_1(double)) Ref_1(int)
...
X_nv(double) Y_nv(double) (Z_nv(double)) Ref_nv(int)

Edges
[Number of edges](int)
Vertex1_1(int) Vertex2_1(int) Ref_1(int)
...
Vertex1_ne(int) Vertex2_ne(int) Ref_ne(int)

Triangles
[Number of triangles](int)
Vertex1_1(int) Vertex2_1(int) Vertex3_1(int) Ref_1(int)
...
Vertex1_nt(int) Vertex2_nt(int) Vertex3_nt(int) Ref_nt(int)

Quadrilaterals
```

(continues on next page)

(continued from previous page)

```
[Number of Quadrilaterals](int)
Vertex1_1(int) Vertex2_1(int) Vertex3_1(int) Vertex4_1(int) Ref_1(int)
...
Vertex1_nq(int) Vertex2_nq(int) Vertex3_nq(int) Vertex4_nq(int) Ref_nq(int)

Geometry
[File name of geometric support](char*)

   VertexOnGeometricVertex
   [Number of vertex on geometric vertex](int)
   Vertex_1(int) VertexGeometry_1(int)
   ...
   Vertex_nvg(int) VertexGeometry_nvg(int)

   EdgeOnGeometricEdge
   [Number of geometric edge](int)
   Edge_1(int) EdgeGeometry_1(int)
   ...
   Edge_neg(int) EdgeGeometry_neg(int)

CrackedEdges
[Number of cracked edges](int)
Edge1_1(int) Edge2_1(int)
...
Edge1_nce(int) Edge2_nce(int)
```

When the current mesh refers to a previous mesh, we have in addition

```
MeshSupportOfVertices
[File name of mesh support](char*)

   VertexOnSupportVertex
   [Number of vertex on support vertex](int)
   Vertex_1(int) VertexSupport_1(int)
   ...
   Vertex_nvsv(int) VertexSupport_nvsv(int)

   VertexOnSupportEdge
   [Number of vertex on support edge](int)
   Vertex_1(int) EdgeSupport_1(int) USupport_1(double)
   ...
   Vertex_nvse(int) EdgeSupport_nvse(int) USupport_nvse(double)

   VertexOnSupportTriangle
   [Number of vertex on support triangle](int)
   Vertex_1(int) TriangleSupport_1(int) USupport_1(double) VSupport_1(double)
   ...
   Vertex_nvst(int) TriangleSupport_nvst(int) USupport_nvst(double) VSupport_
↪nvst(double)

   VertexOnSupportQuadrilaterals
   [Number of vertex on support quadrilaterals]
   Vertex_1(int) TriangleSupport_1(int) USupport_1(double) VSupport_1(double)
   ...
   Vertex_nvsq(int) TriangleSupport_nvsq(int) USupport_nvsq(double) VSupport_
↪nvsq(double)
```

- `nv` means the number of vertices
- `ne` means the number of edges
- `nt` means the number of triangles
- `nq` means the number of quadrilaterals
- `nvg` means the number of vertex on geometric vertex
- `neg` means the number of edges on geometric edge
- `nce` means the number of cracked edges

bb file type to Store Solutions

The file is formatted such that:

```
2 [Number of solutions](int) [Number of vertices](int) 2

U_1_1(double) ... U_ns_1(double)
...
U_1_nv(double) ... U_ns_nv(double)
```

- `ns` means the number of solutions
- `nv` means the number of vertices
- `U_i_j` is the solution component `i` at the vertex `j` on the associated mesh.

BB file type to store solutions

The file is formatted such that:

```
2 [Number of solutions](int) [Type 1](int) ... [Type ns](int) [Number of vertices](int) 2

U_1_1_1(double) ... U_(type_k)_1_1(double)
...
U_1_1_1(double) ... U_(type_k)_nbv_1(double)

...

U_1_1_ns(double) ... U_(type_k)_1_ns(double)
...
U_1_nbv_ns(double) ... U_(type_k)_nbv_ns(double)
```

- `ns` means the number of solutions
- `type_k` mean the type of solution `k`:
  - 1: the solution is scalar (1 value per vertex)
  - 2: the solution is vectorial (2 values per vertex)
  - 3: the solution is a 2×2 symmetric matrix (3 values per vertex)
  - 4: the solution is a 2×2 matrix (4 values per vertex)
- `nbv` means the number of vertices
- `U_i_j_k` is the value of the component `i` of the solution `k` at vertex `j` on the associated mesh

Metric file

A metric file can be of two types, isotropic or anisotropic.

The isotropic file is such that

```
[Number of vertices](int) 1
h_0(double)
...
h_nv(double)
```

- `nv` is the number of vertices
- `h_i` is the wanted mesh size near the vertex `i` on associated mesh.

The metric is $\mathcal{M}_i = h_i^{-2} I$ where I is the identity matrix.

The anisotropic file is such that

```
[Number of vertices](int) 3
a11_0(double) a21_0(double) a22_0(double)
...
a11_nv(double) a21_nv(double) a22_nv(double)
```

- `nv` is the number of vertices
- `a11_i`, `a21_i` and `a22_i` represent metric $\mathcal{M}_i = \begin{pmatrix} a_{11,i} & a_{12,i} \\ a12_i & a_{22,i} \end{pmatrix}$ which define the wanted size in a vicinity of the vertex `i` such that h in direction $u \in \mathbb{R}^2$ is equal to $|u|/\sqrt{u \cdot \mathcal{M}_i\, u}$, where $\cdot$ is the dot product in $\mathbb{R}^2$, and $|\cdot|$ is the classical norm.

List of AM_FMT, AMDBA Meshes

The mesh is only composed of triangles and can be defined with the help of the following two integers and four arrays:

- `nbt` the number of triangles
- `nbv` the number of vertices
- `nu(1:3, 1:nbt)` an integer array giving the three vertex numbers counterclockwise for each triangle
- `c(1:2, 1:nbv)` a real array giving tje two coordinates of each vertex
- `refs(1:nbv)` an integer array giving the reference numbers of the vertices
- `reft(1:nbt)` an integer array giving the reference numbers of the triangles

AM_FMT Files

In `Fortran` the `am_fmt` files are read as follows:

```
open (1, file='xxx.am_fmt', form='formatted', status='old')
read (1, *) nbv, nbt
read (1, *) ((nu(i, j), i=1, 3), j=1, nbt)
read (1, *) ((c(i, j), i=1, 2), j=1, nbv)
read (1, *) ( reft(i), i=1, nbt)
read (1, *) ( refs(i), i=1, nbv)
close(1)
```

AM Files

In `Fortran` the `am` files are read as follows:

```
open (1, file='xxx.am', form='unformatted', status='old')
read (1, *) nbv, nbt
read (1) ((nu(i, j), i=1, 3), j=1, nbt),
& ((c(i, j), i=1, 2), j=1, nbv),
& (reft(i), i=1, nbt),
& (refs(i), i=1, nbv)
close(1)
```

AMDBA Files

In `Fortran` the `amdba` files are read as follows:

```
open (1, file='xxx.amdba', form='formatted', status='old')
read (1, *) nbv, nbt
read (1, *) (k, (c(i, k), i=1, 2), refs(k), j=1, nbv)
read (1, *) (k, (nu(i, k), i=1, 3), reft(k), j=1, nbt)
close(1)
```

msh Files

First, we add the notions of boundary edges

- `nbbe` the number of boundary edge
- `nube(1:2, 1:nbbe)` an integer array giving the two vertex numbers of boundary edges
- `refbe(1:nbbe)` an integer array giving the reference numbers of boundary edges

In `Fortran` the `msh` files are read as follows:

```
open (1, file='xxx.msh', form='formatted', status='old')
read (1, *) nbv, nbt, nbbe
read (1, *) ((c(i, k), i=1, 2), refs(k), j=1, nbv)
read (1, *) ((nu(i, k), i=1, 3), reft(k), j=1, nbt)
read (1, *) ((ne(i, k), i=1, 2), refbe(k), j=1, nbbe)
close(1)
```

ftq Files

In `Fortran` the `ftq` files are read as follows:

```
open(1,file='xxx.ftq',form='formatted',status='old')
read (1,*) nbv,nbe,nbt,nbq
read (1,*) (k(j),(nu(i,j),i=1,k(j)),reft(j),j=1,nbe)
read (1,*) ((c(i,k),i=1,2),refs(k),j=1,nbv)
close(1)
```

where if `k(j) = 3` when the element `j` is a triangle and `k(j) = 4` when the the element `j` is a quadrilateral.

sol and solb files

With the keyword `savesol`, we can store a scalar functions, a scalar finite element functions, a vector fields, a vector finite element fields, a symmetric tensor and a symmetric finite element tensor.

Such format is used in `medit`.

Extension file .sol

The first two lines of the file are :

- `MeshVersionFormatted 0`

- `Dimension [DIM](int)`

The following fields begin with one of the following keyword: `SolAtVertices`, `SolAtEdges`, `SolAtTriangles`, `SolAtQuadrilaterals`, `SolAtTetrahedra`, `SolAtPentahedra`, `SolAtHexahedra`.

In each field, we give then in the next line the number of elements in the solutions (`SolAtVertices`: number of vertices, `SolAtTriangles`: number of triangles, ...). In other lines, we give the number of solutions, the type of solution (1: scalar, 2: vector, 3: symmetric tensor). And finally, we give the values of the solutions on the elements.

The file must be ended with the keyword End.

The real element of symmetric tensor :

$$ST^{3d} = \begin{pmatrix} ST^{3d}_{xx} & ST^{3d}_{xy} & ST^{3d}_{xz} \\ ST^{3d}_{yx} & ST^{3d}_{yy} & ST^{3d}_{yz} \\ ST^{3d}_{zx} & ST^{3d}_{zy} & ST^{3d}_{zz} \end{pmatrix} \quad ST^{2d} = \begin{pmatrix} ST^{2d}_{xx} & ST^{2d}_{xy} \\ ST^{2d}_{yx} & ST^{2d}_{yy} \end{pmatrix} \tag{3.32}$$

stored in the extension `.sol` are respectively $ST^{3d}_{xx}, ST^{3d}_{yx}, ST^{3d}_{yy}, ST^{3d}_{zx}, ST^{3d}_{zy}, ST^{3d}_{zz}$ and $ST^{2d}_{xx}, ST^{2d}_{yx}, ST^{2d}_{yy}$

An example of field with the keyword `SolAtTetrahedra`:

```
SolAtTetrahedra
[Number of tetrahedra](int)
[Number of solutions](int) [Type of solution 1](int) ... [Type of soution nt](int)

U_1_1_1(double) ... U_nrs_1_1(double)
...
U_1_ns_1(double) ... U_(nrs_k)_ns_1(double)

...

U_1_1_nt(double) ... U_nrs_1_nt(double)
...
U_1_ns_nt(double) ... U_(nrs_k)_ns_nt(double)
```

- `ns` is the number of solutions
- `typesol_k`, type of the solution number `k`
  - `typesol_k = 1` the solution `k` is scalar
  - `typesol_k = 2` the solution `k` is vectorial
  - `typesol_k = 3` the solution `k` is a symmetric tensor or symmetric matrix
- `nrs_k` is the number of real to describe solution `k`
  - `nrs_k = 1` if the solution `k` is scalar
  - `nrs_k = dim` if the solution `k` is vectorial (`dim` is the dimension of the solution)
  - `nrs_k = dim*(dim+1)/2` if the solution k is a symmetric tensor or symmetric matrix
- `U_i_j_^k` is a real equal to the value of the component `i` of the solution `k` at tetrahedron `j` on the associated mesh

The format `.solb` is the same as format `.sol` but in binary (read/write is faster, storage is less).

A real scalar functions $f1$, a vector fields $\mathbf{\Phi} = [\Phi1, \Phi2, \Phi3]$ and a symmetric tensor ST^{3d} (3.32) at the vertices of the three dimensional mesh `Th3` is stored in the file `f1PhiTh3.sol` using :

```
savesol("f1PhiST3dTh3.sol", Th3, f1, [Phi(1), Phi(2), Phi(3)], VV3, order=1);
```

where $VV3 = [ST^{3d}_{xx}, ST^{3d}_{yx}, ST^{3d}_{yy}, ST^{3d}_{zx}, ST^{3d}_{zy}, ST^{3d}_{zz}]$.

For a two dimensional mesh `Th`, A real scalar functions $f2$, a vector fields $\mathbf{\Psi} = [\Psi 1, \Psi 2]$ and a symmetric tensor ST^{2d} (3.32) at triangles is stored in the file `f2PsiST2dTh3.solb` using :

```
savesol("f2PsiST2dTh3.solb", Th, f2, [Psi(1), Psi(2)], VV2, order=0);
```

where $VV2 = [ST^{2d}_{xx}, ST^{2d}_{yx}, ST^{2d}_{yy}]$

The arguments of `savesol` functions are the name of a file, a mesh and solutions. These arguments must be given in this order.

The parameters of this keyword are :

- `order` = 0 is the solution is given at the center of gravity of elements. 1 is the solution is given at the vertices of elements.

In the file, solutions are stored in this order : scalar solutions, vector solutions and finally symmetric tensor solutions.

3.8.2 Adding a new finite element

Some notations

For a function $\boldsymbol{f}$ taking value in $\mathbb{R}^N$, $N = 1, 2, \cdots$, we define the finite element approximation $\Pi_h \boldsymbol{f}$ of $\boldsymbol{f}$.

Let us denote the number of the degrees of freedom of the finite element by $NbDoF$. Then the i-th base $\boldsymbol{\omega}_i^K$ ($i = 0, \cdots, NbDoF - 1$) of the finite element space has the j-th component ω_{ij}^K for $j = 0, \cdots, N - 1$.

The operator Π_h is called the interpolator of the finite element.

We have the identity $\boldsymbol{\omega}_i^K = \Pi_h \boldsymbol{\omega}_i^K$.

Formally, the interpolator Π_h is constructed by the following formula:

$$\Pi_h \boldsymbol{f} = \sum_{k=0}^{\texttt{kPi}-1} \alpha_k \boldsymbol{f}_{j_k}(P_{p_k}) \boldsymbol{\omega}_{i_k}^K \tag{3.33}$$

where P_p is a set of $npPi$ points,

In the formula (3.33), the list p_k, j_k, i_k depend just on the type of finite element (not on the element), but the coefficient α_k can be depending on the element.

Tip: Classical scalar Lagrange finite element

With the classical scalar Lagrange finite element, we have `kPi` = `npPi` = `NbOfNode` and

- P_p is the point of the nodal points.
- the $\alpha_k = 1$, because we take the value of the function at the point P_k.
- $p_k = k$, $j_k = k$ because we have one node per function.
- $j_k = 0$ because $N = 1$.

Tip: The Raviart-Thomas finite element

$$RT0_h = \{\mathbf{v} \in H(div)/\forall K \in \mathcal{T}_h \quad \mathbf{v}_{|K}(x, y) = \left|\begin{smallmatrix} \alpha_K \\ \beta_K \end{smallmatrix}\right. + \gamma_K \left|\begin{smallmatrix} x \\ y \end{smallmatrix}\right. \} \tag{3.34}$$

The degrees of freedom are the flux through an edge e of the mesh, where the flux of the function $\mathbf{f} : \mathbb{R}^2 \longrightarrow \mathbb{R}^2$ is $\int_e \mathbf{f}.n_e$, n_e is the unit normal of edge e (this implies a orientation of all the edges of the mesh, for example we can use the global numbering of the edge vertices and we just go to small to large number).

To compute this flux, we use a quadrature formula with one point, the middle point of the edge. Consider a triangle T with three vertices $(\mathbf{a}, \mathbf{b}, \mathbf{c})$.

Let denote the vertices numbers by i_a, i_b, i_c, and define the three edge vectors $\mathbf{e}^0, \mathbf{e}^1, \mathbf{e}^2$ by $sgn(i_b - i_c)(\mathbf{b} - \mathbf{c})$, $sgn(i_c - i_a)(\mathbf{c} - \mathbf{a})$, $sgn(i_a - i_b)(\mathbf{a} - \mathbf{b})$.

The three basis functions are:

$$\boldsymbol{\omega}_0^K = \frac{sgn(i_b - i_c)}{2|T|}(x - a), \quad \boldsymbol{\omega}_1^K = \frac{sgn(i_c - i_a)}{2|T|}(x - b), \quad \boldsymbol{\omega}_2^K = \frac{sgn(i_a - i_b)}{2|T|}(x - c),$$

where $|T|$ is the area of the triangle T.

So we have $N = 2$, `kPi` $= 6$; `npPi` $= 3$; and:

- $P_p = \left\{ \frac{\mathbf{b}+\mathbf{c}}{2}, \frac{\mathbf{a}+\mathbf{c}}{2}, \frac{\mathbf{b}+\mathbf{a}}{2} \right\}$
- $\alpha_0 = -\mathbf{e}_2^0, \alpha_1 = \mathbf{e}_1^0,\ \alpha_2 = -\mathbf{e}_2^1, \alpha_3 = \mathbf{e}_1^1, \alpha_4 = -\mathbf{e}_2^2, \alpha_5 = \mathbf{e}_1^2$ (effectively, the vector $(-\mathbf{e}_2^m, \mathbf{e}_1^m)$ is orthogonal to the edge $\mathbf{e}^m = (e_1^m, e_2^m)$ with a length equal to the side of the edge or equal to $\int_{e^m} 1$).
- $i_k = \{0, 0, 1, 1, 2, 2\}$,
- $p_k = \{0, 0, 1, 1, 2, 2\}$, $j_k = \{0, 1, 0, 1, 0, 1, 0, 1\}$.

Which class to add?

Add file `FE_ADD.cpp` in directory `FreeFem-sources/src/femlib` for example first to initialize :

```
#include "error.hpp"
#include "rgraph.hpp"
using namespace std;
#include "RNM.hpp"
#include "fem.hpp"
#include "FESpace.hpp"
#include "AddNewFE.h"

namespace Fem2D { ... }
```

Then add a class which derive for `public TypeOfFE` like:

```
class TypeOfFE_RTortho : public TypeOfFE { public:
    static int Data[]; //some numbers
    TypeOfFE_RTortho():
    TypeOfFE(
        0+3+0,  //nb degree of freedom on element
        2,      //dimension N of vectorial FE (1 if scalar FE)
        Data,   //the array data
        1,      //nb of subdivision for plotting
        1,      //nb of sub finite element (generaly 1)
        6,      //number kPi of coef to build the interpolator
        3,      //number npPi of integration point to build interpolator
        0       //an array to store the coef \alpha_k to build interpolator
        //here this array is no constant so we have
        //to rebuilt for each element
    )
```

(continues on next page)

(continued from previous page)

```
    {
        const R2 Pt[] = {R2(0.5, 0.5), R2(0.0, 0.5), R2(0.5, 0.0) };
        // the set of Point in hat{K}
        for (int p = 0, kk = 0; p < 3; p++){
            P_Pi_h[p] = Pt[p];
            for (int j = 0; j < 2; j++)
                pij_alpha[kk++] = IPJ(p, p, j);
        }
    } //definition of i_k, p_k, j_k in interpolator

    void FB(const bool *watdd, const Mesh &Th, const Triangle &K,
        const R2 &PHat, RNMK_ &val) const;

    void Pi_h_alpha(const baseFElement &K, KN_<double> &v) const;
} ;
```

where the array data is formed with the concatenation of five array of size `NbDoF` and one array of size `N`.

This array is:

```
int TypeOfFE_RTortho::Data[] = {
    //for each df 0, 1, 3:
    3, 4, 5, //the support of the node of the df
    0, 0, 0, //the number of the df on the node
    0, 1, 2, //the node of the df
    0, 0, 0, //the df come from which FE (generally 0)
    0, 1, 2, //which are the df on sub FE
    0, 0
}; //for each component j=0, N-1 it give the sub FE associated
```

where the support is a number $0, 1, 2$ for vertex support, $3, 4, 5$ for edge support, and finally 6 for element support.

The function to defined the function $\boldsymbol{\omega}_i^K$, this function return the value of all the basics function or this derivatives in array `val`, computed at point `Phat` on the reference triangle corresponding to point `R2 P=K(Phat);` on the current triangle `K`.

The index i, j, k of the array $val(i, j, k)$ correspond to:

- i is the basic function number on finite element $i \in [0, NoF[$
- j is the value of component $j \in [0, N[$
- k is the type of computed value $f(P), dx(f)(P), dy(f)(P), \ldots$ $i \in [0,$ `last_operatortype`$[$.

Note: For optimization, this value is computed only if `whatd[k]` is true, and the numbering is defined with

```
enum operatortype {
op_id = 0,
op_dx = 1, op_dy = 2,
op_dxx = 3,op_dyy = 4,
op_dyx = 5,op_dxy = 5,
op_dz = 6,
op_dzz = 7,
op_dzx = 8, op_dxz = 8,
op_dzy = 9, op_dyz = 9
};
const int last_operatortype = 10;
```

The shape function:

```
void TypeOfFE_RTortho::FB(const bool *whatd, const Mesh &Th, const Triangle & K,
    const R2 &PHat,RNMK_ &val) const
{
    R2 P(K(PHat));
    R2 A(K[0]), B(K[1]), C(K[2]);
    R l0 = 1 - P.x-P.y;
    R l1 = P.x, l2 = P.y;
    assert(val.N() >= 3);
    assert(val.M() == 2);
    val = 0;
    R a = 1./(2*K.area);
    R a0 = K.EdgeOrientation(0) * a;
    R a1 = K.EdgeOrientation(1) * a;
    R a2 = K.EdgeOrientation(2) * a;

    if (whatd[op_id]){ //value of the function
        assert(val.K() > op_id);
        RN_ f0(val('.', 0,0)); //value first component
        RN_ f1(val('.', 1,0)); //value second component
        f1[0] = (P.x - A.x)*a0;
        f0[0] = -(P.y - A.y)*a0;

        f1[1] = (P.x - B.x)*a1;
        f0[1] = -(P.y - B.y)*a1;

        f1[2] = (P.x - C.x)*a2;
        f0[2] = -(P.y - C.y)*a2;
    }

    if (whatd[op_dx]){ //value of the dx of function
        assert(val.K() > op_dx);
        val(0,1,op_dx) = a0;
        val(1,1,op_dx) = a1;
        val(2,1,op_dx) = a2;
    }
    if (whatd[op_dy]){
        assert(val.K() > op_dy);
        val(0,0,op_dy) = -a0;
        val(1,0,op_dy) = -a1;
        val(2,0,op_dy) = -a2;
    }

    for (int i = op_dy; i < last_operatortype; i++)
        if (whatd[op_dx])
            assert(op_dy);
}
```

The function to defined the coefficient α_k:

```
void TypeOfFE_RT::Pi_h_alpha(const baseFElement &K, KN_<double> &v) const
{
    const Triangle &T(K.T);

    for (int i = 0, k = 0; i < 3; i++){
        R2 E(T.Edge(i));
        R signe = T.EdgeOrientation(i) ;
```

(continues on next page)

(continued from previous page)

```
        v[k++] = signe*E.y;
        v[k++] = -signe*E.x;
    }
}
```

Now , we just need to add a new key work in **FreeFEM**.

Two way, with static or dynamic link so at the end of the file, we add:

With dynamic link it is very simple (see section *Dynamical link*), just add before the end of `FEM2d namespace`:

```
   static TypeOfFE_RTortho The_TypeOfFE_RTortho;
   static AddNewFE("RT0Ortho", The_TypeOfFE_RTortho);
} //FEM2d namespace
```

Try with `./load.link` command in examples++-load/ and see `BernardiRaugel.cpp` or `Morley.cpp` new finite element examples.

Otherwise with static link (for expert only), add

```
//let the 2 globals variables
static TypeOfFE_RTortho The_TypeOfFE_RTortho;
//the name in freefem
static ListOfTFE typefemRTOrtho("RT0Ortho", &The_TypeOfFE_RTortho);

//link with FreeFEM do not work with static library .a
//so add a extern name to call in init_static_FE
//(see end of FESpace.cpp)
void init_FE_ADD() { };
//end
} //FEM2d namespace
```

To inforce in loading of this new finite element, we have to add the two new lines close to the end of files `src/femlib/FESpace.cpp` like:

```
//correct problem of static library link with new make file
void init_static_FE()
{ //list of other FE file.o
    extern void init_FE_P2h() ;
    init_FE_P2h() ;
    extern void init_FE_ADD(); //new line 1
    init_FE_ADD(); //new line 2
}
```

and now you have to change the makefile.

First, create a file `FE_ADD.cpp` contening all this code, like in file `src/femlib/Element_P2h.cpp`, after modify the `Makefile.am` by adding the name of your file to the variable `EXTRA_DIST` like:

```
# Makefile using Automake + Autoconf
# ----------------------------------
# Id

# This is not compiled as a separate library because its
# interconnections with other libraries have not been solved.

EXTRA_DIST=BamgFreeFem.cpp BamgFreeFem.hpp CGNL.hpp CheckPtr.cpp        \
ConjuguedGradrientNL.cpp DOperator.hpp Drawing.cpp Element_P2h.cpp      \
```

(continues on next page)

(continued from previous page)

```
Element_P3.cpp Element_RT.cpp fem3.hpp fem.cpp fem.hpp FESpace.cpp      \
FESpace.hpp FESpace-v0.cpp FQuadTree.cpp FQuadTree.hpp gibbs.cpp        \
glutdraw.cpp gmres.hpp MatriceCreuse.hpp MatriceCreuse_tpl.hpp          \
MeshPoint.hpp mortar.cpp mshptg.cpp QuadratureFormular.cpp              \
QuadratureFormular.hpp RefCounter.hpp RNM.hpp RNM_opc.hpp RNM_op.hpp    \
RNM_tpl.hpp     FE_ADD.cpp
```

and do in the **FreeFEM** root directory

```
autoreconf
./reconfigure
make
```

For codewarrior compilation add the file in the project an remove the flag in panal PPC linker FreeFm++ Setting Dead-strip Static Initializition Code Flag.

3.8.3 Dynamical link

Now, it's possible to add built-in functionnalites in **FreeFEM** under the three environnents Linux, Windows and MacOS X 10.3 or newer.

It is agood idea to first try the example `load.edp` in directory example++-load.

You will need to install a `compiler` (generally `g++/gcc` compiler) to compile your function.

- Windows Install the `cygwin` environnent or the `mingw` one
- MacOs Install the developer tools `Xcode` on the apple DVD
- Linux/Unix Install the correct compiler (`gcc` for instance)

Now, assume that you are in a shell window (a `cygwin` window under Windows) in the directory example++-load.

Note: In the sub directory `include`, they are all the **FreeFEM** include file to make the link with **FreeFEM**.

Note: If you try to load dynamically a file with command `load "xxx"` - Under Unix (Linux or MacOs), the file `xxx.so` will be loaded so it must be either in the search directory of routine `dlopen` (see the environment variable *$LD_LIBRARY_PATH*) or in the current directory, and the suffix `".so"` or the prefix `"./"` is automatically added.

- Under Windows, the file *xxx.dll* will be loaded so it must be in the *loadLibary* search directory which includes the directory of the application,

Compilation of your module:

The script `ff-c++` compiles and makes the link with **FreeFEM**, but be careful, the script has no way to known if you try to compile for a pure Windows environment or for a cygwin environment so to build the load module under cygwin you must add the `-cygwin` parameter.

A first example myfunction.cpp

The following defines a new function call `myfunction` with no parameter, but using the x, y current value.

```
#include <iostream>
#include <cfloat>
using namespace std;
#include "error.hpp"
#include "AFunction.hpp"
#include "rgraph.hpp"
#include "RNM.hpp"
#include "fem.hpp"
#include "FESpace.hpp"
#include "MeshPoint.hpp"

using namespace Fem2D;
double myfunction(Stack stack){
    //to get FreeFEM data
    MeshPoint &mp = *MeshPointStack(stack); //the struct to get x, y, normal, value
    double x = mp.P.x; //get the current x value
    double y = mp.P.y; //get the current y value
    //cout << "x = " << x << " y=" << y << endl;
    return sin(x)*cos(y);
}
```

Now the Problem is to build the link with **FreeFEM**, to do that we need two classes, one to call the function `myfunction`.

All **FreeFEM** evaluable expression must be a `C++ struct/class` which derivate from `E_F0`. By default this expression does not depend of the mesh position, but if they derivate from `E_F0mps` the expression depends of the mesh position, and for more details see [HECHT2002].

```
//A class build the link with FreeFEM
//generaly this class are already in AFunction.hpp
//but unfortunatly, I have no simple function with no parameter
//in FreeFEM depending of the mesh
template<class R>
class OneOperator0s : public OneOperator {
    //the class to define and evaluate a new function
    //It must devive from E_F0 if it is mesh independent
    //or from E_F0mps if it is mesh dependent
    class E_F0_F :public E_F0mps {
    public:
        typedef R (*func)(Stack stack);
        func f; //the pointeur to the fnction myfunction
        E_F0_F(func ff) : f(ff) {}
        //the operator evaluation in FreeFEM
        AnyType operator()(Stack stack) const {return SetAny<R>(f(stack));}
    };
    typedef R (*func)(Stack);
    func f;
    public:
        //the function which build the FreeFEM byte code
        E_F0 *code(const basicAC_F0 &) const { return new E_F0_F(f); }
        //the constructor to say ff is a function without parameter
        //and returning a R
        OneOperator0s(func ff) : OneOperator(map_type[typeid(R).name()]),f(ff){}
};
```

To finish we must add this new function in **FreeFEM** table, to do that include :

```
 void init(){
     Global.Add("myfunction", "(", new OneOperator0s<double>(myfunction));
 }
 LOADFUNC(init);
```

It will be called automatically at load module time.

To compile and link, use the `ff-c++` script :

```
ff-c++ myfunction.cpp
g++ -c -g -Iinclude myfunction.cpp
g++ -bundle -undefined dynamic_lookup -g myfunction.o -o ./myfunction.dylib
```

To try the simple example under Linux or MacOS, do `FreeFem++-nw load.edp`

The output must be:

```
-- FreeFem++ v  *.****** (date *** ** *** ****, **:**:** (UTC+0*00))
 Load: lg_fem lg_mesh lg_mesh3 eigenvalue
    1 : // Example of dynamic function load
    2 : // -------------------------------
    3 : // $Id$
    4 :
    5 :  load "myfunction"
    6 : // dumptable(cout);
    7 :  mesh Th=square(5,5);
    8 :  fespace Vh(Th,P1);
    9 :  Vh uh= myfunction(); // warning do not forget ()
   10 :  cout << uh[].min << " " << uh[].max << endl;
   11 :  cout << " test io ( " << endl;
   12 :  testio();
   13 :  cout << " ) end test io .. " << endl; sizestack + 1024 =1416  ( 392 )

  -- Square mesh : nb vertices  =36 ,  nb triangles = 50 ,  nb boundary edges 20
0 0.841471
 test io (
 test cout 3.14159
 test cout 512
 test cerr 3.14159
 test cerr 512
 ) end test io ..
times: compile 0.012854s, execution 0.000313s,  mpirank:0
 CodeAlloc : nb ptr  2715,  size :371104 mpirank: 0
Ok: Normal End
```

Under Windows, launch **FreeFEM** with the mouse (or ctrl O) on the example.

Example: Discrete Fast Fourier Transform

This will add FFT to **FreeFEM**, taken from FFTW. To download and install under `download/include` just go in `download/fftw` and try `make`.

The 1D dfft (fast discret fourier transform) for a simple array f of size n is defined by the following formula:

$$\texttt{dfft}(f,\varepsilon)_k = \sum_{j=0}^{n-1} f_i e^{\varepsilon 2\pi i k j/n}$$

The 2D DFFT for an array of size $N = n \times m$ is:

$$\texttt{dfft}(f, m, \varepsilon)_{k+nl} = \sum_{j'=0}^{m-1} \sum_{j=0}^{n-1} f_{i+nj} e^{\varepsilon 2\pi i(kj/n+lj'/m)}$$

Note: The value n is given by $size(f)/m$, and the numbering is row-major order.

So the classical discrete DFFT is $\hat{f} = \texttt{dfft}(f, -1)/\sqrt{n}$ and the reverse dFFT $f = \texttt{dfft}(\hat{f}, 1)/\sqrt{n}$

Note: The 2D Laplace operator is

$$f(x, y) = 1/\sqrt{N} \sum_{j'=0}^{m-1} \sum_{j=0}^{n-1} \hat{f}_{i+nj} e^{\varepsilon 2\pi i(xj+yj')}$$

and we have

$$f_{k+nl} = f(k/n, l/m)$$

So

$$\widehat{\Delta f_{kl}} = -((2\pi)^2((\tilde{k})^2 + (\tilde{l})^2))\widehat{f_{kl}}$$

where $\tilde{k} = k$ if $k \leq n/2$ else $\tilde{k} = k - n$ and $\tilde{l} = l$ if $l \leq m/2$ else $\tilde{l} = l - m$.

And to have a real function we need all modes to be symmetric around zero, so n and m must be odd.

Compile to build a new library

```
ff-c++ dfft.cpp ../download/install/lib/libfftw3.a -I../download/install/include
export MACOSX_DEPLOYMENT_TARGET=10.3
g++ -c -Iinclude -I../download/install/include dfft.cpp
g++ -bundle -undefined dynamic_lookup dfft.o -o ./dfft.dylib ../download/install/lib/
↪libfftw3.a
```

To test, try *FFT example*.

Load Module for Dervieux P0-P1 Finite Volume Method

The associed edp file is examples++-load/convect_dervieux.edp.

See mat_dervieux.cpp.

More on Adding a new finite element

First read the *Adding a new finite element section*, we add two new finite elements examples in the directory examples++-load.

The Bernardi-Raugel Element

The Bernardi-Raugel finite element is meant to solve the Navier Stokes equations in u, p formulation; the velocity space P_K^{br} is minimal to prove the inf-sup condition with piecewise constant pressure by triangle.

The finite element space V_h is

$$V_h = \{u \in H^1(\Omega)^2; \quad \forall K \in T_h, u_{|K} \in P_K^{br}\}$$

where

$$P_K^{br} = span\{\lambda_i^K e_k\}_{i=1,2,3,k=1,2} \cup \{\lambda_i^K \lambda_{i+1}^K n_{i+2}^K\}_{i=1,2,3}$$

with notation $4 = 1, 5 = 2$ and where λ_i^K are the barycentric coordinates of the triangle K, $(e_k)_{k=1,2}$ the canonical basis of $\mathbb{R}^2$ and n_k^K the outer normal of triangle K opposite to vertex k.

See BernardiRaugel.cpp.

A way to check the finite element

```
load "BernardiRaugel"

// Macro
//a macro the compute numerical derivative
macro DD(f, hx, hy) ( (f(x1+hx, y1+hy) - f(x1-hx, y1-hy))/(2*(hx+hy)) ) //

// Mesh
mesh Th = square(1, 1, [10*(x+y/3), 10*(y-x/3)]);

// Parameters
real x1 = 0.7, y1 = 0.9, h = 1e-7;
int it1 = Th(x1, y1).nuTriangle;

// Fespace
fespace Vh(Th, P2BR);
Vh [a1, a2], [b1, b2], [c1, c2];


for (int i = 0; i < Vh.ndofK; ++i)
    cout << i << " " << Vh(0,i) << endl;

for (int i = 0; i < Vh.ndofK; ++i)
{
    a1[] = 0;
    int j = Vh(it1, i);
    a1[][j] = 1;
    plot([a1, a2], wait=1);
    [b1, b2] = [a1, a2]; //do the interpolation

    c1[] = a1[] - b1[];
    cout << " ---------" << i << " " << c1[].max << " " << c1[].min << endl;
    cout << " a = " << a1[] <<endl;
    cout << " b = " << b1[] <<endl;
    assert(c1[].max < 1e-9 && c1[].min > -1e-9); //check if the interpolation is correct

    // check the derivative and numerical derivative
    cout << " dx(a1)(x1, y1) = " << dx(a1)(x1, y1) << " == " << DD(a1, h, 0) << endl;
```

(continues on next page)

(continued from previous page)

```
    assert( abs(dx(a1)(x1, y1) - DD(a1, h, 0) ) < 1e-5);
    assert( abs(dx(a2)(x1, y1) - DD(a2, h, 0) ) < 1e-5);
    assert( abs(dy(a1)(x1, y1) - DD(a1, 0, h) ) < 1e-5);
    assert( abs(dy(a2)(x1, y1) - DD(a2, 0, h) ) < 1e-5);
}
```

A real example using this finite element, just a small modification of the Navier-Stokes P2-P1 example, just the begenning is change to

```
load "BernardiRaugel"

real s0 = clock();
mesh Th = square(10, 10);
fespace Vh2(Th, P2BR);
fespace Vh(Th, P0);
Vh2 [u1, u2], [up1, up2];
Vh2 [v1, v2];
```

And the plot instruction is also changed because the pressure is constant, and we cannot plot isovalues of peacewise constant functions.

The Morley Element

See the example bilapMorley.edp.

3.9 ffddm

In the acronym `ffddm`, `ff` stands for FreeFEM and `ddm` for domain decomposition methods. The idea behind ffddm is to simplify the use of parallel solvers in FreeFEM: distributed direct methods and domain decomposition methods.

Parallelism is an important issue because, since about 2004, the clock speed of cores stagnates at 2-3 GHz. The increase in performance is almost entirely due to the increase in the number of cores per processor. All major processor vendors are producing multicore chips and now every machine is a parallel machine. Waiting for the next generation machine does not guarantee anymore a better performance of a software. To keep doubling performance parallelism must double. It implies a huge effort in algorithmic development.

Thanks to `ffddm`, FreeFEM users have access to high-level functionalities for specifying and solving their finite element problems in parallel. The first task handled by `ffddm` is the data distribution among the processors. This is done via an overlapping domain decomposition and a related distributed linear algebra. Then, solving a linear system is possible either via an interface to the parallel MUMPS solver or by using domain decomposition methods as preconditioners to the GMRES Krylov method. The `ffddm` framework makes it easy to use scalable Schwarz methods enhanced by a coarse space correction built either from a coarse mesh or a GenEO (Generalized Eigenvalue in the Overlap) coarse space, see also the book An Introduction to Domain Decomposition Methods: algorithms, theory, and parallel implementation. State-of-the-art three level methods are also implemented in `ffddm`.

The `ffddm` framework is entirely written in the FreeFEM language and the 'idp' scripts can be found here ('ffddm*.idp' files). It makes it also a very good tool for learning and prototyping domain decomposition methods without compromising efficiency.

`ffddm` can also act as a wrapper for the HPDDM library. HPDDM is an efficient implementation of various domain decomposition methods and a variety of Krylov subspace algorithms, with advanced block and recycling methods for solving sequences of linear systems with multiple right-hand sides: GMRES and Block GMRES, CG, Block CG,

and Breakdown-Free Block CG, GCRO-DR and Block GCRO-DR. For more details on how to use HPDDM within `ffddm`, see *the ffddm documentation*.

Getting Started

```
macro dimension 2// EOM            // 2D or 3D
include "ffddm.idp"
mesh Th = square(50,50);     // global mesh
// Step 1: Decompose the mesh
ffddmbuildDmesh( P , Th , mpiCommWorld )
// Step 2: Define your finite element
macro def(u)  u // EOM
macro init(u) u // EOM
ffddmbuildDfespace( P , P , real , def , init , P2 )
// Step 3: Define your problem
macro grad(u) [dx(u), dy(u)] // EOM
macro Varf(varfName, meshName, VhName)
    varf varfName(u,v) = int2d(meshName)(grad(u)'* grad(v)) + int2d(meshName)(1*v)
                       + on(1, u = 0);  // EOM
ffddmsetupOperator( P , P , Varf )
PVhi ui, bi;
ffddmbuildrhs( P , Varf , bi[] )
// Step 4: Define the one level DD preconditioner
ffddmsetupPrecond( P , Varf )
// Step 5: Define the two-level GenEO Coarse Space
ffddmgeneosetup( P , Varf )
// Step 6: Solve the linear system with GMRES
PVhi x0i = 0;
ui[] = PfGMRES(x0i[], bi[], 1.e-6, 200, "right");
ffddmplot(P, ui, "u")
Pwritesummary
```

This example solves a Laplace problem in 2D in parallel with a two-level GenEO domain decomposition method. To try this example, just copy and paste the script above in a file 'test.edp' and run it on 2 cores with

```
ff-mpirun -np 2 test.edp -glut ffglut
```

3.9.1 Domain Decomposition (DD)

When the size of a three dimensional problem is large (whatever it means), it is necessary to distribute data among several processors especially for solving linear systems. A natural way is to do it via domain decomposition.

Mesh Decomposition

The starting point is a collection of N sub-meshes $(Th_i)_{i=1}^N$ that together form a global mesh

$$Th := \cup_{i=1}^N Th_i \,.$$

These meshes may be overlapping or not. This decomposition induces a natural decomposition of the global finite element space Vh on Th into N local finite element spaces $(Vh_i)_{i=1}^N$ each of them defined on Th_i.

Note By global, we mean that the corresponding structure can be refered to in the code (most often only) by its local values. In computer science term, it corresponds to a distributed data where each piece of data is stored by a MPI process.

Distributed Linear Algebra

The domain decomposition induces a natural decomposition of the set of the global degrees of freedom (d.o.f.) $\mathcal{N}$ of the finite element space Vh into the N subsets of d.o.f.'s $(\mathcal{N}_i)_{i=1}^N$ each associated with the local finite element space Vh_i. We have thus

$$\mathcal{N} = \cup_{i=1}^N \mathcal{N}_i \,,$$

but with duplications of some of the d.o.f.'s.

Associated with this decomposition of the set of d.o.f.'s $\mathcal{N}$, a *distributed vector* is a collection of local vectors $(\mathbf{V}_i)_{1\leq i\leq N}$ so that the values on the duplicated d.o.f.'s are the same.

Note: In mathematical terms, it can be described as follows for a real valued problem. Let R_i be the restriction operator from $\mathbb{R}^{\#\mathcal{N}}$ to $\mathbb{R}^{\#\mathcal{N}_i}$, where $\#\mathcal{N}_i$ denotes the number of elements of $\mathcal{N}_i$. A collection of local vectors $(\mathbf{V}_i)_{1\leq i\leq N} \in \Pi_{i=1}^N \mathbb{R}^{\#\mathcal{N}_i}$ is a distributed vector iff there exists a global vector $\mathbf{V} \in \mathbb{R}^{\#\mathcal{N}}$ such that for all subset $1 \leq i \leq N$, we have:

$$\mathbf{V}_i = R_i \, \mathbf{V} \,.$$

We will also say that the collection of local vectors $(\mathbf{V}_i)_{1\leq i\leq N}$ is consistent. For a complex valued problem, simply replace $\mathbb{R}$ with $\mathbb{C}$.

Partition of Unity Matrices (POUM)

Let $(D_i)_{1\leq i\leq N}$ be square diagonal matrices of size $\#\mathcal{N}_i$ which form a partition of unity in the sense that:

$$Id = \sum_{i=1}^N R_i^T \, D_i \, R_i \text{ in } \mathbb{R}^{\#\mathcal{N}\times\#\mathcal{N}} \,.$$

For instance if a degree of freedom is shared by k subdomains defining the corresponding entry of the diagonal matrix D to be $1/k$ yields partition of unity matrices. The matrices R_i and D_i are the heart of distributed linear algebra.

Distributed scalar product

For two global vectors $\mathbf{U}$ and $\mathbf{V}$ of size $\#\mathcal{N}$, the formula for the scalar product $\mathbf{V}^T\,\mathbf{U} = (\mathbf{U},\,\mathbf{V})$ in terms of their distributed vector counterparts is:

$$(\mathbf{U},\mathbf{V}) = \left(\mathbf{U}, \sum_{i=1}^N R_i^T D_i R_i \mathbf{V}\right) = \sum_{i=1}^N (R_i\mathbf{U}, D_i R_i \mathbf{V}) = \sum_{i=1}^N (\mathbf{U}_i, D_i \mathbf{V}_i) \,.$$

Local scalar products are performed concurrently. Thus, the implementation is parallel except for the sum which corresponds to a MPI_Reduce call across the N MPI processes. Note also that the implementation relies on the knowledge of a partition of unity so that the FreeFEM syntax is `dscalprod(Di,u,v)` or equivalently `pr#scalprod(u,v)` where `pr` is a user defined prefix that refers to the domain decomposition and thus implicitely also to the partition of unity.

Update

From a collection of local vectors $(\mathbf{U}_i)_{1\leq i\leq N}$, it is possible ensure consistency of the duplicated data and thus creating a distributed vector $(\mathbf{V}_i)_{1\leq i\leq N}$ by calling the function `pr#update(Ui, TRUE)` where `pr` is a user defined prefix

that refers to the domain decomposition. This function performs the following operation for all $1 \leq i \leq N$:

$$\mathbf{V}_i \leftarrow R_i \sum_{j=1}^{N} R_j^T D_j \mathbf{U}_j$$

Note: The implementation corresponds to

$$\mathbf{V}_i := R_i \sum_{j=1}^{N} R_j^T D_j \mathbf{U}_j = D_i \mathbf{U}_i + \sum_{j \in \mathcal{O}(i)} R_i \, R_j^T \, D_j \mathbf{U}_j$$

where $\mathcal{O}(i)$ is the set of neighbors of subdomain i. Therefore, the matrix vector product is computed in three steps: - concurrent computing of $D_j \mathbf{U}_j$ for all $1 \leq j \leq N$; - neighbor to neighbor MPI-communications ($R_i \, R_j^T$) ; - concurrent sum of neighbor contributions.

Distributed Matrix and Vector resulting from a variational formulation

The discretization of a variational formulation on the global mesh Th yields a global matrix A and a global right hand side $\mathbf{RHS}$. Thanks to the sparsity of finite element matrices for partial differential equations and thanks to the overlap between subdomains, the knowledge of the local matrix $R_i A R_i^T$ on each subdomain $1 \leq i \leq N$ is sufficient to perform the matrix-vector product $A \times \mathbf{U}$ for any global vector $\mathbf{U}$. Once the problem has been set up by a call to `ffddmsetupOperator(myprefix, myFEprefix, myVarf)`, the matrix-vector product is performed by calling the function `pr#A(Ui)` where `pr` is a user defined prefix that refers to the problem at hand which itself implicitly refers to the triplet (domain decomposition, finite element, variational formulation). See more on problem defintion in this *documentation* and more on distributed linear algebra in chapter 8 of "An Introduction to Domain Decomposition Methods: algorithms, theory and parallel implementation" SIAM 2015.

Distributed Linear Solvers

In many cases, we are interested in the solution of the problem in terms of the vector of d.o.f.'s $\mathbf{X}$ that satisfies:

$$A \, \mathbf{X} = \mathbf{RHS} \, .$$

`ffddm` offers two parallel solvers: *direct factorization* and *Schwarz* domain decomposition methods.

Distributed Direct Solvers

In order to benefit from the sparsity of the matrix arising from a finite element discretization of a partial differential equation, a variant of Gauss elimination, the frontal method, that automatically avoids a large number of operations involving zero terms was developed. A frontal solver builds a LU or Cholesky decomposition of a sparse matrix given as the assembly of element matrices by eliminating equations only on a subset of elements at a time. This subset is called the *front* and it is essentially the transition region between the part of the system already finished and the part not touched yet. These methods are basically sequential since the unknowns are processed the one after another or one front after another. In order to benefit from multicore processors, a multifrontal solver is an improvement of the frontal solver that uses several independent fronts at the same time. The fronts can be worked on by different processors, which enables parallel computing. `ffddm` provides an interface to the parallel sparse direct solver MUMPS.

Schwarz methods

We consider the solve of the equation $A \, \mathbf{X} = \mathbf{RHS}$ by a flexible GMRES method preconditioned by domain decomposition methods.

Restricted Additive Schwarz (RAS)

The RAS preconditioner reads:

$$M_{RAS}^{-1} := \sum_{j=1}^{N} R_j^T D_j (R_j \, A \, R_j^T)^{-1} R_j \, .$$

Let A_i denote the local matrix $(R_i \, A \, R_i^T)$. The application of the operator M_{RAS}^{-1} to a distributed right hand side $(\mathbf{RHS}_i)_{i=1}^N$ consists in computing:

$$R_i \sum_{j=1}^{N} R_j^T \, D_j \, A_j^{-1} \, \mathbf{RHS}_j = D_i \, A_i^{-1} \, \mathbf{RHS}_i + \sum_{j \in \mathcal{O}(i)} (R_i \, R_j^T) \, D_j \, A_j^{-1} \, \mathbf{RHS}_j \, .$$

This task is performed by first solving concurrently on all subdomains a linear system for $\mathbf{Y}_j$ for all $1 \leq j \leq N$:

$$A_j \, \mathbf{Y}_j = \mathbf{RHS}_j \, .$$

Each local vector $\mathbf{Y}_j$ is weighted by the partition of unity matrix D_j. Then data transfers between neighboring subdomains implement the $R_i \, R_j^T \, D_j \, \mathbf{Y}_j$ formula. The contribution from neighboring subdomains are summed locally. This pattern is very similar to that of the *update* procedure.

Optimized Restricted Additive Schwarz (ORAS)

The ORAS preconditioner may be seen as a variant of the RAS preconditioner. It reads:

$$M_{RAS}^{-1} := \sum_{j=1}^{N} R_j^T \, D_j \, B_j^{-1} \, R_j$$

where B_j are local matrices of size $\#\mathcal{N}_j \times \#\mathcal{N}_j$ for $1 \leq j \leq N$. This variant is very useful when dealing with wave propagation phenomena such as Helmholtz problems in acoustics or Maxwell system in the frequency domain for electromagnetism. Defining B_j as the discretization of the physical equation with impedance conditions on the boundary of the subdomain has been proved to be a good choice.

Two level methods

The RAS and ORAS methods are called a one-level method in the sense that sub-domains only interact with their direct neighbors. For some problems such as Darcy problems or static elasticity problems and when the number of subdomains is large, such one-level methods may suffer from a slow convergence. The fix is to add to the preconditioner an auxiliary coarse problem that couples all subdomains at each iteration and is inexpensive to calculate.

In mathematical terms, we first choose a full rank rectangular matrix $Z \in \mathbb{R}^{\#\mathcal{N} \times NC}$ where $NC \ll \#\mathcal{N}$ denotes the dimension of the coarse space spanned by the columns of Z. We also pick a coarse matrix $A_C \in \mathbb{R}^{N_C \times N_C}$. A generic one-level method preconditioner M_1^{-1} is enriched by a solve on the coarse space. The simplest correction formula is additive:

$$M_2^{-1} := Z \, A_C^{-1} \, Z^T + M_1^{-1}$$

Other correction formulas are given in *documentation*.

We consider two ways to build Z and thus the coarse space and the coarse problem A_C, see below *Coarse Mesh* and *GenEO*

Coarse Mesh

A first possibility is to discretize the problem on a coarse mesh, following the same principle as multi-grid methods. For 3-D problems, a coarsening of the mesh size by a factor 2, reduces by a factor $2^3 = 8$ the size of the coarse problem which is then easier to solve by a direct method. Then, Z is the interpolation matrix from the coarse finite element space to the fine one.

GenEO

For highly heterogeneous or anisotropic problems, two level methods based on coarse meshes might fail and a more sophisticated construction must be used. A provable robust coarse space called GenEO is built by first solving the following local generalized eigenvalue problem in parallel for each subdomain $1 \leq i \leq N$, where A_i^{Neu} denotes the local matrix resulting from the variational formulation:

$$D_i A_i D_i \, V_{i,k} = \lambda_{i,k} \, A_i^{\text{Neu}} \, V_{i,k}$$

The eigenvectors selected to enter the coarse space correspond to eigenvalues $\lambda_{i,k} \geq \tau$, where the threshold parameter τ is user-defined. The precise formulas are given in this *documentation*. From a mathematical point of view, it has been proved that for a symmetric positive definite matrix A, the spectrum of the preconditioned by the two-level method with a GenEO coarse space lies in the interval $[\frac{1}{1 + k_1 \, \tau}, k_0]$.

Note A heuristic that justifies this construction is as follows. We first introduce the Additive Schwarz method (ASM) which can be seen as a symmetrized variant of the RAS preconditioner:

$$M_{ASM}^{-1} := \sum_{j=1}^{N} R_j^T \, A_j^{-1} R_j \, .$$

It can be proved that the lower bound for the eigenvalue of $M_{ASM}^{-1} \, A$ is close to zero (which is bad for convergence) whereas the upper bound depends only on the number of neigbors of a subdomain (which is good for convergence).

Second, we also introduce the following preconditioner M_{NN}^{-1}:

$$M_{NN}^{-1} := \sum_{1 \leq j \leq N} D_i \, (A_j^{\text{Neu}})^{-1} D_j \, .$$

We have a very good lower bound for the preconditioned operator $M_{NN}^{-1} \, A$ that does not depend on the number of subdomains but only on the maximum multiplicity of intersections k_1 (which is good for convergence). But the upper bound for this preconditioner is very large (which is bad for convergence).

Now, if we compare formulas for M_{NN}^{-1} and M_{ASM}^{-1}, we may suspect that vectors $\mathbf{V}_{ik}$ for which $D_i \, (A_i^{\text{Neu}})^{-1} \, D_i \, \mathbf{V}_{ik}$ and $A_i^{-1} \, \mathbf{V}_{ik}$ have very different values are responsible for the slow convergence and should contribute to the coarse space. This is a way to interpret the above generalized eigenvalue problem which controls the lower bound of the two-level preconditioned system.

3.9.2 ffddm documentation

Minimal example

```
macro dimension 3// EOM            // 2D or 3D

include "ffddm.idp"

```

(continues on next page)

(continued from previous page)

```
int[int] LL = [2,2, 1,2, 2,2];
mesh3 ThGlobal = cube(10, 10, 10, [x, y, z], label = LL);      // global mesh

macro grad(u) [dx(u), dy(u), dz(u)]// EOM    // three-dimensional gradient

macro Varf(varfName, meshName, VhName)
    varf varfName(u,v) = int3d(meshName)(grad(u)'* grad(v)) + int3d(meshName)(v) +
↪on(1, u = 1.0);
// EOM

// Domain decomposition
ffddmbuildDmesh( Lap , ThGlobal , mpiCommWorld )

macro def(i)i// EOM                        // scalar field definition
macro init(i)i// EOM                       // scalar field initialization
ffddmbuildDfespace( Lap , Lap , real , def , init , P1 )

ffddmsetupOperator( Lap ,Lap , Varf )

real[int] rhsi(0);
ffddmbuildrhs( Lap , Varf , rhsi )

LapVhi def(ui);

//Direct solve
ui[] = Lapdirectsolve(rhsi);

Lapwritesummary

ffddmplot(Lap,ui,"u");
```

Overlapping mesh decomposition

```
ffddmbuildDmesh(pr,Th,comm)
```

decomposes the mesh **Th** into overlapping submeshes. The mesh will be distributed over the mpi ranks of communicator **comm**. This will create and expose variables whose names will be prefixed by **pr**, see below (# is the concatenation operator). The way the initial mesh **Th** is partitioned depends on the value of *ffddmpartitioner*.

The size of the overlap between subdomains (its width in terms of number of mesh elements) is given by *ffddmoverlap*.

The level of refinement of the resulting submeshes with respect to the input mesh **Th** is given by *ffddmsplit*.

If *ffddmexclude* $\neq 0$, the first *ffddmpCS* mpi ranks of **comm** will be excluded from the spatial domain decomposition, in order to dedicate them later to the coarse problem (for two-level preconditioners).

The label of the new border of the submeshes (the interface between the subdomains) is given by *ffddminterfacelabel*.

defines:

- `int pr#npart` number of subdomains for this decomposition; should be equal to mpiSize(**comm**) - *ffddmexclude* * *ffddmpCS*
- `meshN[int] pr#aTh` array (size `pr#npart`) of local meshes of the subdomains. In the standard parallel case, only the local mesh for this mpi rank `pr#aTh[mpiRank(pr#commddm)]` is defined (unless this mpi rank is excluded from the spatial domain decomposition, i.e. `prmesh#excluded` = 1, see below). In the sequential case, all local meshes are defined.

- `meshN pr#Thi` the local mesh of the subdomain for this mpi rank, i. e. `pr#aTh[mpiRank(pr#commddm)]` in the parallel case - `int pr#numberIntersection` the number of neighbors for this mpi rank
- `int[int] pr#arrayIntersection` the list of neighbor ranks in `pr#commddm` for this mpi rank
- `int pr#pCS` equal to *ffddmpCS*
- `int pr#exclude` equal to *ffddmexclude*
- `int pr#excluded` *true* if *ffddmexclude* is *true* ($\neq 0$) and mpiRank(**comm**) < `pr#pCS`. In this case, this mpi rank will be excluded from the spatial domain decomposition and will only work on the coarse problem.
- `mpiComm pr#commddm` mpi communicator for ranks participating in the spatial domain decomposition (ranks 0 to `pr#npart`-1 in **comm** if `pr#exclude` is *false*, ranks `pr#pCS` to `pr#pCS`+`pr#npart`-1 otherwise)
- `mpiComm pr#commCS` mpi communicator for ranks participating in the assembly and resolution of the coarse problem for two-level preconditioners (ranks 0 to `pr#pCS` - 1 in **comm**)
- `mpiComm pr#commself` self mpi communicator (this mpi rank only), used for factorizing local matrices

Remark for sequential use (see *-seqddm*):

- `meshN[int] pr#aTh` array (size `pr#npart`) of local meshes of the subdomains

Local finite element spaces

```
ffddmbuildDfespace(pr,prmesh,scalar,def,init,Pk)
```

builds the local finite element spaces and associated distributed operators on top of the mesh decomposition **prmesh**. This will create and expose variables whose names will be prefixed by **pr**, see below. It is assumed that *ffddmbuildDmesh* has already been called with prefix **prmesh** in order to build the mesh decomposition.

The local finite element spaces of type **Pk** (where **Pk** is the type of finite element: P1, [P2,P2,P1], ...) are defined on the local meshes of the subdomains based on the mesh decomposition previously created with prefix **prmesh**.

scalar determines the type of data for this finite element: *real* or *complex*.

Two macros, **def** and **init**, are needed: **def** specifies how to define a finite element function in the finite element space **Pk**, and **init** specifies how to interpolate a scalar function onto the (possibly multiple) components of **Pk**. Two examples are given below:

For scalar P2 finite elements and complex-valued problems:

```
macro def(u) u// EOM
macro init(u) u// EOM
ffddmbuildDfespace(myFEprefix,mymeshprefix,complex,def,init,P2)
```

For vectorial [P2,P2,P1] finite elements and real-valued problems:

```
macro def(u) [u, u#B, u#C]// EOM
macro init(u) [u, u, u]// EOM
ffddmbuildDfespace(myFEprefix,mymeshprefix,real,def,init,[P2,P2,P1])
```

In practice, this builds the necessary distributed operators associated to the finite element space: the local partition of unity functions $(D_i)_{i=1,\dots,N}$ (see `pr#Dk` and `pr#Dih` below) as well as the function `pr#update` (see below) which synchronizes local vectors $(u_i)_{i=1,\dots,N}$ between neighboring subdomains, performing the equivalent of $u_i = R_i(\sum_{j=1}^N R_j^T u_j)$ or $u_i = R_i(\sum_{j=1}^N R_j^T D_j u_j)$ in a distributed parallel environment.

`pr#scalprod` (see below) performs the parallel scalar product for vectors defined on this finite element.

defines:

- `pr#prmesh` macro, saves the parent prefix **prmesh** of the mesh decomposition
- `pr#K` macro, saves the type of data **scalar** for this finite element space (*real* or *complex*)
- `func pr#fPk` saves the type of finite element **Pk**, e.g. *P1, [P2,P2,P1], ...*
- `fespace pr#Vhi` the local finite element space for this mpi rank, defined on the local mesh `prmesh#Thi`
- `int pr#Ndofglobal` the total number of degrees of freedom n for this finite element discretization
- `pr#mdef` macro, saves the macro **def** giving the definition of a finite element function in the finite element space **Pk**
- `pr#minit` macro, saves the macro **init** specifying how to interpolate a scalar function onto the (possibly multiple) components of a finite element function of **Pk**. This is used to create the local partition of unity function in `pr#Vhi`, by interpolating the local P1 partition of unity function onto the components of `pr#Vhi`. For non Lagrange finite element spaces (e.g. *RT0*, *Edge03d*, ...), see *ffddmbuildDfespaceEdge*.
- `pr#K[int][int] pr#Dk` array (size `prmesh#npart`) of local partition of unity vectors in the subdomains, equivalent to $(D_i)_{i=1,\dots,N}$. In the standard parallel case, only the local partition of unity vector for this mpi rank `pr#Dk[mpiRank(prmesh#commddm)]` is defined (unless this mpi rank is excluded from the spatial domain decomposition, i. e. `prmesh#excluded` = 1). In the sequential case, all local partition of unity vectors are defined.
- `matrix<pr#K>[int] pr#Dih` array (size `prmesh#npart`) similar to `pr#Dk` but in *matrix* form, allowing for easier *matrix-matrix* multiplications. `pr#Dih[i]` is a diagonal matrix, with the diagonal equal to `pr#Dk[i]`.
- `fespace pr#Vhglob` the global finite element space defined on the global mesh `prmesh#Thglob`. Defined only if *-noGlob* is not used.
- `matrix<pr#K>[int] pr#Rih` array (size `prmesh#npart`) of restriction matrices from the global finite element space to the local finite element spaces on the local submeshes of the subdomains. In the standard parallel case, only the restriction matrix for this mpi rank `pr#Rih[mpiRank(prmesh#commddm)]` is defined (unless this mpi rank is excluded from the spatial domain decomposition, i. e. `prmesh#excluded` = 1). In the sequential case, all restriction matrices are defined. The restriction matrices `pr#Rih` are defined only if *-noGlob* is not used.
- `func int pr#update(scalar[int] ui, bool scale)` The function `pr#update` synchronizes the local vector *ui* between subdomains by exchanging the values of *ui* shared with neighboring subdomains (in the overlap region) using point-to-point MPI communications. If *scale* is *true*, *ui* is multiplied by the local partition of unity beforehand. This is equivalent to $u_i = R_i(\sum_{j=1}^N R_j^T u_j)$ when *scale* is *false* and $u_i = R_i(\sum_{j=1}^N R_j^T D_j u_j)$ when *scale* is *true*.
- `func scalar pr#scalprod(scalar[int] ai, scalar[int] bi)` The function `pr#scalprod` computes the global scalar product of two vectors whose local restriction to the subdomain of this mpi rank are *ai* and *bi*. The result is computed as $\sum_{j=1}^N (D_j a_j, b_j)$.

Define the problem to solve

```
ffddmsetupOperator(pr,prfe,Varf)
```

builds the distributed operator associated to the variational problem given by **Varf**, on top of the distributed finite element space **prfe**. This will create and expose variables whose names will be prefixed by **pr**, see below. It is assumed that *ffddmbuildDfespace* has already been called with prefix **prfe** in order to define the distributed finite element space.

In practice, this builds the so-called local 'Dirichlet' matrices $A_i = R_i A R_i^T$, the restrictions of the global operator A to the subdomains (see `pr#aRd`below). The matrices correspond to the discretization of the bilinear form given by the macro **Varf**, which represents the abstract variational form of the problem. These matrices are then used to implement the action of the global operator A on a local vector (the parallel matrix-vector product with A), see `pr#A` below.

At this point, we already have the necessary data to be able to solve the problem with a parallel direct solver (*MUMPS*), which is the purpose of the function `pr#directsolve` (see below). See *ffddmbuildrhs* for building the right-hand side.

The macro **Varf** is required to have three parameters: the name of the variational form, the mesh, and the finite element space. The variational form given in this 'abstract' format will then be used by *ffddm* to assemble the discrete operators by setting the appropriate mesh and finite element space as parameters. An example is given below:

```
macro myVarf(varfName, meshName, VhName)
    varf varfName(u,v) = int3d(meshName)(grad(u)''* grad(v)) + on(1, u = 1.0);
// EOM

ffddmsetupOperator(myprefix,myFEprefix,myVarf)
```

Remark In this simple example, the third parameter *VhName* is not used. However, for more complex cases such as non-linear or time dependent problems where the problem depends on a solution computed at a previous step, it is useful to know for which discrete finite element space the variational form is being used. See for example TODO

defines:

- `pr#prfe` macro, saves the parent prefix **prfe** of the finite element space
- `int pr#verbosity` the level of verbosity for this problem, initialized with the value of *ffddmverbosity*
- `pr#writesummary` macro, prints a summary of timings for this problem, such as the time spent to assemble local matrices or solve the linear system.
- `matrix<prfe#K> pr#Aglobal` the global matrix A corresponding to the discretization of the variational form given by the macro **Varf** on the global finite element space `prfe#Vhglob`. Defined only in the sequential case.
- `matrix<prfe#K>[int] pr#aRd` array (size `prfe#prmesh#npart`) of so-called local 'Dirichlet' matrices in the subdomains; these are the restrictions of the global operator to the subdomains, equivalent to $A_i = R_i A R_i^T$ with A the global matrix corresponding to the discretization of the variational form given by the macro **Varf** on the global finite element space. In the standard parallel case, only the local matrix for this mpi rank `pr#aRd[mpiRank(prmesh#commddm)]` is defined (unless this mpi rank is excluded from the spatial domain decomposition, i. e. `prmesh#excluded` = 1). In the sequential case, all local matrices are defined.
- `func prfe#K[int] pr#A(prfe#K[int] &ui)` The function `pr#A` computes the parallel matrix-vector product, i.e. the action of the global operator A on the local vector u_i. The computation is equivalent to $R_i(\sum_{j=1}^N R_j^T D_j A_j u_j)$ and is performed in parallel using local matrices `pr#aRd` and the function `prfe#update`. In the sequential case, the global matrix `pr#Aglobal` is used instead.
- `func prfe#K[int] pr#directsolve(prfe#K[int]& rhsi)` The function `pr#directsolve` allows to solve the linear system $Ax = b$ in parallel using the parallel direct solver *MUMPS*. The matrix is given to *MUMPS* in distributed form through the local matrices `pr#aRd`. The input *rhsi* is given as a distributed vector (*rhsi* is the restriction of the global right-hand side b to the subdomain of this mpi rank, see *ffddmbuildrhs*) and the returned vector is local as well.

```
ffddmbuildrhs(pr,Varfrhs,rhs)
```

builds the right-hand side associated to the variational form given by **Varfrhs** for the problem corresponding to prefix **pr**. The resulting right-hand side vector **rhs** corresponds to the discretization of the abstract linear form given by the macro **Varfrhs** (see *ffddmsetupOperator* for more details on how to define the abstract variational form as a macro).

The input vector **rhs** is resized and contains the resulting local right-hand side $R_i b$, the restriction of the global right-hand side b to the subdomain of this mpi rank. In the sequential case, the global right-hand side vector b is assembled instead.

An example is given below:

```
macro myVarfrhs(varfName, meshName, VhName)
    varf varfName(u,v) = intN(meshName)(v) + on(1, u = 1.0);
// EOM

real[int] rhsi(0);
ffddmbuildrhs(myprefix,myVarfrhs,rhsi)
```

One level preconditioners

```
ffddmsetupPrecond(pr,VarfPrec)
```

builds the one level preconditioner for problem **pr**. This will create and expose variables whose names will be prefixed by **pr**, see below. It is assumed that *ffddmsetupOperator* has already been called with prefix **pr** in order to define the problem to solve.

In practice, this builds and performs the factorization of the local matrices used in the one level preconditioner. The local matrices depend on the choice of *ffddmprecond* and **VarfPrec**, see `pr#aR`below.

defines:

- `string pr#prec` equal to *ffddmprecond*. Sets the type of one level preconditioner M_1^{-1} to be used: "asm" (*Additive Schwarz*), "ras" (*Restricted Additive Schwarz*), "oras" (*Optimized Restricted Additive Schwarz*), "soras" (*Symmetric Optimized Restricted Additive Schwarz*) or "none" (no preconditioner).
- `matrix<pr#prfe#K>[int] pr#aR` array (size `prfe#prmesh#npart`) of local matrices used for the one level preconditioner. Each mpi rank of the spatial domain decomposition performs the LU (or LDL^T) factorization of the local matrix corresponding to its subdomain using the direct solver *MUMPS*.
  - If **VarfPrec** is not a previously defined macro (just put *null* for example), the matrices `pr#aR` are set to be equal to the so-called local 'Dirichlet' matrices `pr#aRd` (see *ffddmsetupOperator*). This is for the classical ASM preconditioner $M_1^{-1} = M_{\text{ASM}}^{-1} = \sum_{i=1}^N R_i^T A_i^{-1} R_i$ or classical RAS preconditioner $M_1^{-1} = M_{\text{RAS}}^{-1} = \sum_{i=1}^N R_i^T D_i A_i^{-1} R_i$ (it is assumed that *ffddmprecond* is equal to "asm" or "ras").
  - If **VarfPrec** is a macro, it is assumed that **VarfPrec** defines an abstract bilinear form (see *ffddmsetupOperator* for more details on how to define the abstract variational form as a macro).
    * If *ffddmprecond* is equal to "asm" or "ras", the matrices `pr#aR` will be assembled as local 'Dirichlet' matrices in the same manner as `pr#aRd`, but using the bilinear form defined by **VarfPrec** instead. This defines the ASM preconditioner as $M_1^{-1} = M_{\text{ASM}}^{-1} = \sum_{i=1}^N R_i^T (A_i^{\text{Prec}})^{-1} R_i$ and the RAS preconditioner as $M_1^{-1} = M_{\text{RAS}}^{-1} = \sum_{i=1}^N R_i^T D_i (A_i^{\text{Prec}})^{-1} R_i$, where $A_i^{\text{Prec}} = R_i A^{\text{Prec}} R_i^T$.
    * If *ffddmprecond* is equal to "oras" or "soras", the matrices `pr#aR` will correspond to the discretization of the variational form **VarfPrec** in the subdomains Ω_i. In particular, various boundary conditions can be imposed at the interface between subdomains (corresponding to mesh boundary of label *ffddminterfacelabel* set by the parent call to *ffddmbuildDmesh*), such as Optimized Robin boundary conditions. We note the ORAS preconditioner as $M_1^{-1} = M_{\text{ORAS}}^{-1} = \sum_{i=1}^N R_i^T D_i (B_i^{\text{Prec}})^{-1} R_i$ and the SORAS preconditioner as $M_1^{-1} = M_{\text{SORAS}}^{-1} = \sum_{i=1}^N R_i^T D_i (B_i^{\text{Prec}})^{-1} D_i R_i$.
- `func pr#prfe#K[int] pr#PREC1(pr#prfe#K[int] &ui)` The function `pr#PREC1` computes the parallel application of the one level preconditioner M_1^{-1}, i.e. the action of M_1^{-1} on the local vector u_i. In the sequential case, it computes the action of M_1^{-1} on a global vector. The action of the inverse of local matrices `pr#aRd` is computed by forward-backward substitution using their LU (or LDL^T) decomposition.

- `func pr#prfe#K[int] pr#PREC(pr#prfe#K[int] &ui)` The function `pr#PREC` corresponds to the action of the preconditioner M^{-1} for problem **pr**. It coincides with the one level preconditioner `pr#PREC1` after the call to *ffddmsetupPrecond*. If a second level is subsequently added (see the next section about *Two level preconditioners*), it will then coincide with the two level preconditioner M_2^{-1} (see `pr#PREC2level`).
- `func pr#prfe#K[int] pr#fGMRES(pr#prfe#K[int]& x0i, pr#prfe#K[int]& bi, real eps, int nbiter, string sprec)` The function `pr#fGMRES` allows to solve the linear system $Ax = b$ in parallel using the flexible GMRES method preconditioned by M^{-1}. The action of the global operator A is given by `pr#A`, the action of the preconditioner M^{-1} is given by `pr#PREC` and the scalar products are computed by `pr#scalprod`. More details are given in the section *Solving the linear system*.

Two level preconditioners

The main ingredient of a two level preconditioner is the so-called 'coarse space' matrix Z.

Z is a rectangular matrix of size $n \times n_c$, where usually $n_c \ll n$.

Z is used to build the 'coarse space operator' $E = Z^T AZ$, a square matrix of size $n_c \times n_c$. We can then define the 'coarse space correction operator' $Q = ZE^{-1}Z^T = Z(Z^T AZ)^{-1}Z^T$, which can then be used to enrich the one level preconditioner through a correction formula. The simplest one is the *additive* coarse correction: $M_2^{-1} = M_1^{-1} + Q$. See `pr#corr` below for all other available correction formulas.

There are multiple ways to define a relevant coarse space Z for different classes of problems. *ffddmgeneosetup* defines a coarse space correction operator by building the GenEO coarse space, while *ffddmcoarsemeshsetup* builds the coarse space using a coarse mesh.

After a call to either *ffddmgeneosetup* or *ffddmcoarsemeshsetup*, the following variables and functions are set up:

- `int pr#ncoarsespace` the size of the coarse space n_c.
- `string pr#corr` initialized with the value of *ffddmcorrection*. Specifies the type of coarse correction formula to use for the two level preconditioner. The possible values are:

"AD" :	*Additive*,	$M^{-1} = M_2^{-1} = M_1^{-1} + Q$
"BNN" :	*Balancing Neumann-Neumann*,	$M^{-1} = M_2^{-1} = (I - QA)M_1^{-1}(I - AQ) + Q$
"ADEF1" :	*Adapted Deflation Variant 1*,	$M^{-1} = M_2^{-1} = M_1^{-1}(I - AQ) + Q$
"ADEF2" :	*Adapted Deflation Variant 2*,	$M^{-1} = M_2^{-1} = (I - QA)M_1^{-1} + Q$
"RBNN1" :	*Reduced Balancing Variant 1*,	$M^{-1} = M_2^{-1} = (I - QA)M_1^{-1}(I - AQ)$
"RBNN2" :	*Reduced Balancing Variant 2*,	$M^{-1} = M_2^{-1} = (I - QA)M_1^{-1}$
"none" :	*no coarse correction*,	$M^{-1} = M_2^{-1} = M_1^{-1}$

- Note that *AD*, *ADEF1* and *RBNN2* only require one application of Q, while *BNN*, *ADEF2* and *RBNN1* require two. The default coarse correction is *ADEF1*, which is cheaper and generally as robust as *BNN* or *ADEF2*.
- `func pr#prfe#K[int] pr#Q(pr#prfe#K[int] &ui)` The function `pr#Q` computes the parallel application of the coarse correction operator Q, i.e. the action of $Q = ZE^{-1}Z^T$ on the local vector u_i. In the sequential case, it computes the action of Q on a global vector. The implementation differs depending on the method used to build the coarse space (with GenEO or using a coarse mesh), but the idea is the same: the action of the transpose of the distributed operator Z on the distributed vector u_i is computed in parallel, with the contribution of all subdomains being gathered in a vector of size n_c in the mpi process of rank 0. The action of the inverse of the coarse space operator E is then computed by forward-backward substitution using its LU (or LDL^T) decomposition previously computed by the first `pr#prfe#prmesh#pCS` ranks of the mpi communicator. The result is then sent back to all subdomains to perform the last application of Z and obtain the resulting local vector in each subdomain.
- `func pr#prfe#K[int] pr#PREC2level(pr#prfe#K[int] &ui)` The function `pr#PREC2level` computes the parallel application of the two level preconditioner M_2^{-1}, i.e. the action of M_2^{-1} on the local vector u_i. In the sequential case, it computes the action of M_2^{-1} on a global vector.

The two level preconditioner depends on the choice of the coarse correction formula which is determined by `pr#corr`, see above.

Building the GenEO coarse space

```
ffddmgeneosetup(pr,Varf)
```

This builds the GenEO coarse space for problem **pr**. This will create and expose variables whose names will be prefixed by **pr**, see below. It is assumed that *ffddmsetupPrecond* has already been called for prefix **pr** in order to define the one level preconditioner for problem **pr**. The GenEO coarse space is $Z = (R_i^T D_i V_{i,k})_{\lambda_{i,k} \geq \tau}^{i=1,\ldots,N}$, where $V_{i,k}$ are eigenvectors corresponding to eigenvalues $\lambda_{i,k}$ of the following local generalized eigenvalue problem in subdomain i:

$D_i A_i D_i V_{i,k} = \lambda_{i,k} A_i^{\text{Neu}} V_{i,k}$,

where A_i^{Neu} is the local Neumann matrix of subdomain i (with Neumann boundary conditions at the subdomain interface).

In practice, this builds and factorizes the local Neumann matrices A_i^{Neu} corresponding to the abstract bilinear form given by the macro **Varf** (see *ffddmsetupOperator* for more details on how to define the abstract variational form as a macro). In the GenEO method, the abstract bilinear form **Varf** is assumed to be the same as the one used to define the problem **pr** through the previous call to *ffddmsetupOperator*. The local generalized eigenvalue problem is then solved in each subdomain to find the eigenvectors $V_{i,k}$ corresponding to the largest eigenvalues $\lambda_{i,k}$ (see `pr#Z` below). The number of computed eigenvectors ν is given by *ffddmnu*. The eigenvectors selected to enter Z correspond to eigenvalues $\lambda_{i,k}$ larger than τ, where the threshold parameter τ is given by *ffddmtau*. If *ffddmtau* $= 0$, all *ffddmnu* eigenvectors are selected. Finally, the coarse space operator $E = Z^T A Z$ is assembled and factorized (see `pr#E` below).

defines:

- `pr#prfe#K[int][int] pr#Z` array of local eigenvectors $Z_{i,k} = D_i V_{i,k}$ obtained by solving the local generalized eigenvalue problem above in the subdomain of this mpi rank using *Arpack*. The number of computed eigenvectors ν is given by *ffddmnu*. The eigenvectors selected to enter Z correspond to eigenvalues $\lambda_{i,k}$ larger than τ, where the threshold parameter τ is given by *ffddmtau*. If *ffddmtau* $= 0$, all *ffddmnu* eigenvectors are selected.
- `matrix<pr#prfe#K> pr#E` the coarse space operator $E = Z^T A Z$. The matrix `pr#E` is assembled in parallel and is factorized by the parallel direct solver *MUMPS* using the first `pr#prfe#prmesh#pCS` ranks of the mpi communicator, with mpi rank 0 as the master process. The number of mpi processes dedicated to the coarse problem is set by the underlying mesh decomposition of problem **pr**, which also specifies if these mpi ranks are excluded from the spatial decomposition or not. These parameters are set by *ffddmpCS* and *ffddmexclude* when calling *ffddmbuildDmesh* (see *ffddmbuildDmesh* for more details).

Building the coarse space from a coarse mesh

```
ffddmcoarsemeshsetup(pr,Thc,VarfEprec,VarfAprec)
```

builds the coarse space for problem **pr** from a coarse problem which corresponds to the discretization of a variational form on a coarser mesh **Thc** of Ω. This will create and expose variables whose names will be prefixed by **pr**, see below. It is assumed that *ffddmsetupPrecond* has already been called for prefix **pr** in order to define the one level preconditioner for problem **pr**. The abstract variational form for the coarse problem can differ from the original problem **pr** and is given by macro **VarfEprec** (see *ffddmsetupOperator* for more details on how to define the abstract variational form as a macro). For example, absorption can be added in the preconditioner for wave propagation problems, see examples for Helmholtz and Maxwell equations in the *Examples* section.

The coarse space Z corresponds to the interpolation operator from the coarse finite element space to the original finite element space of the problem. Thus, the coarse space operator $E = Z^T A^{\text{Eprec}} Z$ corresponds to the matrix of the problem given by **VarfEprec** discretized on the coarse mesh **Thc** and is assembled as such.

Similarly, **VarfAprec** specifies the global operator involved in multiplicative coarse correction formulas. For example, $M_{2,\text{ADEF1}}^{-1} = M_1^{-1}(I - A^{\text{Aprec}}Q) + Q$ (where $Q = ZE^{-1}Z^T$). A^{Aprec} defaults to A if **VarfAprec** is not a valid macro (you can put *null* for example).

defines:

- `meshN pr#ThCoarse` the coarse mesh **Thc**
- `fespace pr#VhCoarse` the coarse finite element space of type `pr#prfe#fPk` defined on the coarse mesh `pr#ThCoarse`
- `matrix<pr#prfe#K> pr#AglobEprec` the global matrix A^{Aprec} corresponding to the discretization of the variational form given by the macro **VarfAprec** on the global finite element space `pr#prfe#Vhglob`. Defined only in the sequential case. `pr#AglobEprec` is equal to `pr#Aglobal` if **VarfAprec** is not a valid macro.
- `matrix<pr#prfe#K> pr#aRdEprec` the local 'Dirichlet' matrix corresponding to **VarfAprec**; it is the local restriction of the global operator A^{Aprec} to the subdomain, equivalent to $A_i^{\text{Aprec}} = R_i A^{\text{Aprec}} R_i^T$ with A^{Aprec} the global matrix corresponding to the discretization of the variational form given by the macro **VarfAprec** on the global finite element space. Defined only if this mpi rank is not excluded from the spatial domain decomposition, i. e. `prmesh#excluded` = 0. `pr#aRdEprec` is equal to `pr#aRd[mpiRank(prmesh#commddm)]` if **VarfAprec** is not a valid macro.
- `func pr#prfe#K[int] pr#AEprec(pr#prfe#K[int] &ui)` The function `pr#AEprec` computes the parallel matrix-vector product, i.e. the action of the global operator A^{Aprec} on the local vector u_i. The computation is equivalent to $R_i(\sum_{j=1}^N R_j^T D_j A_j^{\text{Aprec}} u_j)$ and is performed in parallel using local matrices `pr#aRdEprec` and the function `pr#prfe#update`. In the sequential case, the global matrix `pr#AglobEprec` is used instead.
- `matrix<pr#prfe#K> pr#ZCM` the interpolation operator Z from the coarse finite element space `pr#VhCoarse` to the global finite element space `pr#prfe#Vhglob`. Defined only in the sequential case.
- `matrix<pr#prfe#K> pr#ZCMi` the local interpolation operator Z_i from the coarse finite element space `pr#VhCoarse` to the local finite element space `pr#prfe#Vhi`. Defined only if this mpi rank is not excluded from the spatial domain decomposition, i. e. `prmesh#excluded` = 0. `pr#ZCMi` is used for the parallel application of Z and Z^T.
- `matrix<pr#prfe#K> pr#ECM` the coarse space operator $E = Z^T A^{\text{Eprec}} Z$. The matrix `pr#ECM` is assembled by discretizing the variational form given by **VarfEprec** on the coarse mesh and factorized by the parallel direct solver *MUMPS* using the first `pr#prfe#prmesh#pCS` ranks of the mpi communicator, with mpi rank 0 as the master process. The number of mpi processes dedicated to the coarse problem is set by the underlying mesh decomposition of problem **pr**, which also specifies if these mpi ranks are excluded from the spatial decomposition or not. These parameters are set by *ffddmpCS* and *ffddmexclude* when calling *ffddmbuildDmesh* (see *ffddmbuildDmesh* for more details).

Solving the linear system

```
func pr#prfe#K[int] pr#fGMRES(pr#prfe#K[int]& x0i, pr#prfe#K[int]& bi, real eps, int␣
↪itmax, string sp)
```

solves the linear system for problem **pr** using the flexible GMRES algorithm with preconditioner M^{-1} (corresponding to `pr#PREC`). Returns the local vector corresponding to the restriction of the solution to `pr#prfe#Vhi`. **x0i** and **bi** are local distributed vectors corresponding respectively to the initial guess and the right-hand side (see *ffddmbuildrhs*). **eps** is the stopping criterion in terms of the relative decrease in residual norm. If **eps** < 0, the residual norm itself is

used instead. **itmax** sets the maximum number of iterations. **sp** selects between the `"left"` or `"right"` preconditioning variants: *left* preconditioned GMRES solves $M^{-1}Ax = M^{-1}b$, while *right* preconditioned GMRES solves $AM^{-1}y = b$ for y, with $x = M^{-1}y$.

Using *HPDDM* within *ffddm*

ffddm allows you to use **HPDDM** to solve your problem, effectively replacing the **ffddm** implementation of all parallel linear algebra computations. **ffddm** can then be viewed as a finite element interface for **HPDDM**.

You can use **HPDDM** features unavailable in **ffddm** such as advanced Krylov subspace methods implementing block and recycling techniques.

To switch to **HPDDM**, simply define the macro `pr#withhpddm` before using *ffddmsetupOperator*. You can then pass **HPDDM** options with command-line arguments or directly to the underlying **HPDDM** operator `pr#hpddmOP`:

```
macro PBwithhpddm()1 // EOM
ffddmsetupOperator( PB , FE , Varf )
set(PBhpddmOP,sparams="-hpddm_krylov_method gcrodr");
```

You can also choose to replace only the Krylov solver, by defining the macro `pr#withhpddmkrylov` before using *ffddmsetupOperator*. Doing so, a call to `pr#fGMRES` will call the **HPDDM** Krylov solver, with **ffddm** providing the operator and preconditioner through `pr#A` and `pr#PREC`.

An example can be found in **Helmholtz-2d-HPDDM-BGMRES.edp**, see the *Examples* section.

3.9.3 Parameters

Command-line arguments

- `-ffddm_verbosity N`, the level of verbosity of **ffddm**, see *ffddmverbosity* (default 3).
- `-seqddm N` use **ffddm** in sequential mode, with N the number of subdomains.
- `-noGlob` if present, do not define any global quantity (such as saving the global mesh for plotting or building the global restriction matrices). Cannot be used in sequential mode or with plotting.
- `-ffddm_partitioner N` specifies how to partition the initial domain, see *ffddmpartitioner* (default 1, *metis*).
- `-ffddm_overlap N` specifies the width of the overlap region between subdomains, see *ffddmoverlap* (default 1).
- `-ffddm_master_p N`, number of master processes for the coarse problem (for two level preconditioners), see *ffddmpCS* (default 1).
- `-ffddm_master_exclude 0|1` exclude master processes from the domain decomposition, see *ffddmexclude* (default 0).
- `-ffddm_split N`, level of refinement of the local submeshes with respect to the initial global mesh, see *ffddmsplit* (default 1).
- `-ffddm_schwarz_method S`, specifies the type of one level preconditioner M_1^{-1}: "asm" (*Additive Schwarz*), "ras" (*Restricted Additive Schwarz*), "oras" (*Optimized Restricted Additive Schwarz*), "soras" (*Symmetric Optimized Restricted Additive Schwarz*) or "none" (no preconditioner), see *ffddmprecond* (default "ras").
- `-ffddm_geneo_nu N`, number of local eigenvectors to compute in each subdomain when solving the local generalized eigenvalue problem for the GenEO method, see *ffddmnu* (default 20).

- `-ffddm_geneo_threshold R`, threshold parameter for selecting local eigenvectors when solving the local generalized eigenvalue problems for the GenEO method, see *ffddmtau* (default 0.5). If the command-line parameter **-ffddm_geneo_nu N** is used, then *ffddmtau* is initialized to 0.
- `-ffddm_schwarz_coarse_correction S`, specifies the coarse correction formula to use for the two level preconditioner: "AD" (*Additive*), "BNN" (*Balancing Neumann-Neumann*), "ADEF1" (*Adapted Deflation Variant 1*), "ADEF2" (*Adapted Deflation Variant 2*), "RBNN1" (*Reduced Balancing Variant 1*), "RBNN2" (*Reduced Balancing Variant 2*) or "none" (no coarse correction), see *ffddmcorrection* (default "ADEF1").

Global parameters

- `ffddmverbosity` initialized by command-line argument **-ffddm_verbosity N**, specifies the level of verbosity of **ffddm** (default 3).
- `ffddmpartitioner` initialized by command-line argument **-ffddm_partitioner N**, specifies how to partition the initial domain:
  - N=0: user-defined partition through the definition of a macro, see *ffddmbuildDmesh*
  - N=1: use the automatic graph partitioner *metis* (default)
  - N=2: use the automatic graph partitioner *scotch*
- `ffddmoverlap` initialized by command-line argument **-ffddm_overlap N**, specifies the number of layers of mesh elements in the overlap region between subdomains N >= 1 (default 1). **Remark** The actual width of the overlap region between subdomains is 2N, since each subdomain is extended by N layers of elements in a symmetric way.
- `ffddminterfacelabel` the label of the new border of the subdomain meshes (the interface between the subdomains) (default 10). Used for imposing problem-dependent boundary conditions at the interface between subdomains for the preconditioner, for example optimized Robin boundary conditions (see ORAS).
- `ffddmpCS` initialized by command-line argument **-ffddm_master_p N**, number of mpi processes used for the assembly and resolution of the coarse problem for two level preconditioners (default 1).
- `ffddmexclude` initialized by command-line argument **-ffddm_master_exclude**, 0 or 1 (default 0). If true, mpi ranks participating in the assembly and resolution of the coarse problem for two level preconditioners will be excluded from the spatial domain decomposition and will only work on the coarse problem.
- `ffddmsplit` initialized by command-line argument **ffddm_split N**, level of refinement of the local submeshes with respect to the initial global mesh (default 1). This is useful for large problems, where we want to avoid working with a very large global mesh. The idea is to start from a coarser global mesh, and generate finer local meshes in parallel during the mesh decomposition step in order to reach the desired level of refinement for the subdomains. For example, calling *ffddmbuildDmesh* with *ffddmsplit* = 3 will generate local submeshes where each mesh element of the initial mesh is split into 3^d elements.
- `ffddmprecond` initialized by command-line argument **-ffddm_schwarz_method S**, specifies the type of one level preconditioner M_1^{-1} to build when calling *ffddmsetupPrecond*: "asm" (*Additive Schwarz*), "ras" (*Restricted Additive Schwarz*), "oras" (*Optimized Restricted Additive Schwarz*), "soras" (*Symmetric Optimized Restricted Additive Schwarz*) or "none" (no preconditioner). Default is "ras". See *ffddmsetupPrecond* for more details.
- `ffddmnu` initialized by command-line argument **-ffddm_geneo_nu N**, number of local eigenvectors to compute in each subdomain when solving the local generalized eigenvalue problem for the GenEO method (default 20). See *ffddmgeneosetup* for more details.
- `ffddmtau` initialized by command-line argument **-ffddm_geneo_threshold R**, threshold parameter for selecting local eigenvectors when solving the local generalized eigenvalue problems for the GenEO method (default 0.5). If the command-line parameter **-ffddm_geneo_nu N** is used, then *ffddmtau* is initialized to 0. See *ffddmgeneosetup* for more details.

- `ffddmcorrection` initialized by command-line argument **-ffddm_schwarz_coarse_correction S**, specifies the coarse correction formula to use for the two level preconditioner: "AD" (*Additive*), "BNN" (*Balancing Neumann-Neumann*), "ADEF1" (*Adapted Deflation Variant 1*), "ADEF2" (*Adapted Deflation Variant 2*), "RBNN1" (*Reduced Balancing Variant 1*), "RBNN2" (*Reduced Balancing Variant 2*) or "none" (no coarse correction). Default is "ADEF1". See the section about *Two level preconditioners* for more details.

3.9.4 Tutorial

Authors: Pierre-Henri Tournier - Frédéric Nataf - Pierre Jolivet

What is ffddm ?

- **ffddm** implements a class of parallel solvers in *FreeFEM*: *overlapping Schwarz domain decomposition methods*
- **The entire ffddm framework is written in the *FreeFEM* language ffddm aims at simplifying the use of parallel solvers in *F***
  You can find the **ffddm** scripts here ('ffddm*.idp' files) and examples here
- **ffddm provides a set of high-level macros and functions to**
  - handle data distribution: distributed meshes and linear algebra
  - build DD preconditioners for your variational problems
  - solve your problem using preconditioned Krylov methods
- **ffddm** implements scalable two level Schwarz methods, with a coarse space correction built either from a coarse mesh or a GenEO coarse space *Ongoing research*: approximate coarse solves and three level methods
- **ffddm can also act as a wrapper for the HPDDM library. HPDDM** is an efficient C++11 implementation of various domain decomposition methods and Krylov subspace algorithms with advanced block and recycling techniques More details on how to use **HPDDM** within **ffddm** *here*

Why Domain Decomposition Methods ?

How can we solve a large sparse linear system $Au = b \in \mathbb{R}^n$?

Step 1: Decompose the mesh

See *documentation*

Build a collection of N overlapping sub-meshes $(Th_i)_{i=1}^N$ from the global mesh Th

```
ffddmbuildDmesh( prmesh , ThGlobal , comm )
```

- mesh distributed over the MPI processes of communicator **comm**
- initial mesh **ThGlobal** partitioned with *metis* by default
- size of the overlap given by *ffddmoverlap* (default 1)

prmesh#Thi is the local mesh of the subdomain for each mpi process

```
macro dimension 2// EOM            // 2D or 3D

include "ffddm.idp"

mesh ThGlobal = square(100,100);     // global mesh

// Step 1: Decompose the mesh
ffddmbuildDmesh( M , ThGlobal , mpiCommWorld )

medit("Th"+mpirank, MThi);
```

Copy and paste this to a file 'test.edp' and run it:

```
ff-mpirun -np 2 test.edp -glut ffglut
```

Step 2: Define your finite element

See *documentation*

```
ffddmbuildDfespace( prfe , prmesh , scalar , def , init , Pk )
```

builds the local finite element spaces and associated distributed operators on top of the mesh decomposition **prmesh**

- **scalar**: type of data for this finite element: *real* or *complex*
- **Pk**: your type of finite element: P1, [P2,P2,P1], . . .

- **def**, **init**: macros specifying how to define and initialize a **Pk** FE function

prfe#Vhi is the local FE space defined on *prmesh#Thi* for each mpi process

Example for P2 *complex*:

```
macro def(u)  u // EOM
macro init(u) u // EOM
ffddmbuildDfespace( FE, M, complex,
                    def, init, P2 )
```

Example for [P2,P2,P1] *real*:

```
macro def(u)  [u, u#B, u#C] // EOM
macro init(u) [u, u, u]     // EOM
ffddmbuildDfespace( FE, M, real, def,
                    init, [P2,P2,P1] )
```

Distributed vectors and restriction operators

Natural decomposition of the set of d.o.f.'s $\mathcal{N}$ of Vh into the N subsets of d.o.f.'s $(\mathcal{N}_i)_{i=1}^N$ each associated with the local FE space Vh_i

$$\mathcal{N} = \cup_{i=1}^N \mathcal{N}_i \,,$$

but with duplications of the d.o.f.'s in the overlap

Definition a *distributed vector* is a collection of local vectors $(\mathbf{V}_i)_{1 \le i \le N}$ so that the values on the duplicated d.o.f.'s are the same:

$$\mathbf{V}_i = R_i \, \mathbf{V}, \quad i = 1, ..., N$$

where $\mathbf{V}$ is the corresponding global vector and R_i is the *restriction operator* from $\mathcal{N}$ into $\mathcal{N}_i$

Remark R_i^T is the *extension operator*: extension by 0 from $\mathcal{N}_i$ into $\mathcal{N}$

Partition of unity

Duplicated unknowns coupled via a *partition of unity*:

$$I = \sum_{i=1}^{N} R_i^T D_i R_i$$

$(D_i)_{1 \leq i \leq N}$ are square diagonal matrices of size $\#\mathcal{N}_i$

$$\mathbf{V} = \sum_{i=1}^{N} R_i^T D_i R_i \mathbf{V} = \sum_{i=1}^{N} R_i^T D_i \mathbf{V}_i$$

Data exchange between neighbors

```
func prfe#update(K[int] vi, bool scale)
```

synchronizes local vectors $\mathbf{V}_i$ between subdomains $\Rightarrow$ exchange the values of $mathbfV_i$ shared with neighbors in the overlap region

$$\mathbf{V}_i \leftarrow R_i \left(\sum_{j=1}^{N} R_j^T D_j \mathbf{V}_j \right) = D_i \mathbf{V}_i + \sum_{j \in \mathcal{O}(i)} R_i \, R_j^T \, D_j \mathbf{V}_j$$

where $\mathcal{O}(i)$ is the set of neighbors of subdomain i. Exchange operators $R_i \, R_j^T$ correspond to neighbor-to-neighbor MPI communications

```
FEupdate(vi, false);
```

$$\mathbf{V}_i \leftarrow R_i \left(\sum_{j=1}^{N} R_j^T \mathbf{V}_j \right)$$

```
FEupdate(vi, true);
```

$$\mathbf{V}_i \leftarrow R_i \left(\sum_{j=1}^{N} R_j^T D_j \mathbf{V}_j \right)$$

```
macro dimension 2// EOM            // 2D or 3D

include "ffddm.idp"

mesh ThGlobal = square(100,100);     // global mesh

// Step 1: Decompose the mesh
ffddmbuildDmesh( M , ThGlobal , mpiCommWorld )

// Step 2: Define your finite element
macro def(u)  u // EOM
macro init(u) u // EOM
ffddmbuildDfespace( FE , M , real , def , init , P2 )

FEVhi vi = x;
medit("v"+mpirank, MThi, vi);

vi[] = FEDk[mpirank];
medit("D"+mpirank, MThi, vi);

vi = 1;
FEupdate(vi[],true);
ffddmplot(FE,vi,"1")

FEupdate(vi[],false);
ffddmplot(FE,vi,"multiplicity")
```

Step 3: Define your problem

See *documentation*

```
ffddmsetupOperator( pr , prfe , Varf )
```

builds the distributed operator associated to your variational form on top of the distributed FE **prfe**

Varf is a macro defining your abstract variational form

```
macro Varf(varfName, meshName, VhName)
    varf varfName(u,v) = int2d(meshName)(grad(u)'* grad(v))
                       + int2d(meshName)(f*v) + on(1, u = 0);  // EOM
```

$\Rightarrow$ assemble local 'Dirichlet' matrices $A_i = R_i A R_i^T$

$$A = \sum_{i=1}^{N} R_i^T D_i A_i R_i$$

> **Warning:** only true because $D_i R_i A = D_i A_i R_i$ due to the fact that D_i vanishes at the interface **!!**

pr#A applies A to a distributed vector: $\mathbf{U}_i \leftarrow R_i \sum_{j=1}^{N} R_j^T D_j A_j \mathbf{V}_j$

$\Rightarrow$ multiply by A_i + *prfe#update*

```
macro dimension 2// EOM            // 2D or 3D

include "ffddm.idp"

mesh ThGlobal = square(100,100);     // global mesh

// Step 1: Decompose the mesh
ffddmbuildDmesh( M , ThGlobal , mpiCommWorld )

// Step 2: Define your finite element
macro def(u)  u // EOM
macro init(u) u // EOM
ffddmbuildDfespace( FE , M , real , def , init , P2 )

// Step 3: Define your problem
macro grad(u) [dx(u), dy(u)] // EOM
macro Varf(varfName, meshName, VhName)
    varf varfName(u,v) = int2d(meshName)(grad(u)'* grad(v))
                       + int2d(meshName)(1*v) + on(1, u = 0);  // EOM
ffddmsetupOperator( PB , FE , Varf )

FEVhi ui, bi;
ffddmbuildrhs( PB , Varf , bi[] )

ui[] = PBA(bi[]);
ffddmplot(PB, ui, "A*b")
```

Summary so far: translating your sequential *FreeFEM* script

Step 1: Decompose the mesh

See *documentation*

```
mesh Th = square(100,100);
```

```
mesh Th = square(100,100);
ffddmbuildDmesh(M, Th, mpiCommWorld)
```

Step 2: Define your finite element

See *documentation*

```
fespace Vh(Th, P1);
```

```
macro def(u)  u // EOM
macro init(u) u // EOM
ffddmbuildDfespace(FE, M, real, def, init, P1)
```

Step 3: Define your problem

See *documentation*

```
varf Pb(u, v) = ...
matrix A = Pb(Vh, Vh);
```

```
macro Varf(varfName, meshName, VhName)
    varf varfName(u,v) = ... // EOM
ffddmsetupOperator(PB, FE, Varf)
```

Solve the linear system

See *documentation*

```
u[] = A^-1 * b[];
```

```
ui[] = PBdirectsolve(bi[]);
```

Solve the linear system with the parallel direct solver *MUMPS*

See *documentation*

```
func K[int] pr#directsolve(K[int]& bi)
```

We have A and b in distributed form, we can solve the linear system $Au = b$ using the parallel direct solver *MUMPS*

```
// Solve the problem using the direct parallel solver MUMPS
ui[] = PBdirectsolve(bi[]);
ffddmplot(PB, ui, "u")
```

Step 4: Define the one level DD preconditioner

See *documentation*

```
ffddmsetupPrecond( pr , VarfPrec )
```

builds the one level preconditioner for problem **pr**.

By default it is the *Restricted Additive Schwarz (RAS)* preconditioner:

$$M_1^{-1} = M_{\text{RAS}}^{-1} = \sum_{i=1}^{N} R_i^T D_i A_i^{-1} R_i \quad \text{with } A_i = R_i A R_i^T$$

Setup step: compute the LU (or LDL^T) factorization of local matrices A_i

pr#PREC1 applies M_1^{-1} to a distributed vector: $\mathbf{U}_i \leftarrow R_i \sum_{j=1}^{N} R_j^T D_j A_j^{-1} \mathbf{V}_i$

$\Rightarrow$ apply A_i^{-1} (forward/backward substitutions) + *prfe#update*

Step 5: Solve the linear system with preconditioned GMRES

See *documentation*

```
func K[int] pr#fGMRES(K[int]& x0i, K[int]& bi, real eps, int itmax, string sp)
```

solves the linear system with flexible GMRES with DD preconditioner M^{-1}

- **x0i**: initial guess
- **bi**: right-hand side
- **eps**: relative tolerance
- **itmax**: maximum number of iterations
- **sp**: *"left"* or *"right"* preconditioning

left preconditioning

solve $M^{-1}Ax = M^{-1}b$

right preconditioning

solve $AM^{-1}y = b$

$\Rightarrow x = M^{-1}y$

```
macro dimension 2// EOM            // 2D or 3D
include "ffddm.idp"

mesh ThGlobal = square(100,100);    // global mesh
// Step 1: Decompose the mesh
ffddmbuildDmesh( M , ThGlobal , mpiCommWorld )
// Step 2: Define your finite element
macro def(u)  u // EOM
macro init(u) u // EOM
ffddmbuildDfespace( FE , M , real , def , init , P2 )
// Step 3: Define your problem
macro grad(u) [dx(u), dy(u)] // EOM
macro Varf(varfName, meshName, VhName)
    varf varfName(u,v) = int2d(meshName)(grad(u)'* grad(v))
                       + int2d(meshName)(1*v) + on(1, u = 0);  // EOM
ffddmsetupOperator( PB , FE , Varf )

FEVhi ui, bi;
ffddmbuildrhs( PB , Varf , bi[] )

// Step 4: Define the one level DD preconditioner
ffddmsetupPrecond( PB , Varf )

// Step 5: Solve the linear system with GMRES
FEVhi x0i = 0;
ui[] = PBfGMRES(x0i[], bi[], 1.e-6, 200, "right");

ffddmplot(PB, ui, "u")
PBwritesummary
```

Define a two level DD preconditioner

See *documentation*

Goal improve scalability of the one level method

$\Rightarrow$ enrich the one level preconditioner with a *coarse problem* coupling all subdomains

Main ingredient is a rectangular matrix Z of size $n \times n_c$, where $n_c \ll n$ Z is the *coarse space* matrix

The *coarse space operator* $E = Z^T AZ$ is a square matrix of size $n_c \times n_c$

The simplest way to enrich the one level preconditioner is through the *additive coarse correction* formula:

$$M_2^{-1} = M_1^{-1} + ZE^{-1}Z^T$$

How to choose $color{red}{Z}$?

Build the GenEO coarse space

See *documentation*

```
ffddmgeneosetup( pr , Varf )
```

The *GenEO* method builds a robust coarse space for highly heterogeneous or anisotropic **SPD** problems

$\Rightarrow$ solve a local generalized eigenvalue problem in each subdomain

$$D_i A_i D_i \, V_{i,k} = \lambda_{i,k} \, A_i^{\text{Neu}} \, V_{i,k}$$

with A_i^{Neu} the local Neumann matrices built from **Varf** (same **Varf** as *Step 3*

The GenEO coarse space is $Z = (R_i^T D_i V_{i,k})_{\lambda_{i,k} \geq \tau}^{i=1,\dots,N}$ The eigenvectors $V_{i,k}$ selected to enter the coarse space correspond to eigenvalues $\lambda_{i,k} \geq \tau$, where τ is a threshold parameter

> **Theorem** the spectrum of the preconditioned operator lies in the interval $[\frac{1}{1+k_1\tau}, k_0]$ where $k_0 - 1$ is the # of neighbors and k_1 is the multiplicity of intersections $\Rightarrow$ k_0 and k_1 do not depend on N nor on the PDE

```
macro dimension 2// EOM            // 2D or 3D
include "ffddm.idp"

mesh ThGlobal = square(100,100);    // global mesh
// Step 1: Decompose the mesh
ffddmbuildDmesh( M , ThGlobal , mpiCommWorld )
// Step 2: Define your finite element
macro def(u)  u // EOM
macro init(u) u // EOM
ffddmbuildDfespace( FE , M , real , def , init , P2 )
// Step 3: Define your problem
macro grad(u) [dx(u), dy(u)] // EOM
macro Varf(varfName, meshName, VhName)
    varf varfName(u,v) = int2d(meshName)(grad(u)'* grad(v))
                       + int2d(meshName)(1*v) + on(1, u = 0);  // EOM
ffddmsetupOperator( PB , FE , Varf )

FEVhi ui, bi;
ffddmbuildrhs( PB , Varf , bi[] )

// Step 4: Define the one level DD preconditioner
ffddmsetupPrecond( PB , Varf )

// Build the GenEO coarse space
ffddmgeneosetup( PB , Varf )

// Step 5: Solve the linear system with GMRES
FEVhi x0i = 0;
ui[] = PBfGMRES(x0i[], bi[], 1.e-6, 200, "right");
```

Build the coarse space from a coarse mesh

See *documentation*

```
ffddmcoarsemeshsetup( pr , Thc , VarfEprec , VarfAprec )
```

For **non SPD** problems, an alternative is to build the coarse space by discretizing the PDE on a coarser mesh **Thc**

Z will be the *interpolation matrix* from the coarse FE space Vh_c to the original FE space Vh

$\Rightarrow E = Z^T AZ$ is the matrix of the problem discretized on the coarse mesh

The variational problem to be discretized on **Thc** is given by macro **VarfEprec**

VarfEprec can differ from the original **Varf** of the problem

Example: added absorption for wave propagation problems

Similarly, **VarfAprec** specifies the global operator involved in multiplicative coarse correction formulas. It defaults to A if **VarfAprec** is not defined

```
macro dimension 2// EOM            // 2D or 3D
include "ffddm.idp"

mesh ThGlobal = square(100,100);    // global mesh
// Step 1: Decompose the mesh
ffddmbuildDmesh( M , ThGlobal , mpiCommWorld )
// Step 2: Define your finite element
macro def(u)  u // EOM
macro init(u) u // EOM
ffddmbuildDfespace( FE , M , real , def , init , P2 )
// Step 3: Define your problem
macro grad(u) [dx(u), dy(u)] // EOM
macro Varf(varfName, meshName, VhName)
    varf varfName(u,v) = int2d(meshName)(grad(u)'* grad(v))
                       + int2d(meshName)(1*v) + on(1, u = 0);  // EOM
ffddmsetupOperator( PB , FE , Varf )

FEVhi ui, bi;
ffddmbuildrhs( PB , Varf , bi[] )

// Step 4: Define the one level DD preconditioner
ffddmsetupPrecond( PB , Varf )

// Build the coarse space from a coarse mesh
mesh Thc = square(10,10);
ffddmcoarsemeshsetup( PB , Thc , Varf , null )

// Step 5: Solve the linear system with GMRES
FEVhi x0i = 0;
ui[] = PBfGMRES(x0i[], bi[], 1.e-6, 200, "right");
```

Use HPDDM within ffddm

See *documentation*

ffddm allows you to use **HPDDM** to solve your problem, effectively replacing the **ffddm** implementation of all parallel linear algebra computations

$\Rightarrow$ define your problem with **ffddm**, solve it with **HPDDM**

⇒ **ffddm** acts as a finite element interface for **HPDDM**

- you can use **HPDDM** features unavailable in **ffddm** such as advanced Krylov subspace methods implementing block and recycling techniques
- conversely, some features of **ffddm** such as two level methods built from a coarse mesh are not implemented in **HPDDM**

To switch to **HPDDM**, simply define the macro *pr#withhpddm* before using *ffddmsetupOperator* (*Step 3*). You can then pass **HPDDM** options with command-line arguments or directly to the underlying **HPDDM** operator:

```
macro PBwithhpddm()1 // EOM
ffddmsetupOperator( PB , FE , Varf )
set(PBhpddmOP,sparams="-hpddm_krylov_method gcrodr");
```

Or, define *pr#withhpddmkrylov* to use **HPDDM** only for the Krylov method

Example here: Helmholtz problem with multiple rhs solved with Block GMRES

Some results: Heterogeneous 3D elasticity with GenEO

Heterogeneous 3D linear elasticity equation discretized with P2 FE solved on 4096 MPI processes $n \approx 262$ million

Some results: 2-level DD for Maxwell equations, scattering from the COBRA cavity

f = 10 GHz

f = 16 GHz

Some results: 2-level DD for Maxwell equations, scattering from the COBRA cavity

- order 2 Nedelec edge FE
- fine mesh: 10 points per wavelength
- coarse mesh: 3.33 points per wavelength
- two level ORAS preconditioner with added absorption
- f = 10 GHz: $n \approx 107$ million, $n_c \approx 4$ million

  f = 16 GHz: $n \approx 198$ million, $n_c \approx 7.4$ million

$\rightarrow$ coarse problem too large for a direct solver $\Rightarrow$ inexact coarse solve: GMRES + one level ORAS preconditioner

			Total #	Total times (seconds)			
f	N	# it	inner it	Total	Setup	GMRES	inner
10GHz	1536	32	1527	515.8	383.2	132.6	61.8
10GHz	3072	33	2083	285.0	201.6	83.4	40.6
16GHz	3072	43	3610	549.2	336.8	212.4	118.6
16GHz	6144	46	4744	363.0	210.5	152.5	96.8

speedup of 1.81 from 1536 to 3072 cores at 10GHz

1.51 from 3072 to 6144 cores at 16GHz

You can find the script here

3.9.5 Examples

File name	M_1^{-1}	M_2^{-1}	inexact CS	comments
diffusion-3d-minimal-direct.edp				direct solver MUMPS
diffusion-3d-minimal-ddm.edp	RAS	GenEO		
diffusion-3d-simple.edp	RAS	GenEO		comparison with direct solver
diffusion-2d-thirdlevelgeneo.edp	RAS	GenEO	RAS + GenEO	
elasticity-3d-simple.edp	RAS	GenEO		
elasticity-3d-thirdlevelgeneo.edp	RAS	GenEO	RAS + GenEO	
Helmholtz-2d-simple.edp	ORAS	Coarse Mesh		
Helmholtz-2d-marmousi.edp	ORAS	Coarse Mesh		
Helmholtz-3d-simple.edp	ORAS	Coarse Mesh		
Helmholtz-3d-overthrust.edp	ORAS			
Helmholtz-2d-HPDDM-BGMRES.edp	ORAS			multi-rhs Block GMRES with HPDDM
Navier-2d-marmousi2.edp	ORAS	Coarse Mesh		
Maxwell-3d-simple.edp	ORAS	Coarse Mesh		
Maxwell_Cobracavity.edp	ORAS	Coarse Mesh	ORAS	
natural_convection.edp	ORAS	Coarse Mesh		nonlinear
natural_convection_3D_obstacle.edp	ORAS	Coarse Mesh		nonlinear
Richards-2d.edp	RAS			nonlinear time dependent mesh adaptation

CHAPTER

FOUR

LANGUAGE REFERENCES

In essence **FreeFEM** is a compiler: its language is typed, polymorphic, with exception and reentrant. Every variable must be declared of a certain type, in a declarative statement; each statement are separated from the next by a semicolon `;`.

The language allows the manipulation of basic types integers (`int`), reals (`real`), strings (`string`), arrays (example: `real[int]`), bi-dimensional (2D) finite element meshes (`mesh`), 2D finite element spaces (`fespace`), analytical functions (`func`), arrays of finite element functions (`func[basic_type]`), linear and bilinear operators, sparse matrices, vectors , etc. For example:

```
int i, n = 20; //i, n are integer
real[int] xx(n), yy(n); //two array of size n
for (i = 0; i <= 20; i++){ //which can be used in statements such as
    xx[i] = cos(i*pi/10);
    yy[i] = sin(i*pi/10);
}
```

The life of a variable is the current block `{...}`, except the `fespace` variable, and the variables local to a block are destroyed at the end of the block as follows.

Tip: Example

```
real r = 0.01;
mesh Th = square(10, 10); //unit square mesh
fespace Vh(Th, P1); //P1 Lagrange finite element space
Vh u = x + exp(y);
func f = z*x + r*log(y);
plot(u, wait=true);
{ // new block
    real r = 2; //not the same r
    fespace Vh(Th, P1); //error because Vh is a global name
}// end of block
//here r back to 0.01
```

The type declarations are mandatory in **FreeFEM**; in the end this feature is an asset because it is easy to make bugs in a language with many implicit types.

The variable name is just an alphanumeric string, the underscore character _ is not allowed, because it will be used as an operator in the future.

4.1 Types

4.1.1 Standard types

int

Integer value (equivalent to `long` in `C++`).

```
int i = 0;
```

bool

Boolean value.

```
bool b = true;
```

Tip: The result of a comparison is a boolean

`bool b = (1 < 2);`

real

Real value (equivalent to `double` in `C++`).

```
real r = 0.;
```

complex

Complex value (equivalent to two `double` or `complex<double>` in `C++`).

```
complex c = 0. + 1i;
```

The imaginary number i is defined as `1i`

Tip: Example

```
complex a = 1i, b = 2 + 3i;
cout << "a + b = " << a + b << endl;
cout << "a - b = " << a - b << endl;
cout << "a*b = " << a*b << endl;
cout << "a/b = " << a/b << endl;
```

The output of this script is:

```
a + b = (2,4)
a - b = (-2,-2)
a*b = (-3,2)
a/b = (0.230769,0.153846)
```

Note: See *Complex example* for a detailed example.

string

String value.

```
string s = "this is a string";
```

Note: `string` value is enclosed within double quotes.

Other types can be concatenate to a string, like:

```
int i = 1;
real r = 1.;
string s = "the int i = " + i +", the real r = " + r + ", the complex z = " + (1. +
↪1i);
```

To append a string in a string at position 4:

```
s(4:3) = "+++";
```

To copy a substring in an other string:

```
string s2 = s1(5:10);
```

See *String Example* for a complete example.

4.1.2 Mesh design

border

Border type.

```
border b(t=0., 1.){x=cos(2.*pi*t); y=sin(2.*pi*t);}
```

Define the 2D geometrical border in parametric coordinates.

Note: Label

A label can be defined with the border:

```
border b(t=0., 1.){x=cos(2.*pi*t); y=sin(2.*pi*t); label=1;}
```

Note: Inner variable

An inner variable can be defined inside a border:

```
border b(t=0., 1.){real tt=2.*pi*t; x=cos(tt); y=sin(tt);}
```

Note: From vector

A border can be defined from two vectors using :freefem‘P.x‘ and `P.y`:

```
border b(t=0, vectorX.n-1){x=vectorX[t]; P.x=vectorY[t];}
```

mesh

2D Mesh type (see *Mesh Generation*).

```
mesh Th;
```

mesh3

3D mesh type (see *Mesh Generation*).

```
mesh3 Th;
```

4.1.3 Finite element space design

fespace

Finite element space type (see *Finite Element*).

```
fespace Uh(Th, P1);
fespace UPh(Th, [P2, P2, P1]);
```

A finite element space is based on a mesh (`Th`) with an element definition, scalar (`P1`) or vector (`[P2, P2, P1]`).

Available finite element space:

Generic:

- `P0 / P03d`
- `P0Edge`
- `P1 / P13d`
- `P1dc`
- `P1b / P1b3d`
- `P1bl / P1bl3d`
- `P1nc`
- `P2 / P23d`
- `P2b`
- `P2dc`

- `P2h`
- `RT0 / RT03d`
- `RT0Ortho`
- `Edge03d`

Using *Element_P3*:

- `P3`

Using *Element_P3dc*:

- `P3dc`

Using *Element_P4*:

- `P4`

Using *Element_P4dc*:

- `P4dc`

Using *Element_PkEdge*:

- `P1Edge`
- `P2Edge`
- `P3Edge`
- `P4Edge`
- `P5Edge`

Using *Morlay*:

- `P2Morley`

Using *HCT*:

- `HCT`

Using *BernardiRaugel*:

- `P2BR`

Using *Element_Mixte*:

- `RT1`
- `RT1Ortho`
- `RT2`
- `RT2Ortho`
- `BDM1`
- `BDM1Ortho`

Using *Element_Mixte3d*:

- `Edge13d`
- `Edge23d`

Using *Element_QF*:

- `FEQF`

A finite element function is defined as follow:

```
fespace Uh(Th, P1);
Uh u;

fespace UPh(Th, [P2, P2, P1]);
UPh [Ux, Uy, p];
```

4.1.4 Macro design

macro

Macro type.

```
macro vU() [Ux, Uy] //
macro grad(u) [dx(u), dy(u)] //
```

Macro ends with `//`.

Note: Macro concatenation

You can use the C concatenation operator `##` inside a macro using `#`.

If *Ux* and *Uy* are defined as finite element function, you can define:

```
macro Grad(U) [grad(U#x), grad(U#y)] // End of macro
```

See *Macro example*

NewMacro / EndMacro

Warning: In developement - Not tested

Set and end a macro

```
NewMacro grad(u) [dx(u), dy(u)] EndMacro
```

IFMACRO

Check if a macro exists and check its value.

```
IFMACRO(AA) //check if macro AA exists
...
ENDIFMACRO

IFMACRO(AA, tt) //check if amcro exists and is equall to tt
...
ENDIFMACRO
```

ENDIFMACRO

4.1.5 Functions design

func

Function type.

Function without parameters (x, y and z are implicitly considered):

```
func f = x^2 + y^2;
```

Note: Function's type is defined by the expression's type.

Function with parameters:

```
func real f (real var){
    return x^2 + y^2 + var^2;
}
```

Elementary functions

Class of basic functions (polynomials, exponential, logarithmic, trigonometric, circular) and the functions obtained from those by the four arithmetic operations

$$f(x) + g(x),\ f(x) - g(x),\ f(x)g(x),\ f(x)/g(x)$$

and by composition $f(g(x))$, each applied a finite number of times.

In **FreeFEM**, all elementary functions can thus be created. The derivative of an elementary function is also an elementary function; however, the indefinite integral of an elementary function cannot always be expressed in terms of elementary functions.

See *Elementary function example* for a complete example.

Random functions

FreeFEM includes the Mersenne Twister random number generator. It is a very fast and accurate random number generator of period $2^{219937} - 1$.

See *randint32()*, *randint31()*, *randreal1()*, *randreal2()*, *randreal3()*, *randres53()*, *randinit(seed)*.

In addition, the `ffrandom` plugin interface `random`, `srandom` and `srandomdev` functions of the Unix `libc` library. The range is $0 - -2^{31} - 1$.

Note: If `srandomdev` is not defined, a seed based on the current time is used.

`gsl` plugin equally allows usage of all random functions of the `gsllib`, see *gsl external library*.

FE-functions

Finite element functions are also constructed like elementary functions by an arithmetic formula involving elementary functions.

The difference is that they are evaluated at declaration time and **FreeFEM** stores the array of its values at the places associated with he degree of freedom of the finite element type. By opposition, elementary functions are evaluated only when needed. Hence FE-functions are not defined only by their formula but also by the mesh and the finite element which enter in their definitions.

If the value of a FE-function is requested at a point which is not a degree of freedom, an interpolation is used, leading to an interpolation error, while by contrast, an elementary function can be evaluated at any point exactly.

```
func f = x^2*(1+y)^3 + y^2;
mesh Th = square(20, 20, [-2+4*x, -2+4*y]); // ]-2, 2[^2
fespace Vh(Th, P1);
Vh fh=f; //fh is the projection of f to Vh (real value)
func zf = (x^2*(1+y)^3 + y^2)*exp(x + 1i*y);
Vh<complex> zh = zf; //zh is the projection of zf to complex value Vh space
```

The construction of `fh = f` is explained in *Finite Element*.

Warning: The `plot` command only works for real or complex FE-functions, not for elementary functions.

4.1.6 Problem design

problem

Problem type.

```
problem Laplacian (u, uh) = ...
```

FreeFEM needs the variational form in the problem definition.

In order to solve the problem, just call:

```
Laplacian;
```

Note: Solver

A solver can be specified in the problem definition:

```
problem Laplacian(u, uh, solver=CG) = ...
```

The default solver is `sparsesolver` or `LU` if any direct sparse solver is available.

Solvers are listed in the *Global variables* section.

Note: Stop test

A criterion ε can be defined for iterative methods, like CG for example:

```
problem Laplacian(u, uh, solver=CG, eps=1.e-6) = ...
```

If $\varepsilon > 0$, the stop test is:

$$||Ax - b|| < \varepsilon$$

Else, the stop test is:

$$||Ax - b|| < \frac{|\varepsilon|}{||Ax_0 - b||}$$

Note: Reconstruction

The keyword `init` controls the reconstruction of the internal problem matrix.

If `init` is set to `false` or `0`, the matrix is reconstructed et each problem calls (or after a mesh modification), else the previously constructed matrix is used.

```
problem Laplacian(u, uh, init=1) = ...
```

Note: Preconditioning

A preconditioner can be specified in the problem definition:

```
problem Laplacian(u, uh, precon=P) = ...
```

The preconditioning function must have a prototype like:

```
func real[int] P(real[int] &xx);
```

Note: "*Très grande valeur*"

The "*Très grand valeur*" `tgv` (or *Terrible giant value*) used to implement the Dirichlet conditions can be modified in the problem definition:

```
problem Laplacian(u, uh, tgv=1e30) = ...
```

Refere to *Problem definition* for a description of the Dirichlet condition implementation.

Note: Pivot tolerance

The tolerance of the pivot in `UMFPACK`, `LU`, `Crout`, `Cholesky` factorization can be modified in the problem definition:

```
problem Laplacian(u, uh, solver=LU, tolpivot=1e-20) = ...
```

Note: `UMFPACK`

Two specific parameters for the `UMFPACK` can be modifed:

- Tolerance of the pivot sym
- strategy

```
problem Laplacian(u, uh, solver=LU, tolpivotsym=1e-1, strategy=0) = ...
```

Refer to the UMFPACK website for more informations.

Note: `dimKrylov`

Dimension of the Krylov space

Usage of `problem` is detailled in the *tutorials*.

solve

Solve type.

Identical to *problem* but automatically solved.

Usage of `solve` is detailled in the *tutorials*.

varf

Variational form type.

```
varf vLaplacian (u, uh) = ...
```

Directly define a variational form.

This is the other way to define a problem in order to directly manage matrix and right hang side.

Usage of `varf` is detailed in the *tutorial*.

4.1.7 Array

An array stores multiple objects, and there are 2 kinds of arrays:

- the first is similar to vector, i.e. array with integer indices
- the second is array with string indices

In the first case, the size of the array must be known at execution time, and implementation is done with the `KN<>` class and all the vector operator of `KN<>` are implemented.

Arrays can be set like in Matlab or Scilab with the operator `::`, the array generator of `a:c` is equivalent to `a:1:c`, and the array set by `a:b:c` is set to size $\lfloor |(b-a)/c| + 1 \rfloor$ and the value i is set by $a + i(b-a)/c$.

There are `int`, `real`, `complex` array with, in the third case, two operators (`.im`, `.re`) to generate the real and imaginary real array from the complex array (without copy).

Note: Quantiles are points taken at regular intervals from the cumulative distribution function of a random variable. Here the array values are random.

This statistical function `a.quantile(q)` computes v from an array a of size n for a given number $q \in]0,1[$ such that:

$$\#\{i/a[i] < v\} \sim q * n$$

it is equivalent to $v = a[q * n]$ when the array a is sorted.

For example, to declare, fill and display an array of `real` of size `n`:

```
int n = 5;
real[int] Ai(n);
for (int i = 0; i < n; i++)
    Ai[i] = i;
cout << Ai << endl;
```

The output of this script is:

```
5
      0   1   2   3   4
```

See the *Array example* for a complete example.

Array index

Array index can be int or string:

```
real[int] Ai = [1, 1, 0, 0];
real[string] As = [1, 1, 0, 0];
```

Array size

The size of an array is obtained using the keyword `n`:

```
int ArraySize = Ai.n;
```

Array sort

To sort an array:

```
Ai.sort;
```

Double array

A double array (matrix) can be defined using two indexes:

```
real[int, int] Aii = [[1, 1], [0, 0]];
```

The two sizes are obtained using the keywords `n` and `m`:

```
int ArraySize1 = Aii.n;
int ArraySize2 = Aii.m;
```

The minimum and maximum values of an array (simple or double) can be obtained using:

```
real ArrayMin = Aii.min;
real ArrayMax = Aii.max;
```

Th minimum and maximum position of an array can be obtained using:

```
int mini = Aii.imin;
int minj = Aii.jmin;

int maxi = Aii.imax;
int maxj = Aii.jmax;
```

Tip: An array can be obtained from a finite element function using:

```
real[int] aU = U[];
```

where `U` is a finite element function.

Array of FE functions

It is also possible to make an array of FE functions, with the same syntax, and we can treat them as vector valued function if we need them.

The syntax for space or vector finite function is

```
int n = 100; //size of the array.
Vh[int] wh(n); //real scalar case
Wh[int] [uh,vh](n); //real vectorial case
Vh<complex>[int] cwh(n); //complex scalar case
Wh<complex>[int] [cuh, cvh](n); //complex vectorial case
[cuh[2], cvh[2]] = [x, y]; //set interpolation of index 2

// Array of Array
real [int][int] V(10);
matrix[int] B(10);
real [int, int][int] A(10);
```

Tip: Example

In the following example, Poisson's equation is solved for 3 different given functions $f = 1$, $\sin(\pi x)\cos(\pi y)$, $|x-1||y-1|$, whose solutions are stored in an array of FE function.

```
// Mesh
mesh Th = square(20, 20, [2*x, 2*y]);

// Fespace
fespace Vh(Th, P1);
Vh u, v, f;

// Problem
problem Poisson (u, v)
   = int2d(Th)(
        dx(u)*dx(v)
      + dy(u)*dy(v)
   )
   + int2d(Th)(
      - f*v
   )
```

(continues on next page)

(continued from previous page)

```
   + on(1, 2, 3, 4, u=0)
   ;

Vh[int] uu(3); //an array of FE function
// Solve problem 1
f = 1;
Poisson;
uu[0] = u;
// Solve problem 2
f = sin(pi*x)*cos(pi*y);
Poisson;
uu[1] = u;
// Solve problem 3
f = abs(x-1)*abs(y-1);
Poisson;
uu[2] = u;

// Plot
for (int i = 0; i < 3; i++)
   plot(uu[i], wait=true);
```

See *FE array example*.

Map arrays

```
real[string] map; //a dynamic array

map["1"] = 2.0;
map[2] = 3.0; //2 is automatically cast to the string "2"

cout << "map[\"1\"] = " << map["1"] << endl;
cout << "map[2] = " << map[2] << endl;
```

It is just a map of the standard template library so no operations on vector are allowed, except the selection of an item.

4.1.8 matrix

Defines a sparse matrix.

Matrices can be defined like vectors:

```
matrix A = [[1, 2, 3],
            [4, 5, 6],
            [7, 8, 9]];
```

or using a variational form type (see *Finite Element*):

```
matrix Laplacian = vLaplacian(Uh, Uh);
```

or from block of matrices:

```
matrix A1, ..., An;
matrix A = [[A1, ...], ..., [..., An]];
```

or using sparse matrix set:

```
A = [I, J, C];
```

Note: `I` and `J` are `int[int]` and `C` is `real[int]`. The matrix is defined as:

$$A = \sum_k C[k] M_{I[k],J[k]}$$

where $M_{a,b} = (\delta_{ia}\delta_{jb})_{ij}$

I, *J* and *C* can be retrieved using *[I, J, C] = A* (arrays are automatically resized).

The size of the matrix is `n = I.max;`, `m = J.max;`.

Matrices are designed using templates, so they can be real or complex:

```
matrix<real> A = ...
matrix<complex> Ai = ...
```

Note: Solver

See *problem*.

The default solver is *GMRES*.

```
matrix A = vLaplacian(Uh, Uh, solver=sparsesolver);
```

or

```
set(A , solver=sparsesolver);
```

Note: Factorize

If `true`, the factorization is done for `LU`, `Cholesky` or `Crout`.

```
matrix A = vLaplacian(Uh, Uh, solver=LU, factorize=1);
```

or

```
set(A , solver=LU, factorize=1);
```

Note: Stop test

See *problem*.

Note: *Très grande valeur*

See *problem*.

Note: Preconditioning

See *problem*.

Note: Pivot tolerance

See *problem*.

Note: `UMFPACK`

See *problem*.

Note: `dimKrylov`

See *problem*.

Note: datafilename

Name of the file containing solver parameters, see *Parallel sparse solvers*

Note: `lparams`

Vector of integer parameters for the solver, see *Parallel sparse solvers*

Note: `dparams`

Vector of real parameters for the solver, see *Parallel sparse solvers*

Note: `sparams`

String parameters for the solver, see *Parallel sparse solvers*

Tip: To modify the `solver`, the stop test,... after the matrix construction, use the *set keyword*.

Matrix size

The size of a matrix is obtain using:

```
int NRows = A.n;
int NColumns = A.m;
```

Matrix resize

To resize a matrix, use:

```
A.resize(n, m);
```

Warning: When resizing, all new terms are set to zero.

Matrix diagonal

The diagonal of the matrix is obtained using:

```
real[int] Aii = A.diag;
```

Matrix renumbering

```
int[int] I(15, J(15);
matrix B = A;
B = A(I, J);
B = A(I^-1, J^-1);
```

Complex matrix

Use `.im` and `.re` to get the imaginary and real part of a complex matrix, respectively:

```
matrix<complex> C = ...
matrix R = C.re;
matrix I = C.im;
```

Dot product / Outer product

The dot product of two matrices is realized using:

```
real d = A' * B;
```

The outer product of two matrices is realized using:

```
matrix C = A * B'
```

See *Matrix operations example* for a complete example.

Matrix inversion

See *Matrix inversion example*.

4.2 Global variables

4.2.1 area

Area of the current triangle.

```
fespace Vh0(Th, P0);
Vh0 A = area;
```

4.2.2 ARGV

Array that contains all the command line arguments.

```
for (int i = 0; i < ARGV.n; i++)
    cout << ARGV[i] << endl;
```

See *Command line arguments example* for a complete example.

4.2.3 BoundaryEdge

Return 1 if the current edge is on a boundary, 0 otherwise.

```
real B = int2d(Th)(BoundaryEdge);
```

4.2.4 CG

Conjugate gradient solver.

Usable in *problem* and *solve* definition

```
problem Laplacian (U, V, solver=CG) = ...
```

Or in *matrix* construction

```
matrix A = vLaplacian(Uh, Uh, solver=CG);
```

Or in *set function*

```
set(A, solver=CG);
```

4.2.5 Cholesky

Cholesky solver.

4.2.6 Crout

Crout solver.

4.2.7 edgeOrientation

Sign of $i - j$ if the current edge is $[q_i, q_j]$.

```
real S = int1d(Th, 1)(edgeOrientation);
```

4.2.8 false

False boolean value.

```
bool b = false;
```

4.2.9 GMRES

GMRES solver (Generalized minimal residual method).

4.2.10 hTriangle

Size of the current triangle.

```
fespace Vh(Th, P0);
Vh h = hTriangle;
```

4.2.11 include

Include an *external library*.

```
include "iovtk"
```

4.2.12 InternalEdge

Return 0 if the current edge is on a boundary, 1 otherwise.

```
real I = int2d(Th)(InternalEdge);
```

4.2.13 label

Label number of a boundary if the current point is on a boundary, 0 otherwise.

```
int L = Th(xB, yB).label;
```

4.2.14 lenEdge

Length of the current edge.

For an edge $[q_i, g_j]$, return $|q_i - q_j|$.

```
real L = int1d(Th, 1)(lenEdge);
```

4.2.15 load

Load a script.

```
load "Element_P3"
```

4.2.16 LU

LU solver.

4.2.17 N

Outward unit normal at the current point if it is on a curve defined by a border. `N.x, N.y, N.z` are respectively the x, y and z components of the normal.

```
func Nx = N.x;
func Ny = N.y;
func Nz = N.z;
```

4.2.18 nTonEdge

Number of adjacent triangles of the current edge.

```
real nTE = int2d(Th)(nTonEdge);
```

4.2.19 nuEdge

Index of the current edge in the triangle.

```
real nE = int2d(Th)(nuEdge);
```

4.2.20 nuTriangle

Index of the current triangle.

```
fespace Vh(Th, P0);
Vh n = nuTriangle;
```

4.2.21 P

Current point.

```
real cx = P.x;
real cy = P.y;
real cz = P.z;
```

4.2.22 pi

Pi = 3.14159.

```
real Pi = pi;
```

This is a real value.

4.2.23 region

Region number of the current point. If the point is outside, then `region == notaregion` where `notaregion` is a **FreeFEM** integer constant.

```
int R = Th(xR, yR).region;
```

4.2.24 sparsesolver

Sparse matrix solver.

4.2.25 true

True boolean value.

```
bool b = true;
```

4.2.26 verbosity

Verbosity level.

```
int Verbosity = verbosity;
verbosity = 0;
```

0 = nothing, 1 = little information, 10 = a lot of information, . . .

This is an integer value.

4.2.27 version

FreeFEM version.

```
cout << version << endl;
```

4.2.28 volume

Volume of the current tetrahedra.

```
fespace Vh0(Th, P0);
Vh0 V = volume;
```

4.2.29 x

The x coordinate at the current point.

```
real CurrentX = x;
```

This is a real value.

4.2.30 y

The y coordinate at the current point.

```
real CurrentY = y;
```

This is a real value.

4.2.31 z

The z coordinate at the current point.

```
real CurrentZ = z;
```

This is a real value.

4.3 Quadrature formulae

The quadrature formula is like the following:

$$\int_D f(\mathbf{x}) \approx \sum_{\ell=1}^{L} \omega_\ell f(\boldsymbol{\xi}_\ell)$$

4.3.1 int1d

Quadrature formula on an edge.

Notations

$|D|$ is the measure of the edge D.

For a shake of simplicity, we denote:

$$f(\mathbf{x}) = g(t)$$

with $0 \leq t \leq 1$; $\mathbf{x} = (1-t)\mathbf{x}_0 + t\mathbf{x}_1$.

qf1pE

```
int1d(Th, qfe=qf1pE)( ... )
```

or

```
int1d(Th, qforder=2)( ... )
```

This quadrature formula is exact on $\mathbb{P}_1$.

$$\int_D f(\mathbf{x}) \approx |D| g\left(\frac{1}{2}\right)$$

qf2pE

```
int1d(Th, qfe=qf2pE)( ... )
```

or

```
int1d(Th, qforder=3)( ... )
```

This quadrature formula is exact on $\mathbb{P}_3$.

$$\int_D f(\mathbf{x}) \approx \frac{|D|}{2}\left(g\left(\frac{1+\sqrt{1/3}}{2}\right)+g\left(\frac{1-\sqrt{1/3}}{2}\right)\right)$$

qf3pE

```
int1d(Th, qfe=qf3pE)( ... )
```

or

```
int1d(Th, qforder=6)( ... )
```

This quadrature formula is exact on $\mathbb{P}_5$.

$$\int_D f(\mathbf{x}) \approx \frac{|D|}{18}\left(5g\left(\frac{1+\sqrt{3/5}}{2}\right)+8g\left(\frac{1}{2}\right)+5g\left(\frac{1-\sqrt{3/5}}{2}\right)\right)$$

qf4pE

```
int1d(Th, qfe=qf4pE)( ... )
```

or

```
int1d(Th, qforder=8)( ... )
```

This quadrature formula is exact on $\mathbb{P}_7$.

$$\int_D f(\mathbf{x}) \approx \frac{|D|}{72}\left((18-\sqrt{30})g\left(\frac{1-\frac{\sqrt{525+70\sqrt{30}}}{35}}{2}\right)+(18-\sqrt{30})g\left(\frac{1+\frac{\sqrt{525+70\sqrt{30}}}{35}}{2}\right)+(18+\sqrt{30})g\left(\frac{1-\frac{\sqrt{525-70\sqrt{30}}}{35}}{2}\right)+$$

qf5pE

```
int1d(Th, qfe=qf5pE)( ... )
```

or

```
int1d(Th, qforder=10)( ... )
```

This quadrature formula is exact on $\mathbb{P}_9$.

$$\int_D f(\mathbf{x}) \approx |D| \left(\frac{(332-13\sqrt{70})}{1800} g\left(\frac{1-\frac{\sqrt{245+14\sqrt{70}}}{21}}{2} \right) + \frac{(332-13\sqrt{70})}{1800} g\left(\frac{1+\frac{\sqrt{245+14\sqrt{70}}}{21}}{2} \right) + \frac{64}{225} g\left(\frac{1}{2}\right) + \frac{(332+13\sqrt{70})}{1800}$$

qf1pElump

```
int1d(Th, qfe=qf1pElump)( ... )
```

This quadrature formula is exact on $\mathbb{P}_2$.

$$\int_D f(\mathbf{x}) \approx \frac{|D|}{2} \left(g(0) + g(1) \right)$$

4.3.2 int2d

Note: Complete formulas are no longer detailed

qf1pT

```
int2d(Th, qfe=qf1pT)( ... )
```

or

```
int2d(Th, qforder=2)( ... )
```

This quadrature formula is exact on $\mathbb{P}_1$.

qf2pT

```
int2d(Th, qfe=qf2pT)( ... )
```

or

```
int2d(Th, qforder=3)( ... )
```

This quadrature formula is exact on $\mathbb{P}_2$.

qf5pT

```
int2d(Th, qfe=qf5pT)( ... )
```

or

```
int2d(Th, qforder=6)( ... )
```

This quadrature formula is exact on $\mathbb{P}_5$.

qf1pTlump

```
int2d(Th, qfe=qf1pTlump)( ... )
```

This quadrature formula is exact on $\mathbb{P}_1$.

qf2pT4P1

```
int2d(Th, qfe=qf2pT4P1)( ... )
```

This quadrature formula is exact on $\mathbb{P}_1$.

qf7pT

```
int2d(Th, qfe=qf7pT)( ... )
```

or

```
int2d(Th, qforder=8)( ... )
```

This quadrature formula is exact on $\mathbb{P}_7$.

qf9pT

```
int2d(Th, qfe=qf9pT)( ... )
```

or

```
int2d(Th, qforder=10)( ... )
```

This quadrature formula is exact on $\mathbb{P}_9$.

4.3.3 int3d

qfV1

```
int3d(Th, qfe=qfV1)( ... )
```

or

```
int3d(Th, qforder=2)( ... )
```

This quadrature formula is exact on $\mathbb{P}_1$.

qfV2

```
int3d(Th, qfe=qfV2)( ... )
```

or

```
int3d(Th, qforder=3)( ... )
```

This quadrature formula is exact on $\mathbb{P}_2$.

qfV5

```
int3d(Th, qfe=qfV5)( ... )
```

or

```
int3d(Th, qforder=6)( ... )
```

This quadrature formula is exact on $\mathbb{P}_5$.

qfV1lump

```
int3d(Th, qfe=qfV1lump)( ... )
```

This quadrature formula is exact on $\mathbb{P}_1$.

4.4 Operators

4.4.1 Addition operator +

```
real a = 1. + 2.;
```

Works for `int`, `real`, `complex`, `string`, `mesh`, `mesh3`, array.

4.4.2 Substraction operator -

```
real a = 1. - 2.;
```

Works for `int`, `real`, `complex`, array.

4.4.3 Multiplication operator *

```
real[int] b;
matrix A
real[int] x = A^-1*b;
```

Works for `int`, `real`, `complex`, array, `matrix`.

4.4.4 Term by term multiplication .*

```
matrix A = B .* C;
```

4.4.5 Division operator /

```
real a = 1. / 2.;
```

Works for `int`, `real`, `complex`.

4.4.6 Term by term division ./

```
matrix A = B ./ C;
```

4.4.7 Remainder from the division %

```
int a = 1 % 2;
```

Works for `int`, `real`.

4.4.8 Power operator ^

```
real a = 2.^2;
```

Works for `int`, `real`, `complex`, `matrix`.

4.4.9 Inverse of a matrix ^-1

```
real[int] Res = A^-1 * b;
```

Warning: This operator can not be used to directly create a matrix, see *Matrix inversion*.

4.4.10 Transpose operator ‘

```
real[int] a = b';
```

Works for array and `matrix`.

Note: For `matrix<complex>`, the ::freefem‘’‘ operator return the Hermitian tranpose.

4.4.11 Tensor scalar product :

$$A : B = \sum_{i,j} A_{ij} B_{ij}$$

4.4.12 C++ arithmetical if expression ? :

`a ? b :` `c` is equal to `b` if the `a` is true, `c` otherwise.

Tip: Example with `int`

```
int a = 12; int b = 5;

cout << a << " + " << b << " = " << a + b << endl;
cout << a << " - " << b << " = " << a - b << endl;
cout << a << " * " << b << " = " << a * b << endl;
cout << a << " / " << b << " = " << a / b << endl;
cout << a << " % " << b << " = " << a % b << endl;
cout << a << " ^ " << b << " = " << a ^ b << endl;
cout << "( " << a << " < " << b << " ? " << a << " : " << b << ") = " << (a < b ? a :
↪b) << endl;
```

The output of this script is:

```
12 + 5 = 17
12 - 5 = 7
12 * 5 = 60
12 / 5 = 2
12 % 5 = 2
12 ^ 5 = 248832
( 12 < 5 ? 12 : 5) = 5
```

Tip: Example with `real`

```
real a = qsrt(2.); real b = pi;

cout << a << " + " << b << " = " << a + b << endl;
cout << a << " - " << b << " = " << a - b << endl;
cout << a << " * " << b << " = " << a * b << endl;
cout << a << " / " << b << " = " << a / b << endl;
cout << a << " % " << b << " = " << a % b << endl;
cout << a << " ^ " << b << " = " << a ^ b << endl;
cout << "( " << a << " < " << b << " ? " << a << " : " << b << ") = " << (a < b ? a :
↪b) << endl;
```

The output of this script is:

```
1.41421 + 3.14159 = 4.55581
1.41421 - 3.14159 = -1.72738
1.41421 * 3.14159 = 4.44288
1.41421 / 3.14159 = 0.450158
1.41421 % 3.14159 = 1
1.41421 ^ 3.14159 = 2.97069
```

4.5 Loops

See *Loop example*.

4.5.1 for

For loop.

```
for (int i = 0; i < N; ++i){
    ...
}
```

4.5.2 if

If condition.

```
if (condition){
    ...
}
else{
    ...
}
```

4.5.3 else

See *if*.

4.5.4 while

While loop.

```
while (condition){
    ...
}
```

4.5.5 continue

Continue a loop.

```
for (int i = 0; i < N; ++i){
    ...
    if (condition) continue;
    ...
}
```

4.5.6 break

Break a loop.

```
while (condition1){
    ...
    if (condition) break;
    ...
}
```

4.5.7 try

Try a part of code.

```
try{
    ...
}
catch(...){
    ...
}
```

See *Basic error handling example* and *Error handling example*.

4.5.8 catch

Catch an error, see *try*

4.5.9 Implicit loop

Array with one index:

```
for [i, ai : a]
```

If `real[int] a(10)`, then `i=0:9` and `ai` is a reference to `a[i]`.

Array with two indices or matrix:

```
for [i, j, aij : a]
```

If `real[int] a(10, 11)`, then `i=0:9`, `j=1:10` and `aij` is a reference to `a(i, j)`.

See *Implicit loop example*.

4.6 I/O

See *I/O example*

See *File stream example*.

4.6.1 cout

Standard C++ output device (default: console).

```
cout << "Some text" << endl;
```

4.6.2 cin

Standard C++ input device (default: keyboard).

```
cin >> var;
```

4.6.3 endl

End of line.

```
cout << "Some text" << endl;
```

4.6.4 ifstream

Open a file in read mode.

```
ifstream file("file.txt");
```

Note: A file is closed at the end of a block.

4.6.5 ofstream

Open a file in write mode.

```
ofstream file("file.txt");
```

Note: A file is closed at the end of a block.

4.6.6 append

Append data to an existing file.

```
ofstream file("file.txt", append);
```

4.6.7 binary

Write a file in binary.

```
ofstream file("file.btxt", binary);
```

4.6.8 seekg

Set the file position.

```
file.seekg(Pos);
```

4.6.9 tellg

Get the file position.

```
int Pos = file.tellg();
```

4.6.10 flush

Flush the buffer of the file.

```
file.flush
```

4.6.11 getline

Get the current line.

```
string s;
getline(file, s);
```

4.6.12 Output format

In the descriptions below, `f` is an output stream, for example `cout` or a `ofstream`.

All this methods, excepted the first, return a stream, so they can be chained:

```
cout.scientific.showpos << 3 << endl;
```

precision

Set the number of digits printed to the right of the decimal point. This applies to all subsequent floating point numbers written to that output stream. However, this won't make floating-point "integers" print with a decimal point. It's necessary to use `fixed` for that effect.

```
int np = f.precision(n)
```

scientific

Formats floating-point numbers in scientific notation

```
f.scientific
```

fixed

Used fixed point notation for floating-point numbers. Opposite of scientific.

```
f.fixed
```

showbase

Converts insertions to an external form that can be read according to the `C++` lexical conventions for integral constants. By default, showbase is not set.

```
f.showbase
```

noshowbase

Unset `showbase` flags.

```
f.noshowbase
```

showpos

Inserts a plus sign (+) into a decimal conversion of a positive integral value.

```
f.showpos
```

noshowpos

Unset `showpos` flags.

```
f.noshowpos
```

default

Reset all the previous flags to the default expect precision.

```
f.default
```

setw

Behaves as if member width were called with `n` as argument on the stream on which it is inserted as a manipulator (it can be inserted on output streams).

```
f.setw(n)
```

4.7 Functions

4.7.1 abs

Return the absolute value.

```
real a = abs(b);
```

Parameters:

- `b` (`real`)

Output:

- `a` (`real`)

4.7.2 acos

arccos function.

```
real theta = acos(x);
```

Parameter:

- `x` (`real`)

Output:

- `theta` (`real`)

4.7.3 acosh

```
real theta = acosh(x);
```

$$\operatorname{arccosh}(x) = \ln\left(x + \sqrt{x^2 - 1}\right)$$

Parameter:

- `x` (`real`)

Output:

Fig. 4.1: arccos function

- `theta` (`real`)

Fig. 4.2: arccosh function

4.7.4 adaptmesh

Mesh adaptation function.

```
mesh Thnew = adaptmesh(Th, [fx, fy], hmin=HMin, hmax=HMax, err=Err, errg=ErrG,
↪nbvx=NbVx, nbsmooth=NbSmooth, nbjacoby=NbJacoby, ratio=Ratio, omega=Omega, iso=Iso,
↪abserror=AbsError, cutoff=CutOff, verbosity=Verbosity, inquire=Inquire,
↪splitpbedge=SplitPbEdge, maxsubdiv=MaxSubdiv, rescaling=Rescaling,
↪keepbackvertices=KeepBackVertices, IsMetric=isMetric, power=Power,
↪thetamax=ThetaMax, splitin2=SplitIn2, metric=Metric,
↪nomeshgeneration=NoMeshGeneration, periodic=Periodic);
```

Parameters:

- `Th` (`mesh`) Mesh to refine

- `[fx, fy]` (`func` or `fespace` function), scalar or vectorial Function to follow for the mesh adaptation
- `hmin=` (`real`) Minimum edge size
- `hmax=` (`real`) Maximum edge size
- `err=` (`real`) Error level (P1 interpolation)
- `errg=` (`real`) Relative geometrical error
- `nbvx=` (`int`) Maximum number of vertices
- `nbsmooth=` (`int`) Number of smoothing iterations
- `nbjacoby=` (`int`) Number of iterations for the smoothing procedure
- `ratio=` (`real`) Ratio of the triangles
- `omega=` (`real`) Relaxation parameter for the smoothing procedure
- `iso=` (`bool`) Isotropic adaptation (if true)
- `abserror=` (`bool`) Error (if true) - Relative error (if false)
- `cutoff=` (`real`) Lower limit of the relative error evaluation
- `verbosity=` (`real`) Verbosity level
- `inquire=` (`bool`) If true, inquire graphically
- `splitpbedge=` (`bool`) If true, split all internal edges in half
- `maxsubdiv=` (`int`) Bound the maximum subdivisions
- `rescaling=` (`bool`) Rescale the function in [0, 1]
- `keepbackvertices=` (`bool`) If true, try to keep vertices of the original mesh
- `IsMetric=` (`bool`) If true, the metric is defined explicitly
- `power=` (`int`) Exponent of the Hessian
- `thetamax=` (`int`) Minimum corner angle (in degree)
- `splitin2=` (`bool`) Split all triangles into 4 sub-triangles if true
- `metric=` (`[real[int], real[int], real[int]]`) Array of 3 real arrays defining the metric
- `nomeshgeneration=` (`bool`) If true, the mesh is not generated
- `periodic=` (`real[int, int]`) Build an adapted periodic mesh

Output:

- `Thnew` (`mesh` or `mesh3`)

4.7.5 adj

Adjacent triangle of the triangle k by the edge e

```
int T = Th[k].adj(e);
```

Parameter:

- `e` (`int`) Edge number

Output:

- `T` (`int`) Triangle number

4.7.6 AffineCG

Affine conjugate gradient solver

Used to solve a problem like $Ax = b$

```
int Conv = AffineCG(A, x, precon=Precon, nbiter=NbIter, eps=Eps, veps=VEps,
↪stop=Stop);
```

Parameters:

- `A` (`matrix`) Matrix of the problem $Ax = b$
- `x` (`real[int]`) Solution vector
- `precon=` (`real[int]`) Preconditionning function
- `nbiter=` (`int`) Maximum number of iterations
- `eps=` (`real`)

  Convergence criterion

  If $\varepsilon > 0$: test $||A(x)||_p \leq \epsilon ||A(x_0)||_p$

  If $\varepsilon < 0$: test $||A(x)||_p^2 \leq |\epsilon|$
- `veps=` (`real`) Same as `eps`, but return `-eps`
- `stop=` (`func`) Convergence criterion as a function

  Prototype is `func bool StopFunc (int Iter, real[int] U, real[int] g)`

  `u`: current solution, `g`: current gradient (not preconditionned)

Output:

- Conv (int) 0: converged - !0: not converged

4.7.7 AffineGMRES

Affine GMRES solver

Parameters and output are the same as *AffineCG*

4.7.8 arg

Return the argument of a complex number.

```
real a = arg(c);
```

Parameters:

- `c` (`complex`)

Output:

- `r` (`real`)

4.7.9 asin

arcsin function.

```
real theta = asin(x);
```

Parameter:

- x (real)

Output:

- theta (real)

Fig. 4.3: arcsin function

4.7.10 asinh

```
real theta = asinh(x);
```

$$\operatorname{arcsinh}(x) = \ln\left(x + \sqrt{x^2 + 1}\right)$$

Parameter:

- x (real)

Output:

- theta (real)

4.7.11 assert

Verify if a condition is true (same as C), if not the program stops.

```
assert(x==0)
```

Parameter:

- Boolean condition

Fig. 4.4: arcsinh function

Output:

- None

4.7.12 atan

arctan function.

```
real theta = atan(x);
```

Parameter:

- `x` (`real`)

Output:

- `theta` (`real`)

Fig. 4.5: arctan function

4.7.13 atan2

$\arctan\left(\frac{y}{x}\right)$ function, returning the correct sign for θ.

```
real theta = atan2(y, x)
```

Parameter:

- `x` (`real`)

Output:

- `theta` (`real`)

4.7.14 atanh

```
real theta = atanh(x);
```

Parameter:

- `x` (`real`)

Output:

- `theta` (`real`)

Fig. 4.6: arctanh function

4.7.15 atoi

Convert a string to an interger.

```
int a = atoi(s);
```

Parameter:

- `s` (`string`)

Output:

- a (int)

4.7.16 atof

Convert a string to a real.

```
real a = atof(s);
```

Parameter:

- s (string)

Output:

- a (real)

4.7.17 BFGS

Todo: todo

4.7.18 buildmesh

Build a 2D mesh using border elements.

```
mesh Th = buildmesh(b1(nn) + b2(nn) + b3(nn) + b4(nn), [nbvx=Nbvx],
↪[fixedborder=FixedBorder]);
```

Parameters:

- b1, b2, b3, b4 (border)

  Geometry border, b1(nn) means b1 border discretized by nn vertices

- nbvx= (int) *[Optional]*

  Maximum number of vertices Default: 9000

- fixedborder= (bool) *[Optional]*

  If true, mesh generator cannot change the boundary mesh

  Default: false

Output:

- Th (mesh) Resulting mesh

4.7.19 ceil

Round fractions up of x.

```
int c = ceil(x);
```

Parameter:

- x (real)

Output:

- `c` (`int`)

4.7.20 change

Change a property of a mesh.

```
int[int] L = [0, 1];
Thnew = change(Th, label=L);
```

Parameters:

- `Th` (`mesh`) Original mesh
- `label=` L (`int[int]`) Pair of old and new label
- `region=` R (`int[int]`) Pair of old and new region
- `flabel=` l (`func int`) Function of int given the new label
- `fregion=` r (`func int`) Function of int given the new region

Output:

- `Thnew` (`mesh`) Mesh with changed parameters

4.7.21 chi

Characteristic function of a mesh.

```
int IsInMesh = chi(Th)(x, y);
```

Parameters:

- `Th` (`mesh` or `mesh3`)
- `x` (`real`) Position x
- `y` (`real`) Position y

Output:

- `IsInMesh` (`int`) 1 if $(x, y) \in$ `Th` 0 if $(x, y) \notin$ `Th`

4.7.22 checkmovemesh

Check a *movemesh* without mesh generation.

```
real minT = checkmovemesh(Th, [Dx, Dy]);
```

Parameters:

Same as *movemesh*

Output:

- `minT` (`real`) Minimum triangle area

4.7.23 clock

Get the clock in second.

```
real t = clock();
```

Parameter:

- None

Output:

- `t` (`real`) Current CPU time

4.7.24 complexEigenValue

Same as *EigenValue* for complex problems.

4.7.25 conj

Caculate the conjuguate of a complex number.

```
complex C1 = 1 + 1i;
complex C2 = conj(C1);
```

Parameter:

- `C1` (`complex`) Complex number

Output:

- `C2` (`complex`) Conjuguate of C1

4.7.26 convect

Characteristics Galerkin method.

```
real cgm = convect([Ux, Uy], dt, c);
real cgm = convect([Ux, Uy, Uz], dt, c);
```

Compute $c \circ \mathbf{X}$ with $\mathbf{X}(\mathbf{x}) = \mathbf{x}_\tau$ and $\mathbf{x}_\tau$ is the solution of:

$$\begin{array}{rcl} \dot{\mathbf{x}}_\tau & = & \mathbf{u}(\mathbf{x}_\tau) \\ \mathbf{x}_\tau & = & \mathbf{x} \end{array}$$

Parameters:

- `ux` (`fespace` function) Velocity: x component
- `uy` (`fespace` function) Velocity: y component
- `uz` (`fespace` function) **3D only**

  Velocity: z component
- `dt` (`real`) Time step
- `c` (`fespace` function) Function to convect

Output:

- `cgm` (`real`) Result

4.7.27 copysign

C++ `copysign` function.

```
real s = copysign(a, b);
```

4.7.28 cos

cos function.

```
real x = cos(theta);
```

Parameters:

- `theta` (`real` or `complex`)

Output:

- `x` (`real` or `complex`)

Fig. 4.7: cos function

4.7.29 cosh

cosh function.

```
real x = cosh(theta);
```

$$\cosh(x) = \frac{e^x + e^{-x}}{2}$$

Parameters:

- `theta` (`real`)

Output:

- x (real)

4.7.30 diffnp

Arithmetic useful function.

```
diffnp(a, b) = (a<0)&(0<b) ? (b-a) : 0;
```

4.7.31 diffpos

Arithmetic useful function.

```
diffpos(a, b) = max(b-a, 0);
```

4.7.32 dist

Arithmetic useful function.

```
dist(a, b) = sqrt(a^2 + b^2);
dist(a, b, c) = sqrt(a^2 + b^2 + c^2);
```

4.7.33 dumptable

Show all types, operators and functions in **FreeFEM**.

```
dumptable(out);
```

Parameters:

- out (ostream) cout of ofstream file.

Output:

- None

4.7.34 dx

x derivative.

```
Uh up = dx(u);
```

$$\frac{\partial u}{\partial x}$$

Parameters:

- u (fespace function)

Output:

- up (fespace function)

4.7.35 dxx

x double derivative.

```
Uh upp = dxx(u);
```

$$\frac{\partial^2 u}{\partial x^2}$$

Parameters:

- `u` (`fespace` function)

Output:

- `upp` (`fespace` function)

4.7.36 dxy

xy derivative.

```
Uh upp = dxy(u);
```

$$\frac{\partial^2 u}{\partial x \partial y}$$

Parameters:

- `u` (`fespace` function)

Output:

- `upp` (`fespace` function)

4.7.37 dxz

xz derivative.

```
Uh upp = dxz(u);
```

$$\frac{\partial^2 u}{\partial x \partial z}$$

Parameters:

- `u` (`fespace` function)

Output:

- `upp` (`fespace` function)

4.7.38 dy

y derivative.

```
Uh up = dy(u);
```

$$\frac{\partial u}{\partial y}$$

Parameters:

- u (fespace function)

Output:

- upp (fespace function)

4.7.39 dyx

yx derivative.

```
Uh upp = dyx(u);
```

$$\frac{\partial^2 u}{\partial y \partial x}$$

Parameters:

- u (fespace function)

Output:

- upp (fespace function)

4.7.40 dyy

y double derivative.

```
Uh upp = dyy(u);
```

$$\frac{\partial^2 u}{\partial x^2}$$

Parameters:

- u (fespace function)

Output:

- upp (fespace function)

4.7.41 dyz

yz derivative.

```
Uh upp = dyz(u);
```

$$\frac{\partial^2 u}{\partial y \partial z}$$

Parameters:

- u (fespace function)

Output:

- upp (fespace function)

4.7.42 dz

z derivative.

```
Uh up = dz(u);
```

$$\frac{\partial u}{\partial z}$$

Parameters:

- `u` (`fespace` function)

Output:

- `upp` (`fespace` function)

4.7.43 dzx

zx derivative.

```
Uh upp = dzx(u);
```

$$\frac{\partial^2 u}{\partial z \partial x}$$

Parameters:

- `u` (`fespace` function)

Output:

- `upp` (`fespace` function)

4.7.44 dzy

zy derivative.

```
Uh upp = dzy(u);
```

$$\frac{\partial^2 u}{\partial z \partial y}$$

Parameters:

- `u` (`fespace` function)

Output:

- `upp` (`fespace` function)

4.7.45 dzz

z double derivative.

```
Uh upp = dzz(u);
```

$$\frac{\partial^2 u}{\partial z^2}$$

Parameters:

- `u` (`fespace` function)

Output:

- `upp` (`fespace` function)

4.7.46 EigenValue

Compute the generalized eigenvalue of $Au = \lambda Bu$. The shifted-inverse method is used by default with `sigma=`σ, the shift of the method. The function `EigenValue` can be used for either matrices or functions returning a matrix vector product. The use of the matrix version is shown below.

```
int k = EigenValue(A,B,nev= , sigma= );
```

Parameters:

- `A`, `B`: matrices of same size
- `nev=n`: number of desired eigenvalues given by an integer `n`
- `sym=`: the problem is symmetric or not
- `tol=`: the relative accuracy to which eigenvalues are to be determined
- `value=`: an array to store the real part of the eigenvalues
- `ivalue=`: an array to store the imaginary part of the eigenvalues
- `vector=`: a Finite Element function array to store the eigenvectors
- `sigma=`: the shift value
- Other parameters are available for more advanced use: see the ARPACK documentation.

Output: The output is the number of converged eigenvalues, which can be different than the number of requested eigenvalues given by `nev=`. Note that the eigenvalues and the eigenvectors are stored for further purposes using the parameters `value=` and `vector=`.

4.7.47 emptymesh

Build an empty mesh.

Useful to handle Lagrange multipliers in mixed and Mortar methods.

```
mesh eTh = emptymesh(Th, ssd);
```

Parameters:

- `Th` (`mesh`) Mesh to empty
- `ssd` (`int[int]`) Pseudo subregion label

Output:

- `eTh` (`mesh`) Empty mesh

4.7.48 erf

The error function:

$$erf(x) = \frac{2}{\sqrt{pi}} \int_0^x \exp(-t^2)dt$$

```
real err = erf(x);
```

Parameters:

- `x` (`real`)

Output:

- `err` (`real`)

4.7.49 erfc

Complementary of the *error function*:

$$erfc(x) = 1 - erf(x)$$

```
real errc = erfc(x);
```

Parameters:

- `x` (`real`)

Output:

- `err` (`real`)

4.7.50 exec

Execute an external command.

```
int v = exec(command);
```

Parameters:

- `command` (`string`) Command to execute

Output:

- `v` (`int`) Value returned by the command

4.7.51 exit

Exit function, equivalent to `return`.

```
exit(N);
```

Parameters:

- `N` (`int`) Return value

Output:

- None

4.7.52 exp

Exponential function.

```
real a = exp(b);
```

Parameters:

- b (real or complex)

Output:

- a (real or complex)

4.7.53 fdim

Positive difference (cmath function).

```
real fd = fdim(a, b);
```

Parameters:

- a (real)
- b (real)

Output:

- fd (real) If $x > y$, return $x - y$If $x \leq y$, return 0

4.7.54 floor

Floor function.

```
real a = floor(b);
```

Return the largest integer value not greater than b.

Parameters:

- b (real)

Output:

- a (real)

4.7.55 fmax

Maximum (cmath function).

```
real Max = fmax(a, b);
```

Parameters:

- a (real)
- b (real)

Output:

- Max (real)

4.7.56 fmin

Minimum (cmath function).

```
real Min = fmin(a, b);
```

Parameters:

- a (real)
- b (real)

Output:

- Min (real)

4.7.57 fmod

Remainder of a/b (cmath function).

```
real Mod = fmod(a, b);
```

Parameters:

- a (real)
- b (real)

Output:

- Min (real)

Parameters:

- a (real)
- b (real)

Output:

- Mod (real)

4.7.58 imag

Imaginary part of a complex number.

```
complex c = 1. + 1i;
real Im = imag(c);
```

4.7.59 int1d

1D integral.

```
int1d(Th, [Label], [qfe=Qfe], [qforder=Qforder])(
    ...
)
```

Used in *problem*, *solve* or *varf* definition to impose a boundary condition only (**FreeFEM** does not support 1D simulation), or outside to calculate a quantity.

Parameters:

- `Th` (`mesh`) Mesh where the integral is calculated
- `Label` (`int`) *[Optional]*

  Label of the 1D border Default: all borders of the mesh
- `qfe=` (*quadrature formula*) *[Optional]*

  Quadrature formula, see *quadrature formulae*
- `qforder=` (*quadrature formula*) *[Optional]*

  Quadrature order, see *quadrature formulae*

Output:

- Depending on the situation: In a `problem`, `solve` or `varf` definition: Non relevant.

  Outside: `real` (example: `real l = int1d(Th, 1)(1.);`).

> **Warning:** In a `problem`, `solve` or `varf` definition, the content of `int1d` must be a linear or bilinear form.

4.7.60 int2d

2D integral.

```
int2d(Th, [Region], [qfe=Qfe], [qforder=Qforder])(
    ...
)
```

Or

```
int2d(Th, [Label], [qfe=Qfe], [qforder=Qforder])(
    ...
)
```

Used in *problem*, *solve* or *varf* definition to: - Calculate integral in 2D simulation - Impose a boundary condition in 3D simulation Or outside to calculate a quantity.

Parameters:

- `Th` (`mesh` or `mesh3`) Mesh where the integral is calculated
- `Region` (`int`) *[Optional]* Label of the 2D region (2D simulation) Default: all regions of the mesh
- `Label` (`int`) *[Optional]* Label of the 2D border (3D simulation) Default: all borders of the mesh

- `qfe=` (*quadrature formula*) *[Optional]*

  Quadrature formula, see *quadrature formulae*

- `qforder=` (*quadrature formula*) *[Optional]*

  Quadrature order, see *quadrature formulae*

Output:

- Depending on the situation: In a `problem`, `solve` or `varf` definition: Non relevant. Outside: `real` (example: `real s = int2d(Th, 1)(1.);`).

Warning: In a `problem`, `solve` or `varf` definition, the content of the `int2d` must be a linear or bilinear form.

4.7.61 int3d

3D integral.

```
int3d(Th, [Region], [qfe=Qfe], [qforder=Qforder])(
    ...
)
```

Used in *problem*, *solve* or *varf* definition to calculate integral in 3D simulation, or outside to calculate a quantity.

Parameters:

- `Th` (`mesh3`) Mesh where the integral is calculated
- `Region` (`int`) *[Optional]*

  Label of the 3D region

  Default: all regions of the mesh

- `qfe=` (*quadrature formula*) *[Optional]*

  Quadrature formula, see *quadrature formulae*

- `qforder=` (*quadrature formula*) *[Optional]*

  Quadrature order, see *quadrature formulae*

Output:

- Depending on the situation: In a `problem`, `solve` or `varf` definition: Non relevant. Outside: `real` (example: `real v = int3d(Th, 1)(1.);`).

Warning: In a `problem`, `solve` or `varf` definition, the content of the `int3d` must be a linear or bilinear form.

4.7.62 intalledges

Integral on all edges.

```
intalledges(Th, [Region])(
    ...
)
```

Parameters:

- `Th` (`mesh`) Mesh where the integral is calculated
- `Region` (`int`) *[Optional]*

  Label of the region

  Default: all regions of the mesh

Output:

- Non relevant

4.7.63 intallfaces

Intergal on all faces.

Same as *intalledges* for `mesh3`.

4.7.64 interpolate

Interpolation operator from a finite element space to another.

```
matrix I = interpolate(Wh, Vh, [inside=Inside], [t=T], [op=Op], [U2Vc=U2VC]);
```

Parameters:

- `Wh` (`fespace`) Target finite element space
- `Vh` (`fespace`) Original finite element space
- `inside=` (`bool`) If true, create a zero extension outside the `Vh` domain
- `t=` (`bool`) If true, return the transposed matrix
- `op=` (`int`) 0: interpolate the function (default value) 1: interpolate ∂_x 2: interpolate ∂_y 3: interpolate ∂_z
- `U2Vc=` (`int[int]`) Array of the same size of `Wh` describing which component of `Vh`is interpolated in `Wh`

Output:

- `I` (`matrix`) Interpolation matrix operator

4.7.65 invdiff

Arithmetic useful function.

```
invdif(a, b) = (-abs(a-b) > 10^(-30)) ? 1(/b-a) : 0
invdif(a, b, e) = (-abs(a-b) > e) ? 1(/b-a) : 0
```

4.7.66 invdiffnp

Arithmetic useful function.

```
invdiffnp(a, b) = (a<0)&(0<b) ? 1/(b-a) : 0
```

4.7.67 invdiffpos

Arithmetic useful function.

```
invdiffpos(a, b) = (a<b) ? 1./(b-a) : 0
```

4.7.68 isInf

The C++ `isInf` function.

```
int i = isInf(a);
```

4.7.69 isNaN

The C++ `isNan` function.

```
int i = isNaN(a);
```

4.7.70 j0

Bessel function of first kind, order 0.

```
real b = j0(x);
```

Parameters:

- `x` (`real`)

Output:

- `b` (`real`)

4.7.71 j1

Bessel function of first kind, order 1.

```
real b = j1(x);
```

Parameters:

- `x` (`real`)

Output:

- `b` (`real`)

4.7.72 jn

Bessel function of first kind, order n.

```
real b = jn(n, x);
```

$$J_n(x) = \sum_{p=0}^{\infty} \frac{(1)^p}{p!(n+p)!} \left(\frac{x}{2}\right)^{2p+n}$$

Parameters:

- `n` (`int`)
- `x` (`real`)

Output:

- `b` (`real`)

4.7.73 jump

Jump function across an edge.

```
intalledges(
    ... jump(c) ...
)
```

Parameters:

- `c` (`fespace` function) Discontinuous function

Output:

- Non relevant

4.7.74 LinearCG

Linear CG solver

Parameters and output are the same as *AffineCG*

4.7.75 LinearGMRES

Linear GMRES solver

Parameters and output are the same as *AffineCG*

4.7.76 lgamma

Natural logarithm of the absolute value of the Γ function of x.

```
real lg = lgamma(x);
```

Parameters:

- `x` (`real`)

Output:

- `lg` (`real`)

4.7.77 log

Natural logarithm.

```
real l = log(x);
```

Parameters:

- `x` (`real` or `complex`)

Output:

- `l` (`real` or `complex`)

Note: Complex value

For complex value, the `log` function is defined as:

$$\log(z) = \log(|z|) + i \arg(z)$$

4.7.78 log10

Common logarithm.

```
real l = log10(x);
```

Parameters:

- `x` (`real`)

Output:

- `l` (`real`)

4.7.79 lrint

Integer value nearest to x.

```
int l = lrint(a);
```

Parameters:

- `a` (`real`)

Output:

- `l` (`int`)

4.7.80 lround

Round a value, and return an integer value.

```
int l = lround(a);
```

Parameters:

- a (real)

Output:

- l (int)

4.7.81 max

Maximum value of two values.

```
real m = max(a, b);
real m = max(a, b, c);
```

Parameters:

- a (int or real)
- b (int or real)
- c (int or real) *[Optional]*

Output:

- b (int or real)

4.7.82 min

Minimum value of two values.

```
real m = min(a, b);
real m = min(a, b, c);
```

Parameters:

- a (int or real)
- b (int or real)
- c (int or real) *[Optional]*

Output:

- b (int or real)

4.7.83 movemesh

Move a mesh.

```
mesh MovedTh = movemesh(Th, [Dx, Dy]);
mesh3 MovedTh = movemesh(Th, [Dx, Dy, Dz], [region=Region], [label=Label],
↪[facemerge=FaceMerge], [ptmerge=PtMerge], [orientation=Orientation]);
```

Parameters:

- Th (mesh of mesh3) Mesh to move
- Dx (fespace function) Displacement along x
- Dy (fespace function) Displacement along y

- `Dz` (`fespace` function) **3D only**

  Displacement along z
- `region=` (`int`) *[Optional]* **3D only**

  Set label to tetrahedra
- `label=` (`int[int]`) *[Optional]* **3D only**

  Set label of faces (see *change* for more information)
- `facemerge=` (`int`) *[Optional]* **3D only**

  If equal to 1, some faces can be merged during the mesh moving Default: 1
- `ptmerge=` (`real`) *[Optional]* **3D only**

  Criteria to define when two points merge
- `orientation=` (`int`) *[Optional]* **3D only**

  If equal to 1, allow orientation reverse if tetrahedra is not positive Default: 1

Output:

- `MovedTh` (`mesh` or `mesh3`) Moved mesh

4.7.84 NaN

C++ `nan` function.

```
real n = NaN([String]);
```

Parameters:

- `String` (`string`) Default: `""`

4.7.85 NLCG

Non-linear conjugate gradient.

Parameters and output are the same as *AffineCG*

4.7.86 on

Dirichlet condition function.

```
problem (u, v)
    ...
    + on(Label, u=uD)
    ...
```

Warning: Used only in problem, solve and varf

Parameters:

- `Label` (`int` or `border` in 2D)

  Boundary reference where to impose the Dirichlet condition

- `uD` (`fespace` function, `func` or `real` or `int`)

  Dirichlet condition (`u` is an unknown of the problem)

Output:

- Non relevant

4.7.87 plot

Plot meshes and results.

```
plot([Th], [u], [[Ux, Uy, Uz]], [wait=Wait], [ps=PS], [coef=Coef], [fill=Fill],
↪cmm=[Cmm], [value=Value], [aspectratio=AspectRatio], [bb=Bb], [nbiso=NbIso],
↪[nbarrow=NbArrow], [viso=VIso], [varrow=VArrow], [bw=Bw], [grey=Grey], [hsv=Hsv],
↪[boundary=Boundary], [dim=Dim], [prev=Prev], [WindowIndex=WI]);
```

Note: Only one of `Th`, `u` or `[Ux, Uy]` / `[Ux, Uy, Uz]` is needed for the `plot` command.

Parameters:

- `Th` (`mesh` or `mesh3`) Mesh to display
- `u` (`fespace` function) Scalar `fespace` function to display
- `[Ux, Uy]` / `[Ux, Uy, Uz]` (`fespace` function array) Vectorial `fespace` function to display
- `[Ux, Uy]` (`[real[int], real[int]]`) Couple a real array to display a curve
- `wait=` (`bool`) If true, wait before continue
- `ps=` (`string`) Name of the file to save the plot (`.ps` or `.eps format`)
- `coef=` (`real`) Arrow size
- `fill=` (`bool`) If true, fill color between isovalue (usable with scalar `fespace` function only)
- `cmm=` (`string`) Text comment in the graphic window
- `value=` (`bool`) If true, show the value scale
- `aspectratio=` (`bool`) If true, preserve the aspect ratio
- `bb=` (`[real[int], real[int]]`) Specify a bounding box using two corner points
- `nbiso=` (`int`) Number of isovalues
- `nbarrow=` (`int`) Number of colors of arrows values
- `viso=` (`real[int]`) Specify an array of isovalues
- `varrow=` (`real[int]`) Specify an array of arrows values color
- `bw=` (`bool`) If true, the plot is in black and white
- `grey=` (`bool`) If true, the plot is in grey scale
- `hsv=` (`real[int]`) Array of $3 \times n$ values defining HSV color model $[h_1, s_1, v_1, ..., h_n, s_n, v_n]$
- `boundary=` (`bool`) If true, display the boundary of the domain

- `dim=` (`int`) Set the dimension of the plot: 2 or 3
- `prev=` (`bool`) Use the graphic state of the previous state
- `WindowIndex=` (`int`) Specify window index for multiple windows graphics

Output:

- None

See the *plot* section for in-graphic commands.

4.7.88 polar

Polar coordinates.

```
complex p = polar(a, b);
```

Parameters:

- `a` (`real`)
- `b` (`real`)

Output:

- `p` (`complex`)

4.7.89 pow

Power function.

```
real p = pow(a, b);
```

$p = a^b$

Parameters:

- `a` (`real`)
- `b` (`real`)

Output:

- `p` (`real`)

4.7.90 projection

Arithmetic useful function.

```
real p = projection(a, b, x);
```

Projection is equivalent to:

```
projection(a, b, x) = min(max(a, x), b)*(a < b) + min(max(b, x), a)*(1-(a < b));
```

Parameters:

- `a` (`real`)

- `b` (`real`)
- `x` (`real`)

Output:

- `p` (`real`)

4.7.91 randinit

Initialize the state vector by using a seed.

```
randinit(seed);
```

Parameters:

- `seed` (`int`)

Output:

- None

4.7.92 randint31

Generate `unsigned int` (31 bits) random number.

```
int r = randint31();
```

Parameters:

- None

Output:

- `r` (`int`)

4.7.93 randint32

Generate `unsigned int` (32 bits) random number.

```
int r = randint32();
```

Parameters:

- None

Output:

- `r` (`int`)

4.7.94 randreal1

Generate uniform `real` in $[0, 1]$ (32 bits).

```
real r = randreal1();
```

Parameters:

- None

Output:

- `r` (`real`)

4.7.95 randreal2

Generate uniform `real` in $[0, 1)$ (32 bits).

```
real r = randreal2();
```

Parameters:

- None

Output:

- `r` (`real`)

4.7.96 randreal3

Generate uniform `real` in $(0, 1)$ (32 bits).

```
real r = randreal3();
```

Parameters:

- None

Output:

- `r` (`real`)

4.7.97 randres53

Generate uniform `real` in $[0, 1)$ (53 bits).

```
real r = randres53();
```

Parameters:

- None

Output:

- `r` (`real`)

4.7.98 readmesh

Read a 2D mesh file at different formats (see *Mesh Generation*).

```
mesh Th = readmesh(MeshFileName);
```

Parameters:

- `MeshFileName` (`string`)

Output:

- `Th` (`mesh`)

4.7.99 readmesh3

Read a 3D mesh file at different formats (see *Mesh Generation*).

```
mesh3 Th = readmesh3(MeshFileName);
```

Parameters:

- `MeshFileName` (`string`)

Output:

- `Th` (`mesh3`)

4.7.100 real

Return the real part of a complex number.

```
real r = real(c);
```

Parameters:

- `c` (`complex`)

Output:

- `r` (`real`)

4.7.101 rint

Integer value nearest to x (real value).

```
real r = rint(a);
```

Parameters:

- `a` (`real`)

Output:

- `r` (`real`)

4.7.102 round

Round a value (real value).

```
real r = round(a);
```

Parameters:

- `a` (`real`)

Output:

- `r` (`real`)

4.7.103 savemesh

Save a 2D or 3D mesh in different formats (see *Mesh Generation 2D* and *Mesh Generation 3D*).

```
savemesh(Th, MeshFileName);
```

Parameters:

- `Th` (`mesh` or `mesh3`)
- `MeshFileName` (`string`)

Output:

- None

4.7.104 set

Set a property to a matrix. See *matrix*.

4.7.105 sign

Sign of a value.

```
int s = sign(a);
```

Parameters:

- `a` (`real` or `int`)

Output:

- `s` (`int`)

4.7.106 signbit

C++ `signbit` function

```
int s = signbit(a);
```

4.7.107 sin

sin function.

```
real x = sin(theta);
```

Parameter:

- `theta` (`real` or `complex`)

Output:

- `x` (`real` or `complex`)

Fig. 4.8: sin function

4.7.108 sinh

sinh function.

```
real x = sinh(theta);
```

$$\sinh(x) = \frac{e^x - e^{-x}}{2}$$

Parameter:

- `theta` (`real`)

Output:

- `x` (`real`)

Fig. 4.9: sinh function

4.7.109 sort

Sort two array in parallel

```
sort(A, B);
```

Parameters:

- A (real[int])
- B (int[int])

Output:

- None

A is sorted in ascending order, B is sorted as A.

4.7.110 splitmesh

Split mesh triangles according to a function.

```
Th = splitmesh(Th0, f);
```

Parameters:

- Th0 (mesh)
- f (func or fespace function)

Output:

- Th (mesh)

4.7.111 sqrt

Square root

```
real s = sqrt(a);
```

Parameter:

- a (real)

Output:

- s (real)

4.7.112 square

1. Square of a number.

```
real S = square(a);
```

Parameter:

- a (real)

Output:

- S (real)

2. Build a structured square mesh.

```
mesh Th = square(nnX, nnY, [[L*x, H*y]], [flags=Flags]);
```

Parameters:

- nnX (int) Discretization along x
- nnY (int) Discretization along y
- L (real) *[Optional]* Length along x
- H (real) *[Optional]* Height along y
- flags= (int) *[Optional]*

  Structured mesh type, see *Mesh Generation chapter* for more information

Output:

- Th (mesh)

4.7.113 swap

Swap values.

```
swap(a, b);
```

Parameters:

- a (real)
- b (real)

Output:

- None

4.7.114 system

Execute a system command.

```
int Res = system(Command);
```

Parameter:

- Command (string) System command

Output:

- Res (int) Value returned by the system command

Note: On Windows, the full path of the command is needed. For example, to execute ls.exe:

```
int Res = exec("C:\\cygwin\\bin\\ls.exe");
```

4.7.115 tan

tan function.

```
real x = tan(theta);
```

Parameter:

- `theta` (`real`)

Output:

- `x` (`real`)

Fig. 4.10: tan function

4.7.116 tanh

tanh function.

```
real x = tanh(theta);
```

Parameter:

- `theta` (`real`)

Output:

- `x` (`real`)

4.7.117 tgamma

Calculate the Γ function of x.

```
real tg = tgamma(x);
```

Parameter:

- `x` (`real`)

Fig. 4.11: tanh function

Output:

- `tg` (`real`)

4.7.118 trunc

Split triangle of a mesh.

```
mesh Th = trunc(Th0, R, [split=Split], [label=Label]);
```

Parameters:

- `Th0` (`mesh`)
- `R` (`bool` or `int`) Split triangles where `R` is true or different from 0
- `split=` (`int`) *[Optional]*

  Level of splitting Default: 1
- `label=` (`int`) *[Optional]*

  Label number of new boundary item Default: 1

Output:

- `Th` (`mesh`)

4.7.119 y0

Bessel function of second kind, order 0.

```
real B = y0(x);
```

Parameters:

- `x` (`real`)

Output:

- b (real)

4.7.120 y1

Bessel function of second kind, order 1.

```
real B = y1(x);
```

Parameters:

- x (real)

Output:

- b (real)

4.7.121 yn

Bessel function of second kind, order n.

```
real B = yn(n, x);
```

$$Y_n(x) = \lim_{\lambda \to n} \frac{J_\lambda(x)\cos(\lambda\pi) - J_{-\lambda}(x)}{\sin(\lambda\pi)}$$

Parameters:

- n (int)
- x (real)

Output:

- b (real)

4.8 External libraries

4.8.1 aniso

boundaniso

Todo: todo

4.8.2 BEC

BECtrap

Todo: todo

GPvortex

Todo: todo

dxGPVortex

Todo: todo

dyGPVortex

Todo: todo

4.8.3 Binary I/O

LoadVec

Todo: todo

LoadFlag

Todo: todo

SaveVec

Todo: todo

flag

Todo: todo

4.8.4 buildlayer

buildlayers

Todo: todo

4.8.5 ClosePoints

radiusSearch

Todo: todo

Voisinage

Todo: todo

neighborhood

Todo: todo

ClosePoints2

Todo: todo

ClosePoint

Todo: todo

ClosePoints1

Todo: todo

4.8.6 Curvature

extractborder

Extract a border of a mesh.

```
int Res = extractborder(Th, Label, Points);
```

Parameters:

- `Th` (`mesh` or `mesh3`)
- `Label` (`int`) Label of the border to extract
- `Points` (`real[int, int]`) Extracted points Must be allocated as `real[int, int] Points(3, 1);`

Output:

- `Res` (`real`) Length of the extracted border

curvature

Todo: todo

raxicurvature

Todo: todo

curves

Todo: todo

setecurveabcisse

Todo: todo

equiparameter

Todo: todo

Tresca

Todo: todo

VonMises

Todo: todo

4.8.7 dfft

Refer to the FFTW documentation for more informations.

plandfft

Todo: todo

execute

Todo: todo

delete

Todo: todo

dfft

Todo: todo

map

Todo: todo

4.8.8 distance

Need

```
load "distance"
```

distance

```
distance(Th, d, dist, [distmax=DistMax]);
```

Parameters:

- `Th` (`mesh`)
- `d`
- `dist` (`real[int]`)

Output:

-

Todo: todo

checkdist

Todo: todo

4.8.9 DxWriter

Dxaddmesh

Todo: todo

Dxaddtimeseries

Todo: todo

Dxaddsol2ts

Todo: todo

4.8.10 Element_P1bl

expert

Todo: todo

4.8.11 exactpartition

exactpartition

Todo: todo

4.8.12 ff-AiryBiry

airy

Todo: todo

biry

Todo: todo

4.8.13 ff-cmaes

cmaes

Todo: todo

4.8.14 ff_gsl_awk

Refer to the GSL documentation for more informations

gslcdfugaussianP

Link to:

```
gsl_cdf_ugaussian_P(a)
```

gslcdfugaussianQ

Link to:

```
gsl_cdf_ugaussian_Q(a)
```

gslcdfugaussianPinv

Link to:

```
gsl_cdf_ugaussian_Pinv(a)
```

gslcdfugaussianQinv

Link to:

```
gsl_cdf_ugaussian_Qinv(a)
```

gslcdfgaussianP

Link to:

```
gsl_cdf_gaussian_P(a, b)
```

gslcdfgaussianQ

Link to:

```
gsl_cdf_gaussian_Q(a, b)
```

gslcdfgaussianPinv

Link to:

```
gsl_cdf_gaussian_Pinv(a, b)
```

gslcdfgaussianQinv

Link to:

```
gsl_cdf_gaussian_Qinv(a, b)
```

gslcdfgammaP

Link to:

```
gsl_cdf_gamma_P(a, b, c)
```

gslcdfgammaQ

Link to:

```
gsl_cdf_gamma_Q(a, b, c)
```

gslcdfgammaPinv

Link to:

```
gsl_cdf_gamma_Pinv(a, b, c)
```

gslcdfgammaQinv

Link to:

```
gsl_cdf_gamma_Pinv(a, b, c)
```

gslcdfcauchyP

Link to:

```
gsl_cdf_cauchy_P(a, b)
```

gslcdfcauchyQ

Link to:

```
gsl_cdf_cauchy_Q(a, b)
```

gslcdfcauchyPinv

Link to:

```
gsl_cdf_cauchy_Pinv(a, b)
```

gslcdfcauchyQinv

Link to:

```
gsl_cdf_cauchy_Qinv(a, b)
```

gslcdflaplaceP

Link to:

```
gsl_cdf_lapalce_P(a, b)
```

gslcdflaplaceQ

Link to:

```
gsl_cdf_lapalce_Q(a, b)
```

gslcdflaplacePinv

Link to:

```
gsl_cdf_lapalce_Pinv(a, b)
```

gslcdflaplaceQinv

Link to:

```
gsl_cdf_lapalce_Qinv(a, b)
```

gslcdfrayleighP

Link to:

```
gsl_cdf_rayleigh_P(a, b)
```

gslcdfrayleighQ

Link to:

```
gsl_cdf_rayleigh_Q(a, b)
```

gslcdfrayleighPinv

Link to:

```
gsl_cdf_rayleigh_Pinv(a, b)
```

gslcdfrayleighQinv

Link to:

```
gsl_cdf_rayleigh_Qinv(a, b)
```

gslcdfchisqP

Link to:

```
gsl_cdf_chisq_P(a, b)
```

gslcdfchisqQ

Link to:

```
gsl_cdf_chisq_Q(a, b)
```

gslcdfchisqPinv

Link to:

```
gsl_cdf_chisq_Pinv(a, b)
```

gslcdfchisqQinv

Link to:

```
gsl_cdf_chisq_Qinv(a, b)
```

gslcdfexponentialP

Link to:

```
gsl_cdf_exponential_P(a, b)
```

gslcdfexponentialQ

Link to:

```
gsl_cdf_exponential_Q(a, b)
```

gslcdfexponentialPinv

Link to:

```
gsl_cdf_exponential_Pinv(a, b)
```

gslcdfexponentialQinv

Link to:

```
gsl_cdf_exponential_Qinv(a, b)
```

gslcdfexppowP

Link to:

```
gsl_cdf_exppow_P(a, b, c)
```

gslcdfexppowQ

Link to:

```
gsl_cdf_exppow_Q(a, b, c)
```

gslcdftdistP

Link to:

```
gsl_cdf_t_dist_P(a, b)
```

gslcdftdistQ

Link to:

```
gsl_cdf_t_dist_Q(a, b)
```

gslcdftdistPinv

Link to:

```
gsl_cdf_t_dist_Pinv(a, b)
```

gslcdftdistQinv

Link to:

```
gsl_cdf_t_dist_Qinv(a, b)
```

gslcdffdistP

Link to:

```
gsl_cdf_fdist_P(a, b, c)
```

gslcdffdistQ

Link to:

```
gsl_cdf_fdist_Q(a, b, c)
```

gslcdffdistPinv

Link to:

```
gsl_cdf_fdist_Pinv(a, b, c)
```

gslcdffdistQinv

Link to:

```
gsl_cdf_fdist_Qinv(a, b, c)
```

gslcdfbetaP

Link to:

```
gsl_cdf_beta_P(a, b, c)
```

gslcdfbetaQ

Link to:

```
gsl_cdf_beta_Q(a, b, c)
```

gslcdfbetaPinv

Link to:

```
gsl_cdf_beta_Pinv(a, b, c)
```

gslcdfbetaQinv

Link to:

```
gsl_cdf_beta_Qinv(a, b, c)
```

gslcdfflatP

Link to:

```
gsl_cdf_flat_P(a, b, c)
```

gslcdfflatQ

Link to:

```
gsl_cdf_flat_Q(a, b, c)
```

gslcdfflatPinv

Link to:

```
gsl_cdf_flat_Pinv(a, b, c)
```

gslcdfflatQinv

Link to:

```
gsl_cdf_flat_Qinv(a, b, c)
```

gslcdflognormalP

Link to:

```
gsl_cdf_lognormal_P(a, b, c)
```

gslcdflognormalQ

Link to:

```
gsl_cdf_lognormal_Q(a, b, c)
```

gslcdflognormalPinv

Link to:

```
gsl_cdf_lognormal_Pinv(a, b, c)
```

gslcdflognormalQinv

Link to:

```
gsl_cdf_lognormal_Qinv(a, b, c)
```

gslcdfgumbel1P

Link to:

```
gsl_cdf_gumbel1_P(a, b, c)
```

gslcdfgumbel1Q

Link to:

```
gsl_cdf_gumbel1_Q(a, b, c)
```

gslcdfgumbel1Pinv

Link to:

```
gsl_cdf_gumbel1_Pinv(a, b, c)
```

gslcdfgumbel1Qinv

Link to:

```
gsl_cdf_gumbel1_Qinv(a, b, c)
```

gslcdfgumbel2P

Link to:

```
gsl_cdf_gumbel2_P(a, b, c)
```

gslcdfgumbel2Q

Link to:

```
gsl_cdf_gumbel2_Q(a, b, c)
```

gslcdfgumbel2Pinv

Link to:

```
gsl_cdf_gumbel2_Pinv(a, b, c)
```

gslcdfgumbel2Qinv

Link to:

```
gsl_cdf_gumbel2_Qinv(a, b, c)
```

gslcdfweibullP

Link to:

```
gsl_cdf_weibull_P(a, b, c)
```

gslcdfweibullQ

Link to:

```
gsl_cdf_weibull_Q(a, b, c)
```

gslcdfweibullPinv

Link to:

```
gsl_cdf_weibull_Pinv(a, b, c)
```

gslcdfweibullQinv

Link to:

```
gsl_cdf_weibull_Qinv(a, b, c)
```

gslcdfparetoP

Link to:

```
gsl_cdf_pareto_P(a, b, c)
```

gslcdfparetoQ

Link to:

```
gsl_cdf_pareto_Q(a, b, c)
```

gslcdfparetoPinv

Link to:

```
gsl_cdf_pareto_Pinv(a, b, c)
```

gslcdfparetoQinv

Link to:

```
gsl_cdf_pareto_Qinv(a, b, c)
```

gslcdflogisticP

Link to:

```
gsl_cdf_logistic_P(a, b)
```

gslcdflogisticQ

Link to:

```
gsl_cdf_logistic_Q(a, b)
```

gslcdflogisticPinv

Link to:

```
gsl_cdf_logistic_Pinv(a, b)
```

gslcdflogisticQinv

Link to:

```
gsl_cdf_logistic_Qinv(a, b)
```

gslcdfbinomialP

Link to:

```
gsl_cdf_binomial_P(a, b, c)
```

gslcdfbinomialQ

Link to:

```
gsl_cdf_binomial_Q(a, b, c)
```

gslcdfpoissonP

Link to:

```
gsl_cdf_poisson_P(a, b)
```

gslcdfpoissonQ

Link to:

```
gsl_cdf_poisson_Q(a, b)
```

gslcdfgeometricP

Link to:

```
gsl_cdf_geometric_P(a, b)
```

gslcdfgeometricQ

Link to:

```
gsl_cdf_geometric_Q(a, b)
```

gslcdfnegativebinomialP

Link to:

```
gsl_cdf_negative_binomial_P(a, b, c)
```

gslcdfnegativebinomialQ

Link to:

```
gsl_cdf_negative_binomial_Q(a, b, c)
```

gslcdfpascalP

Link to:

```
gsl_cdf_pascal_P(a, b, c)
```

gslcdfpascalQ

Link to:

```
gsl_cdf_pascal_Q(a, b, c)
```

gslranbernoullipdf

Link to:

```
gsl_ran_bernoulli_pdf(a, b)
```

gslranbeta

Link to:

```
gsl_ran_beta(a, b, c)
```

gslranbetapdf

Link to:

```
gsl_ran_beta_pdf(a, b, c)
```

gslranbinomialpdf

Link to:

```
gsl_ran_binomial_pdf(a, b, c)
```

gslranexponential

Link to:

```
gsl_ran_exponential(a, b)
```

gslranexponentialpdf

Link to:

```
gsl_ran_exponential_pdf(a, b)
```

gslranexppow

Link to:

```
gsl_ran_exppow(a, b, c)
```

gslranexppowpdf

Link to:

```
gsl_ran_exppow_pdf(a, b, c)
```

gslrancauchy

Link to:

```
gsl_ran_cauchy(a, b)
```

gslrancauchypdf

Link to:

```
gsl_ran_cauchy_pdf(a, b)
```

gslranchisq

Link to:

```
gsl_ran_chisq(a, b)
```

gslranchisqpdf

Link to:

```
gsl_ran_chisq_pdf(a, b)
```

gslranerlang

Link to:

```
gsl_ran_erlang(a, b, c)
```

gslranerlangpdf

Link to:

```
gsl_ran_erlang_pdf(a, b, c)
```

gslranfdist

Link to:

```
gsl_ran_fdist(a, b, c)
```

gslranfdistpdf

Link to:

```
gsl_ran_fdist_pdf(a, b, c)
```

gslranflat

Link to:

```
gsl_ran_flat(a, b, c)
```

gslranflatpdf

Link to:

```
gsl_ran_flat_pdf(a, b, c)
```

gslrangamma

Link to:

```
gsl_ran_gamma(a, b, c)
```

gslrangammaint

Link to:

```
gsl_ran_gamma_int(a, b, c)
```

gslrangammapdf

Link to:

```
gsl_ran_gamma_pdf(a, b, c)
```

gslrangammamt

Link to:

```
gsl_ran_gamma_mt(a, b, c)
```

gslrangammaknuth

Link to:

```
gsl_ran_gamma_knuth(a, b, c)
```

gslrangaussian

Link to:

```
gsl_ran_gaussian(a, b)
```

gslrangaussianratiomethod

Link to:

```
gsl_ran_gaussian_ratio_method(a, b)
```

gslrangaussianziggurat

Link to:

```
gsl_ran_gaussian_ziggurat(a, b)
```

gslrangaussianpdf

Link to:

```
gsl_ran_gaussian_pdf(a, b)
```

gslranugaussian

Link to:

```
gsl_ran_ugaussian(a)
```

gslranugaussianratiomethod

Link to:

```
gsl_ran_ugaussian_ratio_method(a)
```

gslranugaussianpdf

Link to:

```
gsl_ran_ugaussian_pdf(a)
```

gslrangaussiantail

Link to:

```
gsl_ran_gaussian_tail(a, b, c)
```

gslrangaussiantailpdf

Link to:

```
gsl_ran_gaussian_tail_pdf(a, b, c)
```

gslranugaussiantail

Link to:

```
gsl_ran_ugaussian_tail(a, b)
```

gslranugaussiantailpdf

Link to:

```
gsl_ran_ugaussian_tail_pdf(a, b)
```

gslranlandau

Link to:

```
gsl_ran_landau(a)
```

gslranlandaupdf

Link to:

```
gsl_ran_landau_pdf(a)
```

gslrangeometricpdf

Link to:

```
gsl_ran_geometric_pdf(a, b)
```

gslrangumbel1

Link to:

```
gsl_ran_gumbel1(a, b, c)
```

gslrangumbel1pdf

Link to:

```
gsl_ran_gumbel1_pdf(a, b, c)
```

gslrangumbel2

Link to:

```
gsl_ran_gumbel2(a, b, c)
```

gslrangumbel2pdf

Link to:

```
gsl_ran_gumbel2_pdf(a, b, c)
```

gslranlogistic

Link to:

```
gsl_ran_logistic(a, b)
```

gslranlogisticpdf

Link to:

```
gsl_ran_logistic_pdf(a, b)
```

gslranlognormal

Link to:

```
gsl_ran_lognormal(a, b, c)
```

gslranlognormalpdf

Link to:

```
gsl_ran_lognormal_pdf(a, b, c)
```

gslranlogarithmicpdf

Link to:

```
gsl_ran_logarithmic_pdf(a, b)
```

gslrannegativebinomialpdf

Link to:

```
gsl_ran_negative_binomial_pdf(a, b, c)
```

gslranpascalpdf

Link to:

```
gsl_ran_pascal_pdf(a, b, c)
```

gslranpareto

Link to:

```
gsl_ran_pareto(a, b, c)
```

gslranparetopdf

Link to:

```
gsl_ran_pareto_pdf(a, b, c)
```

gslranpoissonpdf

Link to:

```
gsl_ran_poisson_pdf(a, b)
```

gslranrayleigh

Link to:

```
gsl_ran_rayleigh(a, b)
```

gslranrayleighpdf

Link to:

```
gsl_ran_rayleigh_pdf(a, b)
```

gslranrayleightail

Link to:

```
gsl_ran_rayleigh_tail(a, b, c)
```

gslranrayleightailpdf

Link to:

```
gsl_ran_rayleigh_tail_pdf(a, b, c)
```

gslrantdist

Link to:

```
gsl_ran_tdsit(a, b)
```

gslrantdistpdf

Link to:

```
gsl_ran_tdsit_pdf(a, b)
```

gslranlaplace

Link to:

```
gsl_ran_laplace(a, b)
```

gslranlaplacepdf

Link to:

```
gsl_ran_laplace_pdf(a, b)
```

gslranlevy

Link to:

```
gsl_ran_levy(a, b, c)
```

gslranweibull

Link to:

```
gsl_ran_weibull(a, b, c)
```

gslranweibullpdf

Link to:

```
gsl_ran_weibull_pdf(a, b, c)
```

gslsfairyAi

Link to:

```
gsl_sf_airy_Ai(a, b)
```

gslsfairyBi

Link to:

```
gsl_sf_airy_Bi(a, b)
```

gslsfairyAiscaled

Link to:

```
gsl_sf_airy_Ai_scaled(a, b)
```

gslsfairyBiscaled

Link to:

```
gsl_sf_airy_Bi_scaled(a, b)
```

gslsfairyAiderivs

Link to:

```
gsl_sf_airy_Ai_deriv(a, b)
```

gslsfairyBiderivs

Link to:

```
gsl_sf_airy_Bi_deriv(a, b)
```

gslsfairyAiderivscaled

Link to:

```
gsl_sf_airy_Ai_deriv_scaled(a, b)
```

gslsfairyBiderivscaled

Link to:

```
gsl_sf_airy_Bi_deriv_scaled(a, b)
```

gslsfairyzeroAi

Link to:

```
gsl_sf_airy_Ai(a, b)
```

gslsfairyzeroBi

Link to:

```
gsl_sf_airy_aero_Bi(a)
```

gslsfairyzeroAiderivv

Link to:

```
gsl_sf_airy_aero_Ai_deriv(a)
```

gslsfairyzeroBiderivv

Link to:

```
gsl_sf_airy_aero_Bi_deriv(a)
```

gslsfbesselJ0

Link to:

```
gsl_sf_bessel_J0(a)
```

gslsfbesselJ1

Link to:

```
gsl_sf_bessel_J1(a)
```

gslsfbesselJn

Link to:

```
gsl_sf_bessel_Jn(a, b)
```

gslsfbesselY0

Link to:

```
gsl_sf_bessel_Y0(a)
```

gslsfbesselY1

Link to:

```
gsl_sf_bessel_Y1(a)
```

gslsfbesselYn

Link to:

```
gsl_sf_bessel_Yn(a, b)
```

gslsfbesselI0

Link to:

```
gsl_sf_bessel_I0(a)
```

gslsfbesselI1

Link to:

```
gsl_sf_bessel_I1(a)
```

gslsfbesselIn

Link to:

```
gsl_sf_bessel_In(a, b)
```

gslsfbesselI0scaled

Link to:

```
gsl_sf_bessel_I0_scaled(a)
```

gslsfbesselI1scaled

Link to:

```
gsl_sf_bessel_I1_scaled(a)
```

gslsfbesselInscaled

Link to:

```
gsl_sf_bessel_In_scaled(a, b)
```

gslsfbesselK0

Link to:

```
gsl_sf_bessel_K0(a)
```

gslsfbesselK1

Link to:

```
gsl_sf_bessel_K1(a)
```

gslsfbesselKn

Link to:

```
gsl_sf_bessel_Kn(a, b)
```

gslsfbesselK0scaled

Link to:

```
gsl_sf_bessel_K0_scaled(a)
```

gslsfbesselK1scaled

Link to:

```
gsl_sf_bessel_K1_scaled(a)
```

gslsfbesselKnscaled

Link to:

```
gsl_sf_bessel_Kn_scaled(a, b)
```

gslsfbesselj0

Link to:

```
gsl_sf_bessel_j0(a)
```

gslsfbesselj1

Link to:

```
gsl_sf_bessel_j1(a)
```

gslsfbesselj2

Link to:

```
gsl_sf_bessel_j2(a)
```

gslsfbesseljl

Link to:

```
gsl_sf_bessel_jl(a, b)
```

gslsfbessely0

Link to:

```
gsl_sf_bessel_y0(a)
```

gslsfbessely1

Link to:

```
gsl_sf_bessel_y0(a)
```

gslsfbessely2

Link to:

```
gsl_sf_bessel_y0(a)
```

gslsfbesselyl

Link to:

```
gsl_sf_bessel_jl(a, b)
```

gslsfbesseli0scaled

Link to:

```
gsl_sf_bessel_i0_scaled(a)
```

gslsfbesseli1scaled

Link to:

```
gsl_sf_bessel_i1_scaled(a)
```

gslsfbesseli2scaled

Link to:

```
gsl_sf_bessel_i2_scaled(a)
```

gslsfbesselilscaled

Link to:

```
gsl_sf_bessel_il_scaled(a, b)
```

gslsfbesselk0scaled

Link to:

```
gsl_sf_bessel_k0_scaled(a)
```

gslsfbesselk1scaled

Link to:

```
gsl_sf_bessel_k1_scaled(a)
```

gslsfbesselk2scaled

Link to:

```
gsl_sf_bessel_k2_scaled(a)
```

gslsfbesselklscaled

Link to:

```
gsl_sf_bessel_kl_scaled(a, b)
```

gslsfbesselJnu

Link to:

```
gsl_sf_bessel_Jnu(a, b)
```

gslsfbesselYnu

Link to:

```
gsl_sf_bessel_Ynu(a, b)
```

gslsfbesselInuscaled

Link to:

```
gsl_sf_bessel_Inu_scaled(a, b)
```

gslsfbesselInu

Link to:

```
gsl_sf_bessel_Inu(a, b)
```

gslsfbesselKnuscaled

Link to:

```
gsl_sf_bessel_Knu_scaled(a, b)
```

gslsfbesselKnu

Link to:

```
gsl_sf_bessel_Knu(a, b)
```

gslsfbessellnKnu

Link to:

```
gsl_sf_bessel_lnKnu(a, b)
```

gslsfbesselzeroJ0

Link to:

```
gsl_sf_bessel_zero_J0(a)
```

gslsfbesselzeroJ1

Link to:

```
gsl_sf_bessel_zero_J1(a)
```

gslsfbesselzeroJnu

Link to:

```
gsl_sf_bessel_zero_Jnu(a, b)
```

gslsfclausen

Link to:

```
gsl_sf_clausen(a)
```

gslsfhydrogenicR1

Link to:

```
gsl_sf_hydrogenicR_1(a, b)
```

gslsfdawson

Link to:

```
gsl_sf_dawnson(a)
```

gslsfdebye1

Link to:

```
gsl_sf_debye_1(a)
```

gslsfdebye2

Link to:

```
gsl_sf_debye_2(a)
```

gslsfdebye3

Link to:

```
gsl_sf_debye_3(a)
```

gslsfdebye4

Link to:

```
gsl_sf_debye_4(a)
```

gslsfdebye5

Link to:

```
gsl_sf_debye_5(a)
```

gslsfdebye6

Link to:

```
gsl_sf_debye_6(a)
```

gslsfdilog

Link to:

```
gsl_sf_dilog(a)
```

gslsfmultiply

Link to:

```
gsl_sf_multiply(a, b)
```

gslsfellintKcomp

Link to:

```
gsl_sf_ellint_Kcomp(a, b)
```

gslsfellintEcomp

Link to:

```
gsl_sf_ellint_Ecomp(a, b)
```

gslsfellintPcomp

Link to:

```
gsl_sf_ellint_Pcomp(a, b, c)
```

gslsfellintDcomp

Link to:

```
gsl_sf_ellint_Dcomp(a, b)
```

gslsfellintF

Link to:

```
gsl_sf_ellint_F(a, b, c)
```

gslsfellintE

Link to:

```
gsl_sf_ellint_E(a, b, c)
```

gslsfellintRC

Link to:

```
gsl_sf_ellint_RC(a, b, c)
```

gslsferfc

Link to:

```
gsl_sf_erfc(a)
```

gslsflogerfc

Link to:

```
gsl_sf_log_erfc(a)
```

gslsferf

Link to:

```
gsl_sf_erf(a)
```

gslsferfZ

Link to:

```
gsl_sf_erf_Z(a)
```

gslsferfQ

Link to:

```
gsl_sf_erf_Q(a)
```

gslsfhazard

Link to:

```
gsl_sf_hazard(a)
```

gslsfexp

Link to:

```
gsl_sf_exp(a)
```

gslsfexpmult

Link to:

```
gsl_sf_exp_mult(a, b)
```

gslsfexpm1

Link to:

```
gsl_sf_expm1(a)
```

gslsfexprel

Link to:

```
gsl_sf_exprel(a)
```

gslsfexprel2

Link to:

```
gsl_sf_exprel_2(a)
```

gslsfexpreln

Link to:

```
gsl_sf_exprel_n(a, b)
```

gslsfexpintE1

Link to:

```
gsl_sf_expint_E1(a)
```

gslsfexpintE2

Link to:

```
gsl_sf_expint_E2(a)
```

gslsfexpintEn

Link to:

```
gsl_sf_expint_En(a, b)
```

gslsfexpintE1scaled

Link to:

```
gsl_sf_expint_E1_scaled(a)
```

gslsfexpintE2scaled

Link to:

```
gsl_sf_expint_E1_scaled(a)
```

gslsfexpintEnscaled

Link to:

```
gsl_sf_expint_En_scaled(a, b)
```

gslsfexpintEi

Link to:

```
gsl_sf_expint_Ei(a)
```

gslsfexpintEiscaled

Link to:

```
gsl_sf_expint_Ei_scaled(a)
```

gslsfShi

Link to:

```
gsl_sf_Shi(a)
```

gslsfChi

Link to:

```
gsl_sf_Chi(a)
```

gslsfexpint3

Link to:

```
gsl_sf_expint_3(a)
```

gslsfSi

Link to:

```
gsl_sf_Si(a)
```

gslsfCi

Link to:

```
gsl_sf_Ci(a)
```

gslsfatanint

Link to:

```
gsl_sf_atanint(a)
```

gslsffermidiracm1

Link to:

```
gsl_sf_fermi_dirac_m1(a)
```

gslsffermidirac0

Link to:

```
gsl_sf_fermi_dirac_0(a)
```

gslsffermidirac1

Link to:

```
gsl_sf_fermi_dirac_1(a)
```

gslsffermidirac2

Link to:

```
gsl_sf_fermi_dirac_2(a)
```

gslsffermidiracint

Link to:

```
gsl_sf_fermi_dirac_int(a, b)
```

gslsffermidiracmhalf

Link to:

```
gsl_sf_fermi_dirac_mhalf(a)
```

gslsffermidirachalf

Link to:

```
gsl_sf_fermi_dirac_half(a)
```

gslsffermidirac3half

Link to:

```
gsl_sf_fermi_dirac_3half(a)
```

gslsffermidiracinc0

Link to:

```
gsl_sf_fermi_dirac_inc_0(a, b)
```

gslsflngamma

Link to:

```
gsl_sf_lngamma(a)
```

gslsfgamma

Link to:

```
gsl_sf_gamma(a)
```

gslsfgammastar

Link to:

```
gsl_sf_gammastar(a)
```

gslsfgammainv

Link to:

```
gsl_sf_gammainv(a)
```

gslsftaylorcoeff

Link to:

```
gsl_sf_taylorcoeff(a, b)
```

gslsffact

Link to:

```
gsl_sf_fact(a)
```

gslsfdoublefact

Link to:

```
gsl_sf_doublefact(a)
```

gslsflnfact

Link to:

```
gsl_sf_lnfact(a)
```

gslsflndoublefact

Link to:

```
gsl_sf_lndoublefact(a)
```

gslsflnchoose

Link to:

```
gsl_sf_lnchoose(a, b)
```

gslsfchoose

Link to:

```
gsl_sf_choose(a, b)
```

gslsflnpoch

Link to:

```
gsl_sf_lnpoch(a, b)
```

gslsfpoch

Link to:

```
gsl_sf_poch(a, b)
```

gslsfpochrel

Link to:

```
gsl_sf_pochrel(a, b)
```

gslsfgammaincQ

Link to:

```
gsl_sf_gamma_inc_Q(a, b)
```

gslsfgammaincP

Link to:

```
gsl_sf_gamma_inc_P(a, b)
```

gslsfgammainc

Link to:

```
gsl_sf_gamma_inc(a, b)
```

gslsflnbeta

Link to:

```
gsl_sf_lnbeta(a, b)
```

gslsfbeta

Link to:

```
gsl_sf_beta(a, b)
```

gslsfbetainc

Link to:

```
gsl_sf_betaçinc(a, b, c)
```

gslsfgegenpoly1

Link to:

```
gsl_sf_gegenpoly_1(a, b)
```

gslsfgegenpoly2

Link to:

```
gsl_sf_gegenpoly_2(a, b)
```

gslsfgegenpoly3

Link to:

```
gsl_sf_gegenpoly_3(a, b)
```

gslsfgegenpolyn

Link to:

```
gsl_sf_gegenpoly_n(a, b, c)
```

gslsfhyperg0F1

Link to:

```
gsl_sf_hyperg_0F1(a, b)
```

gslsfhyperg1F1int

Link to:

```
gsl_sf_hyperg_1F1_inc(a, b, c)
```

gslsfhyperg1F1

Link to:

```
gsl_sf_hyperg_1F1(a, b, c)
```

gslsfhypergUint

Link to:

```
gsl_sf_hyperg_U_inc(a, b, c)
```

gslsfhypergU

Link to:

```
gsl_sf_hyperg_U(a, b, c)
```

gslsfhyperg2F0

Link to:

```
gsl_sf_hyperg_U_2F0(a, b, c)
```

gslsflaguerre1

Link to:

```
gsl_sf_laguerre_1(a, b)
```

gslsflaguerre2

Link to:

```
gsl_sf_laguerre_2(a, b)
```

gslsflaguerre3

Link to:

```
gsl_sf_laguerre_3(a, b)
```

gslsflaguerren

Link to:

```
gsl_sf_laguerre_n(a, b, c)
```

gslsflambertW0

Link to:

```
gsl_sf_lambert_W0(a)
```

gslsflambertWm1

Link to:

```
gsl_sf_lambert_Wm1(a)
```

gslsflegendrePl

Link to:

```
gsl_sf_legendre_Pl(a, b)
```

gslsflegendreP1

Link to:

```
gsl_sf_legendre_P1(a)
```

gslsflegendreP2

Link to:

```
gsl_sf_legendre_P2(a)
```

gslsflegendreP3

Link to:

```
gsl_sf_legendre_P3(a)
```

gslsflegendreQ0

Link to:

```
gsl_sf_legendre_Q0(a)
```

gslsflegendreQ1

Link to:

```
gsl_sf_legendre_Q1(a)
```

gslsflegendreQl

Link to:

```
gsl_sf_legendre_Ql(a, b)
```

gslsflegendrePlm

Link to:

```
gsl_sf_legendre_Plm(a, b, c)
```

gslsflegendresphPlm

Link to:

```
gsl_sf_legendre_sphP1m(a, b, c)
```

gslsflegendrearraysize

Link to:

```
gsl_sf_legendre_array_size(a, b)
```

gslsfconicalPhalf

Link to:

```
gsl_sf_conicalP_half(a, b)
```

gslsfconicalPmhalf

Link to:

```
gsl_sf_conicalP_mhalf(a, b)
```

gslsfconicalP0

Link to:

```
gsl_sf_conicalP_0(a, b)
```

gslsfconicalP1

Link to:

```
gsl_sf_conicalP_1(a, b)
```

gslsfconicalPsphreg

Link to:

```
gsl_sf_conicalP_sph_reg(a, b, c)
```

gslsfconicalPcylreg

Link to:

```
gsl_sf_conicalP_cyl_reg(a, b, c)
```

gslsflegendreH3d0

Link to:

```
gsl_sf_legendre_H3d_0(a, b)
```

gslsflegendreH3d1

Link to:

```
gsl_sf_legendre_H3d_1(a, b)
```

gslsflegendreH3d

Link to:

```
gsl_sf_legendre_H3d(a, b, c)
```

gslsflog

Link to:

```
gsl_sf_log(a)
```

gslsflogabs

Link to:

```
gsl_sf_log_abs(a)
```

gslsflog1plusx

Link to:

```
gsl_sf_log_1plusx(a)
```

gslsflog1plusxmx

Link to:

```
gsl_sf_log_1plusx_mx(a)
```

gslsfpowint

Link to:

```
gsl_sf_pow_int(a, b)
```

gslsfpsiint

Link to:

```
gsl_sf_psi_int(a)
```

gslsfpsi

Link to:

```
gsl_sf_psi(a)
```

gslsfpsi1piy

Link to:

```
gsl_sf_psi_1piy(a)
```

gslsfpsi1int

Link to:

```
gsl_sf_psi_1_int(a)
```

gslsfpsi1

Link to:

```
gsl_sf_psi_1(a)
```

gslsfpsin

Link to:

```
gsl_sf_psi_n(a, b)
```

gslsfsynchrotron1

Link to:

```
gsl_sf_synchrotron_1(a)
```

gslsfsynchrotron2

Link to:

```
gsl_sf_synchrotron_2(a)
```

gslsftransport2

Link to:

```
gsl_sf_transport_2(a)
```

gslsftransport3

Link to:

```
gsl_sf_transport_3(a)
```

gslsftransport4

Link to:

```
gsl_sf_transport_4(a)
```

gslsftransport5

Link to:

```
gsl_sf_transport_5(a)
```

gslsfsin

Link to:

```
gsl_sf_sin(a)
```

gslsfcos

Link to:

```
gsl_sf_cos(a)
```

gslsfhypot

Link to:

```
gsl_sf_hypot(a, b)
```

gslsfsinc

Link to:

```
gsl_sf_sinc(a)
```

gslsflnsinh

Link to:

```
gsl_sf_lnsinh(a)
```

gslsflncosh

Link to:

```
gsl_sf_lncosh(a)
```

gslsfanglerestrictsymm

Link to:

```
gsl_sf_andle_restrict_symm(a)
```

gslsfanglerestrictpos

Link to:

```
gsl_sf_angle_restrict_pos(a)
```

gslsfzetaint

Link to:

```
gsl_sf_zeta_int(a)
```

gslsfzeta

Link to:

```
gsl_sf_zeta(a)
```

gslsfzetam1

Link to:

```
gsl_sf_zetam1(a)
```

gslsfzetam1int

Link to:

```
gsl_sf_zetam1_int(a)
```

gslsfhzeta

Link to:

```
gsl_sf_hzeta(a, b)
```

gslsfetaint

Link to:

```
gsl_sf_eta_int(a)
```

gslsfeta

Link to:

```
gsl_sf_eta(a)
```

4.8.15 ff-Ipopt

Refer to the Ipopt documentation for more informations.

IPOPT

Todo: todo

4.8.16 fflapack

Refer to the LAPACK documentation for more informations.

inv

Todo: todo

dgeev

Todo: todo

zgeev

Todo: todo

geev

Todo: todo

geev

Todo: todo

dggev

Todo: todo

zggev

Todo: todo

dsygvd

Todo: todo

dgesdd

Todo: todo

zhegv

Todo: todo

dsyev

Todo: todo

zheev

Todo: todo

4.8.17 ff-mmap-semaphore

Wait

Todo: todo

trywait

Todo: todo

Post

Todo: todo

msync

Todo: todo

Read

Todo: todo

Write

Todo: todo

4.8.18 ffnewuoa

newuoa

Todo: todo

4.8.19 ff-NLopt

Refer to the NLOPT documentation for more informations.

nloptDIRECT

Todo: todo

nloptDIRECTL

Todo: todo

nloptDIRECTLRand

Todo: todo

nloptDIRECTScal

Todo: todo

nloptDIRECTNoScal

Todo: todo

nloptDIRECTLNoScal

Todo: todo

nloptDIRECTLRandNoScal

Todo: todo

nloptOrigDIRECT

Todo: todo

nloptOrigDIRECTL

Todo: todo

nloptStoGO

Todo: todo

nloptStoGORand

Todo: todo

nloptLBFGS

Todo: todo

nloptPRAXIS

Todo: todo

nloptVar1

Todo: todo

nloptVar2

Todo: todo

nloptTNewton

Todo: todo

nloptTNewtonRestart

Todo: todo

nloptTNewtonPrecond

Todo: todo

nloptTNewtonPrecondRestart

Todo: todo

nloptCRS2

Todo: todo

nloptMMA

Todo: todo

nloptCOBYLA

Todo: todo

nloptNEWUOA

Todo: todo

nloptNEWUOABound

Todo: todo

nloptNelderMead

Todo: todo

nloptSbplx

Todo: todo

nloptBOBYQA

Todo: todo

nloptISRES

Todo: todo

nloptSLSQP

Todo: todo

nloptMLSL

Todo: todo

nloptMLSLLDS

Todo: todo

nloptAUGLAG

Todo: todo

nloptAUGLAGEQ

Todo: todo

4.8.20 ffrandom

srandomdev

Todo: todo

srandom

Todo: todo

random

Todo: todo

4.8.21 FreeFemQA

MeshGenQA

Todo: todo

4.8.22 freeyams

freeyams

Todo: todo

4.8.23 gmsh

Need

```
load "gsmh"
```

The `gmsh` software is available here

gmshload

Load a 2D mesh build with Gmsh.

```
mesh Th = gmshload(MeshFile, [reftri=RefTri], [renum=Renum]);
```

Parameters:

- `MeshFile` (`string`) Mesh file name
- `reftri=` (.. todo:: todo)
- `renum=` (.. todo:: todo)

Output:

- `Th` (`mesh`)

gmshload3

Load a 3D mesh build with Gmsh.

```
mesh3 Th = gmshload(MeshFile, [reftet=RefTet], [renum=Renum]);
```

Parameters:

- `MeshFile` (`string`) Mesh file name
- `reftet=` (.. todo:: todo)
- `renum=` (.. todo:: todo)

Output:

- `Th` (`mesh3`)

savegmsh

Todo: todo

4.8.24 gsl

gslpolysolvequadratic

Todo: todo

gslpolysolvecubic

Todo: todo

gslpolycomplexsolve

Todo: todo

gslrnguniform

Todo: todo

gslrnguniformpos

Todo: todo

gslname

Todo: todo

gslrngget

Todo: todo

gslrngmin

Todo: todo

gslrngmax

Todo: todo

gslrngset

Todo: todo

gslrngtype

Todo: todo

4.8.25 ilut

applyIlutPrecond

Todo: todo

makeIlutPrecond

Todo: todo

4.8.26 iohdf5

savehdf5sol

Todo: todo

4.8.27 iovtk

savevtk

Save mesh or solution in vtk/vtu format.

```
savevtk(FileName, Th, [Ux, Uy, Uz], p, [dataname=DataName],
↪[withsurfacemesh=WithSurfaceMesh], [order=Order], [floatmesh=FloatMesh],
↪[floatsol=FloatSol], [bin=Bin], [swap=Swap]);
```

Parameters:

- `FileName` (`string`) File name: `*.vtk` or `*.vtu`
- `Th` (`mesh` or `mesh3`)
- `[Ux, Uy, Uz], p` (`fespace` function of vector of `fespace` functions) Solutions to save, as much as wanted
- `dataname=` (`string`) Name of solutions, seprated by a space
- `withsurfacemesh=` (`bool`) .. todo:: todo
- `order=` (`int[int]`) Order of solutions.

  Available: 0 or 1
- `floatmesh=` (`bool`) .. todo:: todo
- `floatsol=` (`bool`) .. todo:: todo
- `bin=` (`bool`) If true, save file in binary format
- `swap` (`bool`) .. todo:: todo

Output:

- None

vtkload

Todo: todo

vtkload3

Todo: todo

4.8.28 isoline

Need

```
load "isoline"
```

isoline

```
int N = isoline(Th, u, xy, iso=Iso, close=Close, smoothing=Smoothing, ratio=Ratio,
↪eps=Eps, beginend=BeginEnd, file=File);
```

Todo: todo

Curve

Todo: todo

Area

Todo: todo

findallocalmin

Todo: todo

4.8.29 lapack

inv

Todo: todo

dgeev

Todo: todo

zgeev

Todo: todo

geev

Todo: todo

dggev

Todo: todo

zggev

Todo: todo

dsygvd

Todo: todo

dgesdd

Todo: todo

zhegv

Todo: todo

dsyev

Todo: todo

zheev

Todo: todo

dgelsy

Todo: todo

4.8.30 lgbmo

bmo

Todo: todo

4.8.31 mat_dervieux

MatUpWind1

Todo: todo

4.8.32 mat_psi

MatUpWind0

Todo: todo

4.8.33 medit

medit

Todo: todo

savesol

Todo: todo

readsol

Todo: todo

4.8.34 metis

metisnodal

Todo: todo

metisdual

Todo: todo

4.8.35 MetricKuate

MetricKuate

Todo: todo

4.8.36 MetricPk

MetricPk

Todo: todo

4.8.37 mmg3d

mmg3d

Todo: todo

4.8.38 mmg3d-v4.0

mmg3d

Todo: todo

4.8.39 msh3

change

Todo: todo

movemesh23

Todo: todo

movemesh2D3Dsurf

Todo: todo

movemesh3

Todo: todo

movemesh

Todo: todo

movemesh3D

Todo: todo

deplacement

Todo: todo

checkbemesh

Todo: todo

buildlayers

Todo: todo

bcube

Todo: todo

cube

Construct a cubic mesh.

```
mesh3 Th = cube(nnX, nnY, nnZ, [X(x), Y(y), Z(z)], [label=Label], [flags=Flags],
↪[region=Region]);
```

Parameters:

- `nnX` (`int`) Number of discretization point along x
- `nnY` (`int`) Number of discretization point along y
- `nnZ` (`int`) Number of discretization point along z
- `X(x)` (`func`) *[Optional]* Affine function of x to define the length Default: `x`
- `Y(y)` (`func`) *[Optional]* Affine function of y to define the width Default: `y`
- `Z(z)` (`func`) *[Optional]* Affine function of z to define the height Default: `z`
- `label=` (`int[int]`) *[Optional]*

  List of surface labels Default: `[1, 2, 3, 4, 5, 6]`
- `flags=` (`int`) *[Optional]*

  Refer to *square*
- `region=` (`int`) *[Optional]*

  Region number of the cube volume Default: `0`

Output:

- `Th` (`mesh3`) Cube mesh

trunc

Todo: todo

gluemesh

Todo: todo

extract

Todo: todo

showborder

Todo: todo

getborder

Todo: todo

AddLayers

Todo: todo

4.8.40 mshmet

mshmet

Todo: todo

4.8.41 MUMPS

defaulttoMUMPSseq

Todo: todo

4.8.42 MUMPS_seq

defaulttoMUMPSseq

Todo: todo

4.8.43 netgen

netg

Todo: todo

netgstl

Todo: todo

netgload

Todo: todo

4.8.44 NewSolver

defaulttoUMFPACK

Todo: todo

4.8.45 PARDISO

defaulttoPARDISO

Todo: todo

ompsetnumthreads

Todo: todo

ompgetnumthreads

Todo: todo

ompgetmaxthreads

Todo: todo

4.8.46 pcm2rnm

readpcm

Todo: todo

4.8.47 pipe

flush

Todo: todo

sleep

Todo: todo

usleep

Todo: todo

4.8.48 qf11to25

QF1d

Todo: todo

QF2d

Todo: todo

QF3d

Todo: todo

tripleQF

4.8.49 scotch

scotch

Todo: todo

4.8.50 shell

readdir

Todo: todo

unlink

Todo: todo

rmdir

Todo: todo

cddir

Todo: todo

chdir

Todo: todo

basename

Todo: todo

dirname

Todo: todo

mkdir

Todo: todo

chmod

Todo: todo

cpfile

Todo: todo

stat

Todo: todo

isdir

Todo: todo

getenv

Todo: todo

setenv

Todo: todo

unsetenv

Todo: todo

4.8.51 splitedges

SplitedgeMesh

Todo: todo

4.8.52 splitmesh12

splitmesh12

Todo: todo

4.8.53 splitmesh3

splitmesh3

Todo: todo

4.8.54 splitmesh4

splimesh4

Todo: todo

4.8.55 splitmesh6

splitmesh6

Todo: todo

4.8.56 SuperLu

defaulttoSuperLu

Todo: todo

4.8.57 symmetrizeCSR

symmetrizeCSR

Todo: todo

4.8.58 tetgen

Refer to the Tetgen documentation for more informations.

tetgconvexhull

Todo: todo

tetgtransfo

Todo: todo

tetg

Build a 3D mesh from a surface.

```
mesh3 Th = tetg(Th0, [reftet=RefTet], [label=Label], [switch=Switch],
↪[nbofholes=NbOfHoles], [holelist=HoleList], [nbofregions=NbOfRegions],
↪[regionlist=RegionList], [nboffacetcl=NbOfFaceTcl], [facetcl=FaceTcl])
```

Todo: todo

tetgreconstruction

Todo: todo

4.8.59 UMFPACK64

defaulttoUMFPACK64

Todo: todo

4.8.60 VTK_writer_3d

Vtkaddmesh

Todo: todo

Vtkaddscalar

Todo: todo

4.8.61 VTK_writer

Vtkaddmesh

Todo: todo

Vtkaddscalar

CHAPTER

FIVE

MATHEMATICAL MODELS

Summary:

This chapter goes deeper into a number of problems that **FreeFEM** can solve. It is a complement to the *Tutorial part* which was only an introduction.

Users are invited to contribute to make this models database grow.

5.1 Static problems

5.1.1 Soap Film

Our starting point here will be the mathematical model to find the shape of **soap film** which is glued to the ring on the $xy-$plane:

$$C = \{(x, y);\ x = \cos t,\ y = \sin t,\ 0 \leq t \leq 2\pi\}$$

We assume the shape of the film is described by the graph $(x, y, u(x, y))$ of the vertical displacement $u(x, y)$ $(x^2+y^2 < 1)$ under a vertical pressure p in terms of force per unit area and an initial tension μ in terms of force per unit length.

Consider the "small plane" ABCD, A:$(x, y, u(x, y))$, B:$(x, y, u(x + \delta x, y))$, C:$(x, y, u(x + \delta x, y + \delta y))$ and D:$(x, y, u(x, y + \delta y))$.

Denote by $\vec{n}(x, y) = (n_x(x, y), n_y(x, y), n_z(x, y))$ the normal vector of the surface $z = u(x, y)$. We see that the vertical force due to the tension μ acting along the edge AD is $-\mu n_x(x, y)\delta y$ and the the vertical force acting along the edge AD is:

$$\mu n_x(x + \delta x, y)\delta y \simeq \mu \left(n_x(x, y) + \frac{\partial n_x}{\partial x}\delta x \right) (x, y)\delta y$$

Similarly, for the edges AB and DC we have:

$$-\mu n_y(x,y)\delta x, \quad \mu\left(n_y(x,y)+\partial n_y/\partial y\right)(x,y)\delta x$$

The force in the vertical direction on the surface ABCD due to the tension μ is given by:

$$\mu\left(\partial n_x/\partial x\right)\delta x\delta y + T\left(\partial n_y/\partial y\right)\delta y\delta x$$

Assuming small displacements, we have:

$$\begin{array}{rcl} \nu_x & = & (\partial u/\partial x)/\sqrt{1+(\partial u/\partial x)^2+(\partial u/\partial y)^2} \simeq \partial u/\partial x, \\ \nu_y & = & (\partial u/\partial y)/\sqrt{1+(\partial u/\partial x)^2+(\partial u/\partial y)^2} \simeq \partial u/\partial y \end{array}$$

Letting $\delta x \to dx$, $\delta y \to dy$, we have the equilibrium of the vertical displacement of soap film on ABCD by p:

$$\mu dxdy\partial^2 u/\partial x^2 + \mu dxdy\partial^2 u/\partial y^2 + pdxdy = 0$$

Using the Laplace operator $\Delta = \partial^2/\partial x^2 + \partial^2/\partial y^2$, we can find the virtual displacement write the following:

$$-\Delta u = f \quad \text{in } \Omega$$

where $f = p/\mu$, $\Omega = \{(x,y);\ x^2+y^2<1\}$.

Poisson's equation appears also in **electrostatics** taking the form of $f = \rho/\epsilon$ where ρ is the charge density, ϵ the dielectric constant and u is named as electrostatic potential.

The soap film is glued to the ring $\partial\Omega = C$, then we have the boundary condition:

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

If the force is gravity, for simplify, we assume that $f = -1$.

```
// Parameters
int nn = 50;
func f = -1;
func ue = (x^2+y^2-1)/4; //ue: exact solution

// Mesh
border a(t=0, 2*pi){x=cos(t); y=sin(t); label=1;}
mesh disk = buildmesh(a(nn));
plot(disk);

// Fespace
fespace femp1(disk, P1);
femp1 u, v;

// Problem
problem laplace (u, v)
    = int2d(disk)( //bilinear form
        dx(u)*dx(v)
        + dy(u)*dy(v)
    )
    - int2d(disk)( //linear form
        f*v
    )
    + on(1, u=0) //boundary condition
    ;

```

(continues on next page)

(continued from previous page)

```
// Solve
laplace;

// Plot
plot (u, value=true, wait=true);

// Error
femp1 err = u - ue;
plot(err, value=true, wait=true);

cout << "error L2 = " << sqrt( int2d(disk)(err^2) )<< endl;
cout << "error H10 = " << sqrt( int2d(disk)((dx(u)-x/2)^2) + int2d(disk)((dy(u)-y/2)^
↪2) )<< endl;

/// Re-run with a mesh adaptation ///

// Mesh adaptation
disk = adaptmesh(disk, u, err=0.01);
plot(disk, wait=true);

// Solve
laplace;
plot (u, value=true, wait=true);

// Error
err = u - ue; //become FE-function on adapted mesh
plot(err, value=true, wait=true);

cout << "error L2 = " << sqrt( int2d(disk)(err^2) )<< endl;
cout << "error H10 = " << sqrt( int2d(disk)((dx(u)-x/2)^2) + int2d(disk)((dy(u)-y/2)^
↪2) )<< endl;
```

Fig. 5.1: Isovalue of u

Fig. 5.2: A side view of u

In the 37th line, the L^2-error estimation between the exact solution u_e,

$$\|u_h - u_e\|_{0,\Omega} = \left(\int_\Omega |u_h - u_e|^2 \, textdxtextdy\right)^{1/2}$$

and in the following line, the H^1-error seminorm estimation:

$$|u_h - u_e|_{1,\Omega} = \left(\int_\Omega |\nabla u_h - \nabla u_e|^2 \, textdxtextdy\right)^{1/2}$$

are done on the initial mesh. The results are $\|u_h - u_e\|_{0,\Omega} = 0.000384045$, $|u_h - u_e|_{1,\Omega} = 0.0375506$.

After the adaptation, we have $\|u_h - u_e\|_{0,\Omega} = 0.000109043$, $|u_h - u_e|_{1,\Omega} = 0.0188411$. So the numerical solution is improved by adaptation of mesh.

5.1.2 Electrostatics

We assume that there is no current and a time independent charge distribution. Then the electric field $\mathbf{E}$ satisfies:

$$\begin{array}{rcl} \mathrm{div}\mathbf{E} & = & \rho/\epsilon \\ \mathrm{curl}\mathbf{E} & = & 0 \end{array} \tag{5.1}$$

where ρ is the charge density and ϵ is called the permittivity of free space.

From the equation (5.1) We can introduce the electrostatic potential such that $\mathbf{E} = -\nabla\phi$. Then we have Poisson's equation $-\Delta\phi = f$, $f = -\rho/\epsilon$.

We now obtain the equipotential line which is the level curve of ϕ, when there are no charges except conductors $\{C_i\}_{1,\cdots,K}$. Let us assume K conductors $C_1, \cdots, C_K$ within an enclosure C_0.

Each one is held at an electrostatic potential φ_i. We assume that the enclosure $C0$ is held at potential 0. In order to know $\varphi(x)$ at any point x of the domain Ω, we must solve:

$$-\Delta\varphi = 0 \quad \text{in } \Omega$$

where Ω is the interior of C_0 minus the conductors C_i, and Γ is the boundary of Ω, that is $\sum_{i=0}^N C_i$.

Here g is any function of x equal to φ_i on C_i and to 0 on C_0. The boundary equation is a reduced form for:

$$\varphi = \varphi_i \text{ on } C_i,\ i = 1...N, \varphi = 0 \text{ on } C_0.$$

First we give the geometrical informations; $C_0 = \{(x,y);\ x^2 + y^2 = 5^2\}$, $C_1 = \{(x,y):\ \frac{1}{0.3^2}(x-2)^2 + \frac{1}{3^2}y^2 = 1\}$, $C_2 = \{(x,y):\ \frac{1}{0.3^2}(x+2)^2 + \frac{1}{3^2}y^2 = 1\}$.

Let Ω be the disk enclosed by C_0 with the elliptical holes enclosed by C_1 and C_2. Note that C_0 is described counterclockwise, whereas the elliptical holes are described clockwise, because the boundary must be oriented so that the computational domain is to its left.

(a) Disk with two elliptical holes

(b) Equipotential lines where C_1 is located in right hand side

```
// Mesh
border C0(t=0, 2*pi){x=5*cos(t); y=5*sin(t);}
border C1(t=0, 2*pi){x=2+0.3*cos(t); y=3*sin(t);}
border C2(t=0, 2*pi){x=-2+0.3*cos(t); y=3*sin(t);}

mesh Th = buildmesh(C0(60) + C1(-50) + C2(-50));
plot(Th);

// Fespace
fespace Vh(Th, P1);
Vh uh, vh;

// Problem
problem Electro (uh, vh)
    = int2d(Th)( //bilinear
        dx(uh)*dx(vh)
        + dy(uh)*dy(vh)
    )
    + on(C0, uh=0) //boundary condition on C_0
    + on(C1, uh=1) //+1 volt on C_1
    + on(C2, uh=-1) //-1 volt on C_2
    ;

// Solve
Electro;
plot(uh);
```

5.1.3 Aerodynamics

Let us consider a wing profile S in a uniform flow. Infinity will be represented by a large circle Γ_∞. As previously, we must solve:

$$\Delta\varphi = 0 \quad \text{in } \Omega, \quad \varphi|_S = c, \quad \varphi|_{\Gamma_\infty} = u_{\infty 1x} - u_{\infty 2x} \tag{5.2}$$

where Ω is the area occupied by the fluid, u_∞ is the air speed at infinity, c is a constant to be determined so that $\partial_n\varphi$ is continuous at the trailing edge P of S (so-called Kutta-Joukowski condition). Lift is proportional to c.

To find c we use a superposition method. As all equations in (5.2) are linear, the solution φ_c is a linear function of c

$$\varphi_c = \varphi_0 + c\varphi_1$$

where φ_0 is a solution of (5.2) with $c = 0$ and φ_1 is a solution with $c = 1$ and zero speed at infinity.

With these two fields computed, we shall determine c by requiring the continuity of $\partial\varphi/\partial n$ at the trailing edge. An equation for the upper surface of a NACA0012 (this is a classical wing profile in aerodynamics; the rear of the wing is called the trailing edge) is:

$$y = 0.17735\sqrt{x} - 0.075597x - 0.212836x^2 + 0.17363x^3 - 0.06254x^4$$

Taking an incidence angle α such that $\tan\alpha = 0.1$, we must solve:

$$-\Delta\varphi = 0 \qquad \text{in } \Omega, \quad \varphi|_{\Gamma_1} = y - 0.1x, \quad \varphi|_{\Gamma_2} = c$$

where Γ_2 is the wing profile and Γ_1 is an approximation of infinity. One finds c by solving:

$$\begin{array}{llll} -\Delta\varphi_0 & = 0 \quad \text{in } \Omega \quad , & \varphi_0|_{\Gamma_1} & = y - 0.1x \quad , \quad \varphi_0|_{\Gamma_2} = 0, \\ -\Delta\varphi_1 & = 0 \quad \text{in } \Omega \quad , & \varphi_1|_{\Gamma_1} & = 0 \quad , \quad \varphi_1|_{\Gamma_2} = 1 \end{array}$$

The solution $\varphi = \varphi_0 + c\varphi_1$ allows us to find c by writing that $\partial_n\varphi$ has no jump at the trailing edge $P = (1, 0)$.

We have $\partial n\varphi - (\varphi(P^+) - \varphi(P))/\delta$ where P^+ is the point just above P in the direction normal to the profile at a distance δ. Thus the jump of $\partial_n\varphi$ is $(\varphi_0|_{P^+} + c(\varphi_1|_{P^+} - 1)) + (\varphi_0|_{P^-} + c(\varphi_1|_{P^-} - 1))$ divided by δ because the normal changes sign between the lower and upper surfaces. Thus

$$c = -\frac{\varphi_0|_{P^+} + \varphi_0|_{P^-}}{(\varphi_1|_{P^+} + \varphi_1|_{P^-} - 2)},$$

which can be programmed as:

$$c = -\frac{\varphi_0(0.99, 0.01) + \varphi_0(0.99, -0.01)}{(\varphi_1(0.99, 0.01) + \varphi_1(0.99, -0.01) - 2)}.$$

```
// Mesh
border a(t=0, 2*pi){x=5*cos(t); y=5*sin(t);}
border upper(t=0, 1) {
    x=t;
    y=0.17735*sqrt(t)-0.075597*t - 0.212836*(t^2) + 0.17363*(t^3) - 0.06254*(t^4);
}
border lower(t=1, 0) {
    x=t;
    y=-(0.17735*sqrt(t) - 0.075597*t - 0.212836*(t^2) + 0.17363*(t^3) - 0.06254*(t^
↪4));
}
border c(t=0, 2*pi){x=0.8*cos(t)+0.5; y=0.8*sin(t);}

mesh Zoom = buildmesh(c(30) + upper(35) + lower(35));
mesh Th = buildmesh(a(30) + upper(35) + lower(35));

// Fespace
fespace Vh(Th, P2);
Vh psi0, psi1, vh;

fespace ZVh(Zoom, P2);

```

(continues on next page)

(a) Isovalue of $cp = -(\partial_x \psi)^2 - (\partial_y \psi)^2$

(b) Zooming of cp

(continued from previous page)

```
// Problem
solve Joukowski0(psi0, vh)
    = int2d(Th)(
        dx(psi0)*dx(vh)
        + dy(psi0)*dy(vh)
    )
    + on(a, psi0=y-0.1*x)
    + on(upper, lower, psi0=0)
    ;

plot(psi0);

solve Joukowski1(psi1,vh)
    = int2d(Th)(
        dx(psi1)*dx(vh)
        + dy(psi1)*dy(vh)
    )
    + on(a, psi1=0)
    + on(upper, lower, psi1=1);

plot(psi1);

//continuity of pressure at trailing edge
real beta = psi0(0.99,0.01) + psi0(0.99,-0.01);
beta = -beta / (psi1(0.99,0.01) + psi1(0.99,-0.01)-2);

Vh psi = beta*psi1 + psi0;
plot(psi);

ZVh Zpsi = psi;
plot(Zpsi, bw=true);

ZVh cp = -dx(psi)^2 - dy(psi)^2;
plot(cp);

ZVh Zcp = cp;
plot(Zcp, nbiso=40);
```

5.1.4 Error estimation

There are famous estimation between the numerical result u_h and the exact solution u of the *Poisson's problem*:

If triangulations $\{\mathcal{T}_h\}_{h\downarrow 0}$ is regular (see *Regular Triangulation*), then we have the estimates:

$$\begin{array}{rcl} |\nabla u - \nabla u_h|_{0,\Omega} & \leq & C_1 h \\ \|u - u_h\|_{0,\Omega} & \leq & C_2 h^2 \end{array} \tag{5.3}$$

with constants $C_1,\ C_2$ independent of h, if u is in $H^2(\Omega)$. It is known that $u \in H^2(\Omega)$ if Ω is convex.

In this section we check (5.3). We will pick up numericall error if we use the numerical derivative, so we will use the following for (5.3).

$$\begin{array}{rcl} \int_\Omega |\nabla u - \nabla u_h|^2 \, textdxtextdy & = & \int_\Omega \nabla u \cdot \nabla (u - 2u_h) \, textdxtextdy + \int_\Omega \nabla u_h \cdot \nabla u_h \, textdxtextdy \\ & = & \int_\Omega f(u - 2u_h) \, textdxtextdy + \int_\Omega f u_h \, textdxtextdy \end{array}$$

The constants $C_1,\ C_2$ are depend on $\mathcal{T}_h$ and f, so we will find them by **FreeFEM**.

In general, we cannot get the solution u as a elementary functions even if spetical functions are added. Instead of the exact solution, here we use the approximate solution u_0 in $V_h(\mathcal{T}_h, P_2),\ h \sim 0$.

```
// Parameters
func f = x*y;

//Mesh
mesh Th0 = square(100, 100);

// Fespace
fespace V0h(Th0, P2);
V0h u0, v0;

// Problem
solve Poisson0 (u0, v0)
    = int2d(Th0)(
        dx(u0)*dx(v0)
        + dy(u0)*dy(v0)
    )
    - int2d(Th0)(
        f*v0
    )
    + on(1, 2, 3, 4, u0=0)
    ;
plot(u0);

// Error loop
real[int] errL2(10), errH1(10);
for (int i = 1; i <= 10; i++){
    // Mesh
    mesh Th = square(5+i*3,5+i*3);

    // Fespace
    fespace Vh(Th, P1);
    Vh u, v;
    fespace Ph(Th, P0);
    Ph h = hTriangle; //get the size of all triangles

    // Problem
    solve Poisson (u, v)
```

(continues on next page)

(continued from previous page)

```
        = int2d(Th)(
            dx(u)*dx(v)
            + dy(u)*dy(v)
        )
        - int2d(Th)(
            f*v
        )
        + on(1, 2, 3, 4, u=0)
        ;

    // Error
    V0h uu = u; //interpolate solution on first mesh
    errL2[i-1] = sqrt( int2d(Th0)((uu - u0)^2) )/h[].max^2;
    errH1[i-1] = sqrt( int2d(Th0)(f*(u0 - 2*uu + uu)) )/h[].max;
}

// Display
cout << "C1 = " << errL2.max << "("<<errL2.min<<")" << endl;
cout << "C2 = " << errH1.max << "("<<errH1.min<<")" << endl;
```

We can guess that $C_1 = 0.0179253(0.0173266)$ and $C_2 = 0.0729566(0.0707543)$, where the numbers inside the parentheses are minimum in calculation.

5.1.5 Periodic Boundary Conditions

We now solve the Poisson equation:

$$-\Delta u = sin(x + \pi/4.) * cos(y + \pi/4.)$$

on the square $]0, 2\pi[^2$ under bi-periodic boundary condition $u(0, y) = u(2\pi, y)$ for all y and $u(x, 0) = u(x, 2\pi)$ for all x.

These boundary conditions are achieved from the definition of the periodic finite element space.

```
// Parameters
func f = sin(x+pi/4.)*cos(y+pi/4.); //right hand side

// Mesh
mesh Th = square(10, 10, [2*x*pi, 2*y*pi]);

// Fespace
//defined the fespace with periodic condition
//label: 2 and 4 are left and right side with y the curve abscissa
//       1 and 2 are bottom and upper side with x the curve abscissa
fespace Vh(Th, P2, periodic=[[2, y], [4, y], [1, x], [3, x]]);
Vh uh, vh;

// Problem
problem laplace (uh, vh)
    = int2d(Th)(
        dx(uh)*dx(vh)
        + dy(uh)*dy(vh)
    )
    + int2d(Th)(
        - f*vh
```

(continues on next page)

(continued from previous page)

```
    )
    ;

// Solve
laplace;

// Plot
plot(uh, value=true);
```

Fig. 5.5: The isovalue of solution u with periodic boundary condition

The periodic condition does not necessarily require parallel boundaries. The following example give such example.

Tip: Periodic boundary conditions - non-parallel boundaries

```
// Parameters
int n = 10;
real r = 0.25;
real r2 = 1.732;
func f = (y+x+1)*(y+x-1)*(y-x+1)*(y-x-1);

// Mesh
border a(t=0, 1){x=-t+1; y=t; label=1;};
border b(t=0, 1){x=-t; y=1-t; label=2;};
border c(t=0, 1){x=t-1; y=-t; label=3;};
border d(t=0, 1){x=t; y=-1+t; label=4;};
border e(t=0, 2*pi){x=r*cos(t); y=-r*sin(t); label=0;};
mesh Th = buildmesh(a(n) + b(n) + c(n) + d(n) + e(n));
plot(Th, wait=true);

// Fespace
//warning for periodic condition:
//side a and c
```

(continues on next page)

(continued from previous page)

```
//on side a (label 1) $ x \in [0,1] $ or $ x-y\in [-1,1] $
//on side c (label 3) $ x \in [-1,0]$ or $ x-y\in[-1,1] $
//so the common abscissa can be respectively $x$ and $x+1$
//or you can can try curviline abscissa $x-y$ and $x-y$
//1 first way
//fespace Vh(Th, P2, periodic=[[2, 1+x], [4, x], [1, x], [3, 1+x]]);
//2 second way
fespace Vh(Th, P2, periodic=[[2, x+y], [4, x+y], [1, x-y], [3, x-y]]);
Vh uh, vh;

// Problem
real intf = int2d(Th)(f);
real mTh = int2d(Th)(1);
real k =  intf / mTh;
problem laplace (uh, vh)
    = int2d(Th)(
        dx(uh)*dx(vh)
        + dy(uh)*dy(vh)
    )
    + int2d(Th)(
        (k-f)*vh
    )
    ;

// Solve
laplace;

// Plot
plot(uh, wait=true);
```

Fig. 5.6: The isovalue of solution u for $\Delta u = ((y+x)^2+1)((y-x)^2+1) - k$, in Ω and $\partial_n u = 0$ on hole, and with two periodic boundary condition on external border

An other example with no equal border, just to see if the code works.

Tip: Periodic boundary conditions - non-equal border

```
// Macro
//irregular boundary condition to build border AB
macro LINEBORDER(A, B, lab)
    border A#B(t=0,1){ real t1=1.-t;
    x=A#x*t1+B#x*t;
    y=A#y*t1+B#y*t;
    label=lab; } //EOM
// compute \||AB|\| A=(ax,ay) et B =(bx,by)
macro dist(ax, ay, bx, by)
    sqrt(square((ax)-(bx)) + square((ay)-(by))) //EOM
macro Grad(u) [dx(u), dy(u)] //EOM

// Parameters
int n = 10;
real Ax = 0.9, Ay = 1;
real Bx = 2, By = 1;
real Cx = 2.5, Cy = 2.5;
real Dx = 1, Dy = 2;
real gx = (Ax+Bx+Cx+Dx)/4.;
real gy = (Ay+By+Cy+Dy)/4.;

// Mesh
LINEBORDER(A,B,1)
LINEBORDER(B,C,2)
LINEBORDER(C,D,3)
LINEBORDER(D,A,4)
mesh Th=buildmesh(AB(n)+BC(n)+CD(n)+DA(n),fixedborder=1);

// Fespace
real l1 = dist(Ax,Ay,Bx,By);
real l2 = dist(Bx,By,Cx,Cy);
real l3 = dist(Cx,Cy,Dx,Dy);
real l4 = dist(Dx,Dy,Ax,Ay);
func s1 = dist(Ax,Ay,x,y)/l1; //absisse on AB = ||AX||/||AB||
func s2 = dist(Bx,By,x,y)/l2; //absisse on BC = ||BX||/||BC||
func s3 = dist(Cx,Cy,x,y)/l3; //absisse on CD = ||CX||/||CD||
func s4 = dist(Dx,Dy,x,y)/l4; //absisse on DA = ||DX||/||DA||
verbosity = 6; //to see the abscisse value of the periodic condition
fespace Vh(Th, P1, periodic=[[1, s1], [3, s3], [2, s2], [4, s4]]);
verbosity = 1; //reset verbosity
Vh u, v;

real cc = 0;
cc = int2d(Th)((x-gx)*(y-gy)-cc)/Th.area;
cout << "compatibility = " << int2d(Th)((x-gx)*(y-gy)-cc) <<endl;

// Problem
solve Poisson (u, v)
    = int2d(Th)(
        Grad(u)'*Grad(v)
        + 1e-10*u*v
    )
    -int2d(Th)(
```

(continues on next page)

(continued from previous page)

```
        10*v*((x-gx)*(y-gy)-cc)
    )
    ;

// Plot
plot(u, value=true);
```

Tip: Periodic boundry conditions - Poisson cube-balloon

```
load "msh3" load "tetgen" load "medit"

// Parameters
real hs = 0.1; //mesh size on sphere
int[int] N = [20, 20, 20];
real [int,int] B = [[-1, 1], [-1, 1], [-1, 1]];
int [int,int] L = [[1, 2], [3, 4], [5, 6]];

real x0 = 0.3, y0 = 0.4, z0 = 06;
func f = sin(x*2*pi+x0)*sin(y*2*pi+y0)*sin(z*2*pi+z0);

// Mesh
bool buildTh = 0;
mesh3 Th;
try { //a way to build one time the mesh or read it if the file exist
    Th = readmesh3("Th-hex-sph.mesh");
}
catch (...){
    buildTh = 1;
}

if (buildTh){
    include "MeshSurface.idp"

    // Surface Mesh
    mesh3 ThH = SurfaceHex(N, B, L, 1);
    mesh3 ThS = Sphere(0.5, hs, 7, 1);

    mesh3 ThHS = ThH + ThS;

    real voltet = (hs^3)/6.;
    real[int] domain = [0, 0, 0, 1, voltet, 0, 0, 0.7, 2, voltet];
    Th = tetg(ThHS, switch="pqaAAYYQ", nbofregions=2, regionlist=domain);

    savemesh(Th, "Th-hex-sph.mesh");
}

// Fespace
fespace Ph(Th, P0);
Ph reg = region;
cout << " centre = " << reg(0,0,0) << endl;
cout << " exterieur = " << reg(0,0,0.7) << endl;

verbosity = 50;
fespace Vh(Th, P1, periodic=[[3, x, z], [4, x, z], [1, y, z], [2, y, z], [5, x, y],
↪[6, x, y]]);
```

(continues on next page)

(a) View of the surface isovalue of periodic solution uh

(b) View a the cut of the solution uh with ffmedit

(continued from previous page)

```
verbosity = 1;
Vh uh,vh;

// Macro
macro Grad(u) [dx(u),dy(u),dz(u)] // EOM

// Problem
problem Poisson (uh, vh)
    = int3d(Th, 1)(
        Grad(uh)'*Grad(vh)*100
    )
    + int3d(Th, 2)(
        Grad(uh)'*Grad(vh)*2
    )
    + int3d(Th)(
        vh*f
    )
    ;

// Solve
Poisson;

// Plot
plot(uh, wait=true, nbiso=6);
medit("uh", Th, uh);
```

5.1.6 Poisson Problems with mixed boundary condition

Here we consider the Poisson equation with mixed boundary conditions:

For given functions f and g, find u such that:

$$\begin{array}{rcll} -\Delta u & = & f & \text{in } \Omega \\ u & = & g & \text{on } \Gamma_D \\ \partial u/\partial n & = & 0 & \text{on } \Gamma_N \end{array}$$

where Γ_D is a part of the boundary Γ and $\Gamma_N = \Gamma \setminus \overline{\Gamma_D}$.

The solution u has the singularity at the points $\{\gamma_1, \gamma_2\} = \overline{\Gamma_D} \cap \overline{\Gamma_N}$.

When $\Omega = \{(x, y);\ -1 < x < 1,\ 0 < y < 1\}$, $\Gamma_N = \{(x, y);\ -1 \le x < 0,\ y = 0\}$, $\Gamma_D = \partial\Omega \setminus \Gamma_N$, the singularity will appear at $\gamma_1 = (0, 0)$, $\gamma_2(-1, 0)$, and u has the expression:

$$u = K_i u_S + u_R,\ u_R \in H^2(\text{near } \gamma_i),\ i = 1, 2$$

with a constants K_i.

Here $u_S = r_j^{1/2} \sin(\theta_j/2)$ by the local polar coordinate $(r_j, \theta_j$ at γ_j such that $(r_1, \theta_1) = (r, \theta)$.

Instead of polar coordinate system (r, θ), we use that r = `sqrt` $(x^2 + y^2)$ and θ = `atan2` (y, x) in **FreeFEM**.

Assume that $f = -2 \times 30(x^2 + y^2)$ and $g = u_e = 10(x^2 + y^2)^{1/4} \sin\left([\tan^{-1}(y/x)]/2\right) + 30(x^2 y^2)$, where u_eS is the exact solution.

```
// Parameters
func f = -2*30*(x^2+y^2); //given function
//the singular term of the solution is K*us (K: constant)
func us = sin(atan2(y,x)/2)*sqrt( sqrt(x^2+y^2) );
real K = 10.;
func ue = K*us + 30*(x^2*y^2);

// Mesh
border N(t=0, 1){x=-1+t; y=0; label=1;};
border D1(t=0, 1){x=t; y=0; label=2;};
border D2(t=0, 1){x=1; y=t; label=2;};
border D3(t=0, 2){x=1-t; y=1; label=2;};
border D4(t=0, 1){x=-1; y=1-t; label=2;};

mesh T0h = buildmesh(N(10) + D1(10) + D2(10) + D3(20) + D4(10));
plot(T0h, wait=true);

// Fespace
fespace V0h(T0h, P1);
V0h u0, v0;

//Problem
solve Poisson0 (u0, v0)
    = int2d(T0h)(
        dx(u0)*dx(v0)
        + dy(u0)*dy(v0)
    )
    - int2d(T0h)(
        f*v0
    )
    + on(2, u0=ue)
    ;

// Mesh adaptation by the singular term
mesh Th = adaptmesh(T0h, us);
for (int i = 0; i < 5; i++)
mesh Th = adaptmesh(Th, us);

// Fespace
fespace Vh(Th, P1);
Vh u, v;

```

(continues on next page)

(continued from previous page)

```
43 // Problem
44 solve Poisson (u, v)
45     = int2d(Th)(
46         dx(u)*dx(v)
47         + dy(u)*dy(v)
48     )
49     - int2d(Th)(
50         f*v
51     )
52     + on(2, u=ue)
53     ;
54
55 // Plot
56 plot(Th);
57 plot(u, wait=true);
58
59 // Error in H1 norm
60 Vh uue = ue;
61 real H1e = sqrt( int2d(Th)(dx(uue)^2 + dy(uue)^2 + uue^2) );
62 Vh err0 = u0 - ue;
63 Vh err = u - ue;
64 Vh H1err0 = int2d(Th)(dx(err0)^2 + dy(err0)^2 + err0^2);
65 Vh H1err = int2d(Th)(dx(err)^2 + dy(err)^2 + err^2);
66 cout << "Relative error in first mesh = "<< int2d(Th)(H1err0)/H1e << endl;
67 cout << "Relative error in adaptive mesh = "<< int2d(Th)(H1err)/H1e << endl;
```

From line 35 to 37, mesh adaptations are done using the base of singular term.

In line 61, `H1e` = $|u_e|_{1,\Omega}$ is calculated.

In lines 64 and 65, the relative errors are calculated, that is:

$$\begin{array}{rcl} \|u_h^0 - u_e\|_{1,\Omega}/H1e & = & 0.120421 \\ \|u_h^a - u_e\|_{1,\Omega}/H1e & = & 0.0150581 \end{array}$$

where u_h^0 is the numerical solution in `T0h` and u_h^a is `u` in this program.

5.1.7 Poisson with mixed finite element

Here we consider the Poisson equation with mixed boundary value problems:

For given functions f , g_d, g_n, find p such that

$$\begin{array}{rcll} -\Delta p & = & 1 & \text{in } \Omega \\ p & = & g_d & \text{on } \Gamma_D \\ \partial p/\partial n & = & g_n & \text{on } \Gamma_N \end{array}$$

where Γ_D is a part of the boundary Γ and $\Gamma_N = \Gamma \setminus \overline{\Gamma_D}$.

The mixed formulation is: find p and $\mathbf{u}$ such that:

$$\begin{array}{rcll} \nabla p + \mathbf{u} & = & \mathbf{0} & \text{in } \Omega \\ \nabla.\mathbf{u} & = & f & \text{in } \Omega \\ p & = & g_d & \text{on } \Gamma_D \\ \partial u.n & = & \mathbf{g}_n.n & \text{on } \Gamma_N \end{array}$$

where $\mathbf{g}_n$ is a vector such that $\mathbf{g}_n.n = g_n$.

The variational formulation is:

$$\begin{array}{rrl} \forall \mathbf{v} \in \mathbb{V}_0 : & \int_\Omega p\nabla.v + \mathbf{v}\mathbf{v} & = \int_{\Gamma_d} g_d \mathbf{v}.n \\ \forall q \in \mathbb{P} : & \int_\Omega q\nabla.u & = \int_\Omega qf \\ & \partial u.n & = \mathbf{g}_n.n \quad \text{on } \Gamma_N \end{array}$$

where the functional space are:

$$\mathbb{P} = L^2(\Omega), \qquad \mathbb{V} = H(div) = \{\mathbf{v} \in L^2(\Omega)^2, \nabla.\mathbf{v} \in L^2(\Omega)\}$$

and:

$$\mathbb{V}_0 = \{\mathbf{v} \in \mathbb{V}; \quad \mathbf{v}.n = 0 \quad \text{on } \Gamma_N\}$$

To write the **FreeFEM** example, we have just to choose the finites elements spaces.

Here $\mathbb{V}$ space is discretize with Raviart-Thomas finite element `RT0` and $\mathbb{P}$ is discretize by constant finite element `P0`.

Example 9.10 LaplaceRT.edp

```
// Parameters
func gd = 1.;
func g1n = 1.;
func g2n = 1.;

// Mesh
mesh Th = square(10, 10);

// Fespace
fespace Vh(Th, RT0);
Vh [u1, u2];
Vh [v1, v2];

fespace Ph(Th, P0);
Ph p, q;

// Problem
problem laplaceMixte ([u1, u2, p], [v1, v2, q], solver=GMRES, eps=1.0e-10, tgv=1e30,
↪dimKrylov=150)
    = int2d(Th)(
        p*q*1e-15 //this term is here to be sure
        // that all sub matrix are inversible (LU requirement)
        + u1*v1
        + u2*v2
        + p*(dx(v1)+dy(v2))
        + (dx(u1)+dy(u2))*q
    )
    + int2d(Th) (
        q
    )
    - int1d(Th, 1, 2, 3)(
        gd*(v1*N.x +v2*N.y)
    )
    + on(4, u1=g1n, u2=g2n)
    ;

// Solve
laplaceMixte;

```

(continues on next page)

(continued from previous page)

```
// Plot
plot([u1, u2], coef=0.1, wait=true, value=true);
plot(p, fill=1, wait=true, value=true);
```

5.1.8 Metric Adaptation and residual error indicator

We do metric mesh adaption and compute the classical residual error indicator η_T on the element T for the Poisson problem.

First, we solve the same problem as in a previous example.

```
// Parameters
real[int] viso(21);
for (int i = 0; i < viso.n; i++)
viso[i] = 10.^(+(i-16.)/2.);
real error = 0.01;
func f = (x-y);

// Mesh
border ba(t=0, 1.0){x=t; y=0; label=1;}
border bb(t=0, 0.5){x=1; y=t; label=2;}
border bc(t=0, 0.5){x=1-t; y=0.5; label=3;}
border bd(t=0.5, 1){x=0.5; y=t; label=4;}
border be(t=0.5, 1){x=1-t; y=1; label=5;}
border bf(t=0.0, 1){x=0; y=1-t; label=6;}
mesh Th = buildmesh(ba(6) + bb(4) + bc(4) + bd(4) + be(4) + bf(6));

// Fespace
fespace Vh(Th, P2);
Vh u, v;

fespace Nh(Th, P0);
Nh rho;

// Problem
problem Probem1 (u, v, solver=CG, eps=1.0e-6)
    = int2d(Th, qforder=5)(
        u*v*1.0e-10
        + dx(u)*dx(v)
        + dy(u)*dy(v)
    )
    + int2d(Th, qforder=5)(
        - f*v
    )
    ;
```

Now, the local error indicator η_T is:

$$\eta_T = \left(h_T^2 ||f + \Delta u_h||^2_{L^2(T)} + \sum_{e \in \mathcal{E}_K} h_e \, || \, [\frac{\partial u_h}{\partial n_k}] \, ||^2_{L^2(e)} \right)^{\frac{1}{2}}$$

where h_T is the longest edge of T, $\mathcal{E}_T$ is the set of T edge not on $\Gamma = \partial\Omega$, n_T is the outside unit normal to K, h_e is the length of edge e, $[g]$ is the jump of the function g across edge (left value minus right value).

Of course, we can use a variational form to compute η_T^2, with test function constant function in each triangle.

(a) Density of the error indicator with isotropic P_2 metric

(b) Density of the error indicator with isotropic P_2 metric

```
// Error
varf indicator2 (uu, chiK)
    = intalledges(Th)(
        chiK*lenEdge*square(jump(N.x*dx(u) + N.y*dy(u)))
    )
    + int2d(Th)(
        chiK*square(hTriangle*(f + dxx(u) + dyy(u)))
    )
    ;

// Mesh adaptation loop
for (int i = 0; i < 4; i++){
    // Solve
    Probem1;
    cout << u[].min << " " << u[].max << endl;
    plot(u, wait=true);

    // Error
    rho[] = indicator2(0, Nh);
    rho = sqrt(rho);
    cout << "rho = min " << rho[].min << " max=" << rho[].max << endl;
    plot(rho, fill=true, wait=true, cmm="indicator density", value=true, viso=viso,
↪nbiso=viso.n);

    // Mesh adaptation
    plot(Th, wait=true, cmm="Mesh (before adaptation)");
    Th = adaptmesh(Th, [dx(u), dy(u)], err=error, anisomax=1);
    plot(Th, wait=true, cmm="Mesh (after adaptation)");
    u = u;
    rho = rho;
    error = error/2;
}
```

If the method is correct, we expect to look the graphics by an almost constant function η on your computer as in Fig. 5.8a and Fig. 5.8b.

5.1.9 Adaptation using residual error indicator

In the previous example we compute the error indicator, now we use it, to adapt the mesh. The new mesh size is given by the following formulae:

$$h_{n+1}(x) = \frac{h_n(x)}{f_n(\eta_K(x))}$$

where $\eta_n(x)$ is the level of error at point x given by the local error indicator, h_n is the previous "mesh size" field, and f_n is a user function define by $f_n = min(3, max(1/3, \eta_n/\eta_n^*))$ where $\eta_n^* = mean(\eta_n)c$, and c is an user coefficient generally close to one.

First a macro `MeshSizecomputation` is defined to get a P_1 mesh size as the average of edge length.

```
// macro the get the current mesh size parameter
// in:
// Th the mesh
// Vh P1 fespace on Th
// out :
// h: the Vh finite element finite set to the current mesh size
macro MeshSizecomputation (Th, Vh, h)
{
    real[int] count(Th.nv);
    /*mesh size (lenEdge = integral(e) 1 ds)*/
    varf vmeshsizen (u, v) = intalledges(Th, qfnbpE=1)(v);
    /*number of edges per vertex*/
    varf vedgecount (u, v) = intalledges(Th, qfnbpE=1)(v/lenEdge);
    /*mesh size*/
    count = vedgecount(0, Vh);
    h[] = 0.;
    h[] = vmeshsizen(0, Vh);
    cout << "count min = " << count.min << " max = " << count.max << endl;
    h[] = h[]./count;
    cout << "-- bound meshsize = " << h[].min << " " << h[].max << endl;
} //
```

A second macro to re-mesh according to the new mesh size.

```
// macro to remesh according the de residual indicator
// in:
// Th the mesh
// Ph P0 fespace on Th
// Vh P1 fespace on Th
// vindicator the varf to evaluate the indicator
// coef on etameam
macro ReMeshIndicator (Th, Ph, Vh, vindicator, coef)
{
    Vh h=0;
    /*evaluate the mesh size*/
    MeshSizecomputation(Th, Vh, h);
    Ph etak;
    etak[] = vindicator(0, Ph);
    etak[] = sqrt(etak[]);
    real etastar= coef*(etak[].sum/etak[].n);
    cout << "etastar = " << etastar << " sum = " << etak[].sum << " " << endl;

    /*etaK is discontinous*/
    /*we use P1 L2 projection with mass lumping*/
    Vh fn, sigma;
    varf veta(unused, v) = int2d(Th)(etak*v);
    varf vun(unused, v) = int2d(Th)(1*v);
    fn[] = veta(0, Vh);
    sigma[] = vun(0, Vh);
    fn[] = fn[]./ sigma[];
    fn = max(min(fn/etastar,3.),0.3333);

```

(continues on next page)

(continued from previous page)

```
    /*new mesh size*/
    h = h / fn;
    /*build the mesh*/
    Th = adaptmesh(Th, IsMetric=1, h, splitpbedge=1, nbvx=10000);
} //
```

```
// Parameters
real hinit = 0.2; //initial mesh size
func f=(x-y);

// Mesh
border ba(t=0, 1.0){x=t; y=0; label=1;}
border bb(t=0, 0.5){x=1; y=t; label=2;}
border bc(t=0, 0.5){x=1-t; y=0.5; label=3;}
border bd(t=0.5, 1){x=0.5; y=t; label=4;}
border be(t=0.5, 1){x=1-t; y=1; label=5;}
border bf(t=0.0, 1){x=0; y=1-t; label=6;}
mesh Th = buildmesh(ba(6) + bb(4) + bc(4) + bd(4) + be(4) + bf(6));

// Fespace
fespace Vh(Th, P1); //for the mesh size and solution
Vh h = hinit; //the FE function for the mesh size
Vh u, v;

fespace Ph(Th, P0); //for the error indicator

//Build a mesh with the given mesh size hinit
Th = adaptmesh(Th, h, IsMetric=1, splitpbedge=1, nbvx=10000);
plot(Th, wait=1);

// Problem
problem Poisson (u, v)
    = int2d(Th, qforder=5)(
        u*v*1.0e-10
        + dx(u)*dx(v)
        + dy(u)*dy(v)
    )
    - int2d(Th, qforder=5)(
        f*v
    )
    ;

varf indicator2 (unused, chiK)
    = intalledges(Th)(
        chiK*lenEdge*square(jump(N.x*dx(u) + N.y*dy(u)))
    )
    + int2d(Th)(
        chiK*square(hTriangle*(f + dxx(u) + dyy(u)))
    )
    ;

// Mesh adaptation loop
for (int i = 0; i < 10; i++){
    u = u;

    // Solve
```

(continues on next page)

(a) The error indicator with isotropic P_1

(b) The mesh and isovalue of the solution

(continued from previous page)

```
    Poisson;
    plot(Th, u, wait=true);

    real cc = 0.8;
    if (i > 5) cc=1;
    ReMeshIndicator(Th, Ph, Vh, indicator2, cc);
    plot(Th, wait=true);
}
```

5.2 Elasticity

Consider an elastic plate with undeformed shape $\Omega\times]-h,h[$ in $\mathbb{R}^3$, $\Omega\subset\mathbb{R}^2$.

By the deformation of the plate, we assume that a point $P(x_1,x_2,x_3)$ moves to $\mathcal{P}(\xi_1,\xi_2,\xi_3)$. The vector $\mathbf{u}=(u_1,u_2,u_3)=(\xi_1-x_1,\xi_2-x_2,\xi_3-x_3)$ is called the *displacement vector*.

By the deformation, the line segment $\overline{\mathbf{x},\mathbf{x}+\tau\Delta\mathbf{x}}$ moves approximately to $\overline{\mathbf{x}+\mathbf{u}(\mathbf{x}),\mathbf{x}+\tau\Delta\mathbf{x}+\mathbf{u}(\mathbf{x}+\tau\Delta\mathbf{x})}$ for small τ, where $\mathbf{x}=(x_1,x_2,x_3)$, $\Delta\mathbf{x}=(\Delta x_1,\Delta x_2,\Delta x_3)$.

We now calculate the ratio between two segments:

$$\eta(\tau)=\tau^{-1}|\Delta\mathbf{x}|^{-1}\left(|\mathbf{u}(\mathbf{x}+\tau\Delta\mathbf{x})-\mathbf{u}(\mathbf{x})+\tau\Delta\mathbf{x}|-\tau|\Delta\mathbf{x}|\right)$$

then we have (see e.g. [NECAS2017], p.32)

$$\lim_{\tau\to 0}\eta(\tau)=(1+2e_{ij}\nu_i\nu_j)^{1/2}-1,\quad 2e_{ij}=\frac{\partial u_k}{\partial x_i}\frac{\partial u_k}{\partial x_j}+\left(\frac{\partial u_i}{\partial x_j}+\frac{\partial u_j}{\partial x_i}\right)$$

where $\nu_i=\Delta x_i|\Delta\mathbf{x}|^{-1}$. If the deformation is *small*, then we may consider that:

$$(\partial u_k/\partial x_i)(\partial u_k/\partial x_i)\approx 0$$

and the following is called *small strain tensor*:

$$\varepsilon_{ij}(u)=\frac{1}{2}\left(\frac{\partial u_i}{\partial x_j}+\frac{\partial u_j}{\partial x_i}\right)$$

The tensor e_{ij} is called *finite strain tensor*.

Consider the small plane $\Delta\Pi(\mathbf{x})$ centered at $\mathbf{x}$ with the unit normal direction $\mathbf{n} = (n_1, n_2, n_3)$, then the surface on $\Delta\Pi(\mathbf{x})$ at $\mathbf{x}$ is:

$$(\sigma_{1j}(\mathbf{x})n_j, \sigma_{2j}(\mathbf{x})n_j, \sigma_{3j}(\mathbf{x})n_j)$$

where $\sigma_{ij}(\mathbf{x})$ is called *stress tensor* at $\mathbf{x}$. Hooke's law is the assumption of a linear relation between σ_{ij} and ε_{ij} such as:

$$\sigma_{ij}(\mathbf{x}) = c_{ijkl}(\mathbf{x})\varepsilon_{ij}(\mathbf{x})$$

with the symmetry $c_{ijkl} = c_{jikl}, c_{ijkl} = c_{ijlk}, c_{ijkl} = c_{klij}$.

If Hooke's tensor $c_{ijkl}(\mathbf{x})$ do not depend on the choice of coordinate system, the material is called *isotropic* at $\mathbf{x}$.

If c_{ijkl} is constant, the material is called *homogeneous*. In homogeneous isotropic case, there is *Lamé constants* λ, μ (see e.g. [NECAS2017], p.43) satisfying

$$\sigma_{ij} = \lambda\delta_{ij}\mathrm{div}\mathbf{u} + 2\mu\varepsilon_{ij}$$

where δ_{ij} is Kronecker's delta.

We assume that the elastic plate is fixed on $\Gamma_D\times]-h,h[$, $\Gamma_D \subset \partial\Omega$. If the body force $\mathbf{f} = (f_1, f_2, f_3)$ is given in $\Omega\times]-h,h[$ and surface force $\mathbf{g}$ is given in $\Gamma_N\times]-h,h[, \Gamma_N = \partial\Omega \setminus \overline{\Gamma_D}$, then the equation of equilibrium is given as follows:

$$\begin{array}{rcl} -\partial_j\sigma_{ij} & = & f_i \ \text{ in } \Omega\times]-h,h[, \quad i = 1,2,3 \\ \sigma_{ij}n_j & = & g_i \ \text{ on } \Gamma_N\times]-h,h[, \quad u_i = 0 \ \text{ on } \Gamma_D\times]-h,h[, \quad i = 1,2,3 \end{array} \tag{5.4}$$

We now explain the plain elasticity.

- **Plain strain:**

  On the end of plate, the contact condition $u_3 = 0,\ g_3 =$ is satisfied.

  In this case, we can suppose that $f_3 = g_3 = u_3 = 0$ and $\mathbf{u}(x_1, x_2, x_3) = \overline{u}(x_1, x_2)$ for all $-h < x_3 < h$.

- **Plain stress:**

  The cylinder is assumed to be very thin and subjected to no load on the ends $x_3 = \pm h$, that is,

  $$\sigma_{3i} = 0, \quad x_3 = \pm h, \quad i\ 1,2,3$$

  The assumption leads that $\sigma_{3i} = 0$ in $\Omega\times]-h,h[$ and $\mathbf{u}(x_1, x_2, x_3) = \overline{u}(x_1, x_2)$ for all $-h < x_3 < h$.

- **Generalized plain stress:**

  The cylinder is subjected to no load at $x_3 = \pm h$. Introducing the mean values with respect to thickness,

  $$\overline{u}_i(x_1, x_2) = \frac{1}{2h}\int_{-h}^{h} u(x_1, x_2, x_3)dx_3$$

  and we derive $\overline{u}_3 \equiv 0$. Similarly we define the mean values $\overline{f}, \overline{g}$ of the body force and surface force as well as the mean values $\overline{\varepsilon}_{ij}$ and $\overline{\sigma}_{ij}$ of the components of stress and strain, respectively.

In what follows we omit the overlines of $\overline{u}, \overline{f}, \overline{g}, \overline{\varepsilon}_{ij}$ and $\overline{\varepsilon}_{ij}$. Then we obtain similar equation of equilibrium given in (5.4) replacing $\Omega\times]-h,h[$ with Ω and changing $i = 1,2$. In the case of plane stress, $\sigma_{ij} = \lambda^*\delta_{ij}\mathrm{div}u + 2\mu\varepsilon_{ij}, \lambda^* = (2\lambda\mu)/(\lambda+\mu)$.

The equations of elasticity are naturally written in variational form for the displacement vector $\mathbf{u}(\mathbf{x}) \in V$ as:

$$\int_\Omega [2\mu\epsilon_{ij}(\mathbf{u})\epsilon_{ij}(\mathbf{v}) + \lambda\epsilon_{ii}(\mathbf{u})\epsilon_{jj}(\mathbf{v})] = \int_\Omega \mathbf{f}\cdot\mathbf{v} + \int_\Gamma \mathbf{g}\cdot\mathbf{v}, \forall\mathbf{v} \in V$$

where V is the linear closed subspace of $H^1(\Omega)^2$.

Tip: Beam

Consider an elastic plate with the undeformed rectangle shape $]0,10[\times]0,2[$. The body force is the gravity force $\mathbf{f}$ and the boundary force $\mathbf{g}$ is zero on lower and upper side. On the two vertical sides of the beam are fixed.

```
// Parameters
real E = 21.5;
real sigma = 0.29;
real gravity = -0.05;

// Mesh
border a(t=2, 0){x=0; y=t; label=1;}
border b(t=0, 10){x=t; y=0; label=2;}
border c(t=0, 2){ x=10; y=t; label=1;}
border d(t=0, 10){ x=10-t; y=2; label=3;}
mesh th = buildmesh(b(20) + c(5) + d(20) + a(5));

// Fespace
fespace Vh(th, [P1, P1]);
Vh [uu, vv];
Vh [w, s];

// Macro
real sqrt2 = sqrt(2.);
macro epsilon(u1, u2) [dx(u1), dy(u2), (dy(u1)+dx(u2))/sqrt2] //
macro div(u,v) (dx(u) + dy(v)) //

// Problem
real mu = E/(2*(1+sigma));
real lambda = E*sigma/((1+sigma)*(1-2*sigma));
solve Elasticity ([uu, vv], [w, s])
    = int2d(th)(
          lambda*div(w,s)*div(uu,vv)
        + 2.*mu*( epsilon(w,s)'*epsilon(uu,vv) )
    )
    + int2d(th)(
        - gravity*s
    )
    + on(1, uu=0, vv=0)
;

// Plot
plot([uu, vv], wait=true);
plot([uu,vv], wait=true, bb=[[-0.5, 2.5], [2.5, -0.5]]);

// Movemesh
mesh th1 = movemesh(th, [x+uu, y+vv]);
plot(th1, wait=true);
```

Tip: Beam 3D

Consider elastic box with the undeformed parallelepiped shape $]0,5[\times]0,1[\times]0,1[$. The body force is the gravity force $\mathbf{f}$ and the boundary force $\mathbf{g}$ is zero on all face except one the one vertical left face where the beam is fixed.

```
include "cube.idp"

// Parameters
int[int] Nxyz = [20, 5, 5];
real [int, int] Bxyz = [[0., 5.], [0., 1.], [0., 1.]];
int [int, int] Lxyz = [[1, 2], [2, 2], [2, 2]];

real E = 21.5e4;
real sigma = 0.29;
real gravity = -0.05;

// Mesh
mesh3 Th = Cube(Nxyz, Bxyz, Lxyz);

// Fespace
fespace Vh(Th, [P1, P1, P1]);
Vh [u1, u2, u3], [v1, v2, v3];

// Macro
real sqrt2 = sqrt(2.);
macro epsilon(u1, u2, u3) [
    dx(u1), dy(u2), dz(u3),
    (dz(u2) + dy(u3))/sqrt2,
    (dz(u1) + dx(u3))/sqrt2,
    (dy(u1) + dx(u2))/sqrt2] //
macro div(u1, u2, u3) (dx(u1) + dy(u2) + dz(u3)) //

// Problem
real mu = E/(2*(1+sigma));
real lambda = E*sigma/((1+sigma)*(1-2*sigma));

solve Lame ([u1, u2, u3], [v1, v2, v3])
    = int3d(Th)(
          lambda*div(u1, u2, u3)*div(v1, v2, v3)
        + 2.*mu*( epsilon(u1, u2, u3)'*epsilon(v1, v2, v3) )
    )
    - int3d(Th)(
          gravity*v3
    )
    + on(1, u1=0, u2=0, u3=0)
    ;

// Display
real dmax = u1[].max;
cout << "max displacement = " << dmax << endl;

// Movemesh
real coef = 0.1/dmax;
int[int] ref2 = [1, 0, 2, 0];
mesh3 Thm = movemesh3(Th, transfo=[x+u1*coef, y+u2*coef, z+u3*coef], label=ref2);
Thm = change(Thm, label=ref2);

// Plot
plot(Th, Thm, wait=true, cmm="coef amplification = "+coef);
```

Fig. 5.10: 3d Beam deformed and undeformed box

5.2.1 Fracture Mechanics

Consider the plate with the crack whose undeformed shape is a curve Σ with the two edges $\gamma_1,\ \gamma_2$.

We assume the stress tensor σ_{ij} is the state of plate stress regarding $(x, y) \in \Omega_\Sigma = \Omega \setminus \Sigma$. Here Ω stands for the undeformed shape of elastic plate without crack.

If the part Γ_N of the boundary $\partial\Omega$ is fixed and a load $\mathcal{L} = (\mathbf{f}, \mathbf{g}) \in L^2(\Omega)^2 \times L^2(\Gamma_N)^2$ is given, then the displacement $\mathbf{u}$ is the minimizer of the potential energy functional:

$$\mathcal{E}(\mathbf{v}; \mathcal{L}, \Omega_\Sigma) = \int_{\Omega_\Sigma} \{w(x, \mathbf{v}) - \mathbf{f} \cdot \mathbf{v}\} - \int_{\Gamma_N} \mathbf{g} \cdot \mathbf{v}$$

over the functional space $V(\Omega_\Sigma)$,

$$V(\Omega_\Sigma) = \left\{\mathbf{v} \in H^1(\Omega_\Sigma)^2;\ \mathbf{v} = 0 \quad \text{on } \Gamma_D = \partial\Omega \setminus \overline{\Gamma_N}\right\},$$

where $w(x, \mathbf{v}) = \sigma_{ij}(\mathbf{v})\varepsilon_{ij}(\mathbf{v})/2$,

$$\sigma_{ij}(\mathbf{v}) = C_{ijkl}(x)\varepsilon_{kl}(\mathbf{v}), \quad \varepsilon_{ij}(\mathbf{v}) = (\partial v_i/\partial x_j + \partial v_j/\partial x_i)/2, \qquad (C_{ijkl}: \quad \text{Hooke's tensor}).$$

If the elasticity is homogeneous isotropic, then the displacement $\mathbf{u}(x)$ is decomposed in an open neighborhood U_k of γ_k as in (see e.g. [OHTSUKA2000])

$$\mathbf{u}(x) = \sum_{l=1}^{2} K_l(\gamma_k) r_k^{1/2} S_{kl}^C(\theta_k) + \mathbf{u}_{k,R}(x) \quad \text{for } x \in \Omega_\Sigma \cap U_k,\ k = 1, 2 \tag{5.5}$$

with $\mathbf{u}_{k,R} \in H^2(\Omega_\Sigma \cap U_k)^2$, where $U_k,\ k = 1, 2$ are open neighborhoods of γ_k such that $\partial L_1 \cap U_1 = \gamma_1,\ \partial L_m \cap U_2 = \gamma_2$, and

$$\begin{aligned} S_{k1}^C(\theta_k) &= \tfrac{1}{4\mu}\tfrac{1}{(2\pi)^{1/2}} \begin{bmatrix} [2\kappa - 1]\cos(\theta_k/2) - \cos(3\theta_k/2) \\ -[2\kappa + 1]\sin(\theta_k/2) + \sin(3\theta_k/2) \end{bmatrix}, \\ S_{k2}^C(\theta_k) &= \tfrac{1}{4\mu}\tfrac{1}{(2\pi)^{1/2}} \begin{bmatrix} -[2\kappa - 1]\sin(\theta_k/2) + 3\sin(3\theta_k/2) \\ -[2\kappa + 1]\cos(\theta_k/2) + \cos(3\theta_k/2) \end{bmatrix}. \end{aligned}$$

where μ is the shear modulus of elasticity, $\kappa = 3 - 4\nu$ (ν is the Poisson's ratio) for plane strain and $\kappa = \frac{3-\nu}{1+\nu}$ for plane stress.

The coefficients $K_1(\gamma_i)$ and $K_2(\gamma_i)$, which are important parameters in fracture mechanics, are called stress intensity factors of the opening mode (mode I) and the sliding mode (mode II), respectively.

For simplicity, we consider the following simple crack

$$\Omega = \{(x,y):\ -1 < x < 1, -1 < y < 1\}, \qquad \Sigma = \{(x,y):\ -1 \le x \le 0, y = 0\}$$

with only one crack tip $\gamma = (0,0)$. Unfortunately, **FreeFEM** cannot treat crack, so we use the modification of the domain with U-shape channel (see *U-shape example*, Fig. 3.20) with $d = 0.0001$. The undeformed crack Σ is approximated by

$$\Sigma_d = \{(x,y):\ -1 \le x \le -10*d, -d \le y \le d\} \cup \{(x,y):\ -10*d \le x \le 0, -d + 0.1*x \le y \le d - 0.1*x\}$$

and Γ_D =R in *U-shape example*, Fig. 3.20.

In this example, we use three technique:

- Fast Finite Element Interpolator from the mesh `Th` to `Zoom` for the scale-up of near γ.
- After obtaining the displacement vector $\mathbf{u} = (u,v)$, we shall watch the deformation of the crack near γ as follows,

```
mesh Plate = movemesh(Zoom, [x+u, y+v]);
plot(Plate);
```

- Adaptivity is an important technique here, because a large singularity occurs at γ as shown in (5.5).

The first example creates mode I deformation by the opposed surface force on `B` and `T` in the vertical direction of Σ, and the displacement is fixed on `R`.

In a laboratory, fracture engineers use photoelasticity to make stress field visible, which shows the principal stress difference

$$\sigma_1 - \sigma_2 = \sqrt{(\sigma_{11} - \sigma_{22})^2 + 4\sigma_{12}^2}$$

where σ_1 and σ_2 are the principal stresses.

In opening mode, the photoelasticity make symmetric pattern concentrated at γ.

Tip: Crack Opening, $K_2(\gamma) = 0$

```
//Parameters
real d = 0.0001; int n = 5; real cb = 1, ca = 1, tip = 0.0;

real E = 21.5;
real sigma = 0.29;

// Mesh
border L1(t=0, ca-d){x=-cb; y=-d-t;}
border L2(t=0, ca-d){x=-cb; y=ca-t;}
border B(t=0, 2){x=cb*(t-1); y=-ca;}
border C1(t=0, 1){x=-ca*(1-t)+(tip-10*d)*t; y=d;}
border C21(t=0, 1){x=(tip-10*d)*(1-t)+tip*t; y=d*(1-t);}
border C22(t=0, 1){x=(tip-10*d)*t+tip*(1-t); y=-d*t;}
border C3(t=0, 1){x=(tip-10*d)*(1-t)-ca*t; y=-d;}
border C4(t=0, 2*d){x=-ca; y=-d+t;}
border R(t=0, 2){x=cb; y=cb*(t-1);}
border T(t=0, 2){x=cb*(1-t); y=ca;}
mesh Th = buildmesh(L1(n/2) + L2(n/2) + B(n)
    + C1(n) + C21(3) + C22(3) + C3(n) + R(n) + T(n));
plot(Th, wait=true);

```

(continues on next page)

(continued from previous page)

```
cb=0.1; ca=0.1;
mesh Zoom = buildmesh(L1(n/2) + L2(n/2) + B(n) + C1(n)
    + C21(3) + C22(3) + C3(n) + R(n) + T(n));
plot(Zoom, wait=true);

// Fespace
fespace Vh(Th, [P2, P2]);
Vh [u, v];
Vh [w, s];

fespace zVh(Zoom, P2);
zVh Sx, Sy, Sxy, N;

// Problem
real mu = E/(2*(1+sigma));
real lambda = E*sigma/((1+sigma)*(1-2*sigma));
solve Problem ([u, v], [w, s])
    = int2d(Th)(
          2*mu*(dx(u)*dx(w) + ((dx(v)+dy(u))*(dx(s)+dy(w)))/4)
        + lambda*(dx(u) + dy(v))*(dx(w) + dy(s))/2
    )
    -int1d(Th, T)(
          0.1*(1-x)*s
    )
    +int1d(Th, B)(
          0.1*(1-x)*s
    )
    +on(R, u=0, v=0)
    ;

// Loop
for (int i = 1; i <= 5; i++){
    mesh Plate = movemesh(Zoom, [x+u, y+v]); //deformation near gamma
    Sx = lambda*(dx(u) + dy(v)) + 2*mu*dx(u);
    Sy = lambda*(dx(u) + dy(v)) + 2*mu*dy(v);
    Sxy = mu*(dy(u) + dx(v));
    N = 0.1*1*sqrt((Sx-Sy)^2 + 4*Sxy^2); //principal stress difference
    if (i == 1){
        plot(Plate, bw=1);
        plot(N, bw=1);
    }
    else if (i == 5){
        plot(Plate, bw=1);
        plot(N, bw=1);
        break;
    }

    // Adaptmesh
    Th = adaptmesh(Th, [u, v]);

    // Solve
    Problem;
}
```

It is difficult to create mode II deformation by the opposed shear force on `B` and `T` that is observed in a laboratory. So

(a) Crack open displacement (COD) on the first mesh

(b) Principal stress difference on the first mesh

(c) COD on the last adaptive mesh

(d) Principal stress difference on the last adaptive mesh

we use the body shear force along Σ, that is, the x-component f_1 of the body force $\mathbf{f}$ is given by

$$f_1(x,y) = H(y-0.001) * H(0.1-y) - H(-y-0.001) * H(y+0.1)$$

where $H(t) = 1$ if $t > 0$; $= 0$ if $t < 0$.

Tip: Crack Sliding, $K_2(\gamma) = 0$

```
// Parameters
real d = 0.0001; int n = 5; real cb = 1, ca = 1, tip = 0.0;

real E = 21.5;
real sigma = 0.29;

// Mesh
border L1(t=0, ca-d){x=-cb; y=-d-t;}
border L2(t=0, ca-d){x=-cb; y=ca-t;}
border B(t=0, 2){x=cb*(t-1); y=-ca;}
border C1(t=0, 1){x=-ca*(1-t)+(tip-10*d)*t; y=d;}
border C21(t=0, 1){x=(tip-10*d)*(1-t)+tip*t; y=d*(1-t);}
border C22(t=0, 1){x=(tip-10*d)*t+tip*(1-t); y=-d*t;}
border C3(t=0, 1){x=(tip-10*d)*(1-t)-ca*t; y=-d;}
border C4(t=0, 2*d){x=-ca; y=-d+t;}
border R(t=0, 2){x=cb; y=cb*(t-1);}
border T(t=0, 2){x=cb*(1-t); y=ca;}
mesh Th = buildmesh(L1(n/2) + L2(n/2) + B(n)
    + C1(n) + C21(3) + C22(3) + C3(n) + R(n) + T(n));
plot(Th, wait=true);

cb=0.1; ca=0.1;
mesh Zoom = buildmesh(L1(n/2) + L2(n/2) + B(n) + C1(n)
    + C21(3) + C22(3) + C3(n) + R(n) + T(n));
plot(Zoom, wait=true);

// Fespace
fespace Vh(Th, [P2, P2]);
Vh [u, v];
Vh [w, s];

fespace zVh(Zoom, P2);
zVh Sx, Sy, Sxy, N;

fespace Vh1(Th,P1);
Vh1 fx = ((y>0.001)*(y<0.1))-((y<-0.001)*(y>-0.1));

// Problem
real mu = E/(2*(1+sigma));
real lambda = E*sigma/((1+sigma)*(1-2*sigma));
solve Problem ([u, v], [w, s])
    = int2d(Th)(
          2*mu*(dx(u)*dx(w) + ((dx(v) + dy(u))*(dx(s)+ dy(w)))/4)
        + lambda*(dx(u) + dy(v))*(dx(w) + dy(s))/2
    )
    -int2d(Th)(
          fx*w
    )
    +on(R, u=0, v=0)
```

(continues on next page)

(continued from previous page)

```
    ;

// Loop
for (int i = 1; i <= 3; i++){
    mesh Plate = movemesh(Zoom, [x+u, y+v]); //deformation near gamma
    Sx = lambda*(dx(u) + dy(v)) + 2*mu*dx(u);
    Sy = lambda*(dx(u) + dy(v)) + 2*mu*dy(v);
    Sxy = mu*(dy(u) + dx(v));
    N = 0.1*1*sqrt((Sx-Sy)^2 + 4*Sxy^2); //principal stress difference
    if (i == 1){
        plot(Plate, bw=1);
        plot(N, bw=1);
    }
    else if (i == 3) {
        plot(Plate, bw=1);
        plot(N, bw=1);
        break;
    }

    // Adaptmesh
    Th=adaptmesh(Th, [u, v]);

    // Solve
    Problem;
}
```

5.3 Non-linear static problems

Here we propose to solve the following non-linear academic problem of minimization of a functional:

$$J(u) = \int_\Omega \frac{1}{2} f(|\nabla u|^2) - u * b$$

where u is function of $H_0^1(\Omega)$ and f defined by:

$$f(x) = a * x + x - ln(1+x),\ f'(x) = a + \frac{x}{1+x},\ f''(x) = \frac{1}{(1+x)^2}$$

5.3.1 Newton-Raphson algorithm

Now, we solve the Euler problem $\nabla J(u) = 0$ with Newton-Raphson algorithm, that is:

$$u^{n+1} = u^n - (\nabla^2 J(u^n))^{-1} * \nabla J(u^n)$$

```
// Parameters
real a = 0.001;
func b = 1.;

// Mesh
mesh Th = square(10, 10);
Th = adaptmesh(Th, 0.05, IsMetric=1, splitpbedge=1);
```

(continues on next page)

(a) COD on the first mesh

(b) Principal stress difference in the first mesh

(c) COD on the last adaptive mesh

(d) Principal stress difference on the last adaptive mesh

(continued from previous page)

```
plot(Th, wait=true);

// Fespace
fespace Vh(Th, P1);
Vh u=0;
Vh v, w;

fespace Ph(Th, P1dc);
Ph alpha; //to store |nabla u|^2
Ph dalpha ; //to store 2f''(|nabla u|^2)

// Function
func real f (real u){
    return u*a + u - log(1.+u);
}
func real df (real u){
    return a +u/(1.+u);
}
func real ddf (real u){
    return 1. / ((1.+u)*(1.+u));
}

// Problem
//the variational form of evaluate dJ = nabla J
//dJ = f'()*(dx(u)*dx(vh) + dy(u)*dy(vh))
varf vdJ (uh, vh)
    = int2d(Th)(
          alpha*(dx(u)*dx(vh) + dy(u)*dy(vh))
        - b*vh
    )
    + on(1, 2, 3, 4, uh=0)
    ;

//the variational form of evaluate ddJ = nabla^2 J
//hJ(uh,vh) = f'()*(dx(uh)*dx(vh) + dy(uh)*dy(vh))
//  + 2*f''()(dx(u)*dx(uh) + dy(u)*dy(uh)) * (dx(u)*dx(vh) + dy(u)*dy(vh))
varf vhJ (uh, vh)
    = int2d(Th)(
          alpha*(dx(uh)*dx(vh) + dy(uh)*dy(vh))
        + dalpha*(dx(u)*dx(vh) + dy(u)*dy(vh))*(dx(u)*dx(uh) + dy(u)*dy(uh))
    )
    + on(1, 2, 3, 4, uh=0)
    ;

// Newton algorithm
for (int i = 0; i < 100; i++){
    // Compute f' and f''
    alpha = df(dx(u)*dx(u) + dy(u)*dy(u));
    dalpha = 2*ddf(dx(u)*dx(u) + dy(u)*dy(u));

    // nabla J
    v[]= vdJ(0, Vh);

    // Residual
    real res = v[]'*v[];
    cout << i << " residu^2 = " << res << endl;
    if( res < 1e-12) break;
```

(continues on next page)

(continued from previous page)

```
    // HJ
    matrix H = vhJ(Vh, Vh, factorize=1, solver=LU);

    // Newton
    w[] = H^-1*v[];
    u[] -= w[];
}

// Plot
plot (u, wait=true, cmm="Solution with Newton-Raphson");
```

5.4 Eigen value problems

This section depends on your installation of **FreeFEM**; you need to have compiled `ARPACK`. This tool is available in **FreeFEM** if the word eigenvalue appears in line `Load:`, like:

```
-- FreeFem++ v*.** (date *** *** ** **:**:** CET ****)
 file : ***.edp
 Load: lg_fem lg_mesh eigenvalue
```

This tool is based on arpack++, the object-oriented version of `ARPACK` eigenvalue package [LEHOUCQ1998].

The function `EigenValue` computes the generalized eigenvalue of $Au = \lambda Bu$. The Shift-invert method is used by default, with sigma =σ the shift of the method.

The matrix OP is defined with $A - \sigma B$.

The return value is the number of converged eigenvalues (can be greater than the number of requested eigenvalues nev=)

```
int k = EigenValue(OP, B, nev=Nev, sigma=Sigma);
```

where the matrix $OP = A - \sigma B$ with a solver and boundary condition, and the matrix B.

There is also a functional interface:

```
int k = EigenValue(n, FOP1, FB, nev=Nev, sigma=Sigma);
```

where n is the size of the problem, and the operators are now defined through functions, defining respectively the matrix product of OP^{-1} and B, as in

```
int n = OP1.n;
func real[int] FOP1(real[int] & u){ real[int] Au = OP^-1*u; return Au; }
func real[int] FB(real[int] & u){ real[int] Au = B*u; return Au; }
```

If you want finer control over the method employed in `ARPACK`, you can specify which mode `ARPACK` will work with (`mode=` , see ARPACK documentation [LEHOUCQ1998]). The operators necessary for the chosen mode can be passed through the optional parameters `A=`, `A1=`, `B=`, `B1=`, (see below).

- `mode=1`: Regular mode for solving $Au = \lambda u$

  ```
  int k = EigenValue(n, A=FOP, mode=1, nev=Nev);
  ```

  where the function FOP defines the matrix product of A

- `mode=2`: Regular inverse mode for solving $Au = \lambda Bu$

```
int k = EigenValue(n, A=FOP, B=FB, B1=FB1, mode=2, nev=Nev);
```

  where the functions FOP, FB and FB1 define respectively the matrix product of A, B and B^{-1}

- `mode=3`: Shift-invert mode for solving $Au = \lambda Bu$

```
int k = EigenValue(n, A1=FOP1, B=FB, mode=3, sigma=Sigma, nev=Nev);
```

  where the functions FOP1 and FB define respectively the matrix product of $OP^{-1} = (A - \sigma B)^{-1}$ and B

You can also specify which subset of eigenvalues you want to compute (`which=`). The default value is `which="LM"`, for eigenvalues with largest magnitude. `"SM"` is for smallest magnitude, `"LA"` for largest algebraic value, `"SA"` for smallest algebraic value, and `"BE"` for both ends of the spectrum.

Remark: For complex problems, you need to use the keyword `complexEigenValue` instead of `EigenValue` when passing operators through functions.

Note: Boundary condition and Eigenvalue Problems

The locking (Dirichlet) boundary condition is make with exact penalization so we put `1e30=tgv` on the diagonal term of the locked degree of freedom (see *Finite element chapter*). So take Dirichlet boundary condition just on A and not on B because we solve $w = OP^{-1} * B * v$.

If you put locking (Dirichlet) boundary condition on B matrix (with key work `on`) you get small spurious modes (10^{-30}), due to boundary condition, but if you forget the locking boundary condition on B matrix (no keywork `on`) you get huge spurious (10^{30}) modes associated to these boundary conditons. We compute only small mode, so we get the good one in this case.

- `sym=` The problem is symmetric (all the eigen value are real)
- `nev=` The number desired eigenvalues (nev) close to the shift.
- `value=` The array to store the real part of the eigenvalues
- `ivalue=` The array to store the imaginary part of the eigenvalues
- `vector=` The FE function array to store the eigenvectors
- `rawvector=` An array of type `real[int,int]` to store eigenvectors by column.

  For real non symmetric problems, complex eigenvectors are given as two consecutive vectors, so if eigenvalue k and $k+1$ are complex conjugate eigenvalues, the kth vector will contain the real part and the $k+1$th vector the imaginary part of the corresponding complex conjugate eigenvectors.

- `tol=` The relative accuracy to which eigenvalues are to be determined;
- `sigma=` The shift value;
- `maxit=` The maximum number of iterations allowed;
- `ncv=` The number of Arnoldi vectors generated at each iteration of `ARPACK`;
- `mode=` The computational mode used by `ARPACK` (see above);
- `which=` The requested subset of eigenvalues (see above).

Tip: Laplace eigenvalue

In the first example, we compute the eigenvalues and the eigenvectors of the Dirichlet problem on square $\Omega =]0, \pi[^2$.

The problem is to find: λ, and ∇u_λ in $\mathbb{R}\times H_0^1(\Omega)$

$$\int_\Omega \nabla u_\lambda \nabla v = \lambda \int_\Omega uv \quad \forall v \in H_0^1(\Omega)$$

The exact eigenvalues are $\lambda_{n,m} = (n^2+m^2), (n,m) \in {\mathbb{N}_*}^2$ with the associated eigenvectors are $u_{m,n} = \sin(nx) * \sin(my)$.

We use the generalized inverse shift mode of the *arpack++* library, to find 20 eigenvalues and eigenvectors close to the shift value $\sigma = 20$.

```
// Parameters
verbosity=0;
real sigma = 20; //value of the shift
int nev = 20; //number of computed eigen value close to sigma

// Mesh
mesh Th = square(20, 20, [pi*x, pi*y]);

// Fespace
fespace Vh(Th, P2);
Vh u1, u2;

// Problem
// OP = A - sigma B ; // the shifted matrix
varf op (u1, u2)
    = int2d(Th)(
          dx(u1)*dx(u2)
        + dy(u1)*dy(u2)
        - sigma* u1*u2
    )
    + on(1, 2, 3, 4, u1=0)
    ;

varf b ([u1], [u2]) = int2d(Th)(u1*u2); //no boundary condition

matrix OP = op(Vh, Vh, solver=Crout, factorize=1); //crout solver because the matrix␣
↪in not positive
matrix B = b(Vh, Vh, solver=CG, eps=1e-20);

// important remark:
// the boundary condition is make with exact penalization:
// we put 1e30=tgv on the diagonal term of the lock degree of freedom.
// So take Dirichlet boundary condition just on $a$ variational form
// and not on $b$ variational form.
// because we solve $ w=OP^-1*B*v $

// Solve
real[int] ev(nev); //to store the nev eigenvalue
Vh[int] eV(nev); //to store the nev eigenvector

int k = EigenValue(OP, B, sym=true, sigma=sigma, value=ev, vector=eV,
    tol=1e-10, maxit=0, ncv=0);

// Display & Plot
for (int i = 0; i < k; i++){
    u1 = eV[i];
    real gg = int2d(Th)(dx(u1)*dx(u1) + dy(u1)*dy(u1));
    real mm = int2d(Th)(u1*u1) ;
```

(continues on next page)

(a) Isovalue of 11th eigenvector $u_{4,3} - u_{3,4}$

(b) Isovalue of 12th eigenvector $u_{4,3} + u_{3,4}$

(continued from previous page)

```
    cout << "lambda[" << i << "] = " << ev[i] << ", err= " << int2d(Th)(dx(u1)*dx(u1)
+ dy(u1)*dy(u1) - (ev[i])*u1*u1) << endl;
    plot(eV[i], cmm="Eigen Vector "+i+" value ="+ev[i], wait=true, value=true);
}
```

The output of this example is:

```
lambda[0] = 5.0002, err= -1.46519e-11
lambda[1] = 8.00074, err= -4.05158e-11
lambda[2] = 10.0011, err= 2.84925e-12
lambda[3] = 10.0011, err= -7.25456e-12
lambda[4] = 13.002, err= -1.74257e-10
lambda[5] = 13.0039, err= 1.22554e-11
lambda[6] = 17.0046, err= -1.06274e-11
lambda[7] = 17.0048, err= 1.03883e-10
lambda[8] = 18.0083, err= -4.05497e-11
lambda[9] = 20.0096, err= -2.21678e-13
lambda[10] = 20.0096, err= -4.16212e-14
lambda[11] = 25.014, err= -7.42931e-10
lambda[12] = 25.0283, err= 6.77444e-10
lambda[13] = 26.0159, err= 3.19864e-11
lambda[14] = 26.0159, err= -4.9652e-12
lambda[15] = 29.0258, err= -9.99573e-11
lambda[16] = 29.0273, err= 1.38242e-10
lambda[17] = 32.0449, err= 1.2522e-10
lambda[18] = 34.049, err= 3.40213e-11
lambda[19] = 34.0492, err= 2.41751e-10
```

5.5 Evolution problems

FreeFEM also solves evolution problems such as the heat equation:

$$\begin{array}{rcll} \frac{\partial u}{\partial t} - \mu\Delta u & = & f & \text{in } \Omega\times]0,T[\\ u(\mathbf{x},0) & = & u_0(\mathbf{x}) & \text{in } \Omega \\ (\partial u/\partial n)\,(\mathbf{x},t) & = & 0 & \text{on } \partial\Omega\times]0,T[\end{array} \tag{5.6}$$

with a positive viscosity coefficient μ and homogeneous Neumann boundary conditions.

We solve (5.6) by FEM in space and finite differences in time.

We use the definition of the partial derivative of the solution in the time derivative:

$$\frac{\partial u}{\partial t}(x,y,t) = \lim_{\tau\to 0}\frac{u(x,y,t)-u(x,y,t-\tau)}{\tau}$$

which indicates that $u^m(x,y) = u(x,y,m\tau)$ will satisfy approximatively:

$$\frac{\partial u}{\partial t}(x,y,m\tau) \simeq \frac{u^m(x,y)-u^{m-1}(x,y)}{\tau}$$

The time discretization of heat equation (5.6) is as follows, $\forall m = 0,\cdots,[T/\tau]$:

$$\begin{array}{rcll} \frac{u^{m+1}-u^m}{\tau} - \mu\Delta u^{m+1} & = & f^{m+1} & \text{in } \Omega \\ u^0(\mathbf{x}) & = & u_0(\mathbf{x}) & \text{in } \Omega \\ \partial u^{m+1}/\partial n(\mathbf{x}) & = & 0 & \text{on } \partial\Omega \end{array}$$

which is so-called *backward Euler method* for (5.6).

To obtain the variational formulation, multiply with the test function v both sides of the equation:

$$\int_\Omega \{u^{m+1}v - \tau\Delta u^{m+1}v\} = \int_\Omega \{u^m + \tau f^{m+1}\}v$$

By the divergence theorem, we have:

$$\int_\Omega \{u^{m+1}v + \tau\nabla u^{m+1}\cdot\nabla v\} - \int_{\partial\Omega}\tau\left(\partial u^{m+1}/\partial n\right)v = \int_\Omega \{u^m v + \tau f^{m+1}v\}$$

By the boundary condition $\partial u^{m+1}/\partial n = 0$, it follows that:

$$\int_\Omega \{u^{m+1}v + \tau\nabla u^{m+1}\cdot\nabla v\} - \int_\Omega \{u^m v + \tau f^{m+1}v\} = 0 \tag{5.7}$$

Using the identity just above, we can calculate the finite element approximation u_h^m of u^m in a step-by-step manner with respect to t.

Tip: Example

We now solve the following example with the exact solution $u(x,y,t) = tx^4$, $\Omega =]0,1[^2$.

$$\begin{array}{rcll} \frac{\partial u}{\partial t} - \mu\Delta u & = & x^4 - \mu 12tx^2 & \text{in } \Omega\times]0,3[\\ u(x,y,0) & = & 0 & \text{on } \Omega \\ u|_{\partial\Omega} & = & t*x^4 & \end{array}$$

```
// Parameters
real dt = 0.1;
real mu = 0.01;

// Mesh
mesh Th = square(16, 16);

// Fespace
fespace Vh(Th, P1);
Vh u, v, uu, f, g;

// Problem
problem dHeat (u, v)
    = int2d(Th)(
          u*v
        + dt*mu*(dx(u)*dx(v) + dy(u)*dy(v))
    )
    + int2d(Th)(
        - uu*v
        - dt*f*v
    )
    + on(1, 2, 3, 4, u=g)
    ;

// Time loop
real t = 0;
uu = 0;
for (int m = 0; m <= 3/dt; m++){
    // Update
    t = t+dt;
    f = x^4 - mu*t*12*x^2;
    g = t*x^4;
    uu = u;

    // Solve
    dHeat;

    // Plot
    plot(u, wait=true);
    cout << "t=" << t << " - L^2-Error=" << sqrt(int2d(Th)((u-t*x^4)^2)) << endl;
}
```

In the last statement, the L^2-error $\left(\int_\Omega |u - tx^4|^2\right)^{1/2}$ is calculated at $t = m\tau, \tau = 0.1$. At $t = 0.1$, the error is 0.000213269. The errors increase with m and 0.00628589 at $t = 3$.

The iteration of the backward Euler (5.7) is made by *for loop*.

Note: The stiffness matrix in the loop is used over and over again. **FreeFEM** support reuses of stiffness matrix.

5.5.1 Mathematical Theory on Time Difference Approximations.

In this section, we show the advantage of implicit schemes. Let V, H be separable Hilbert space and V is dense in H. Let a be a continuous bilinear form over $V \times V$ with coercivity and symmetry.

Then $\sqrt{a(v,v)}$ become equivalent to the norm $\|v\|$ of V.

Problem Ev(f,Omega): For a given $f \in L^2(0,T;V')$, $u^0 \in H$

$$\begin{array}{rcl} \frac{d}{dt}(u(t),v) + a(u(t),v) & = & (f(t),v) \qquad \forall v \in V, \quad a.e.\, t \in [0,T] \\ u(0) & = & u^0 \end{array}$$

where V' is the dual space of V.

Then, there is an unique solution $u \in L^\infty(0,T;H) \cap L^2(0,T;V)$.

Let us denote the time step by $\tau > 0$, $N_T = [T/\tau]$. For the discretization, we put $u^n = u(n\tau)$ and consider the time difference for each $\theta \in [0,1]$

$$\begin{array}{rcl} \frac{1}{\tau}\left(u_h^{n+1} - u_h^n, \phi_i\right) + a\left(u_h^{n+\theta}, \phi_i\right) & = & \langle f^{n+\theta}, \phi_i \rangle \\ i = 1, & \cdots & , m, \quad n = 0, \cdots, N_T \\ u_h^{n+\theta} & = & \theta u_h^{n+1} + (1-\theta) u_h^n, \\ f^{n+\theta} & = & \theta f^{n+1} + (1-\theta) f^n \end{array}$$

Formula (5.8) is the *forward Euler scheme* if $\theta = 0$, *Crank-Nicolson scheme* if $\theta = 1/2$, the *backward Euler scheme* if $\theta = 1$.

Unknown vectors $u^n = (u_h^1, \cdots, u_h^M)^T$ in

$$u_h^n(x) = u_1^n \phi_1(x) + \cdots + u_m^n \phi_m(x), \quad u_1^n, \cdots, u_m^n \in \mathbb{R}$$

are obtained from solving the matrix

$$\begin{array}{c} (M + \theta\tau A) u^{n+1} = \{M - (1-\theta)\tau A\} u^n + \tau\left\{\theta f^{n+1} + (1-\theta) f^n\right\} \\ M = (m_{ij}), \quad m_{ij} = (\phi_j, \phi_i), \qquad A = (a_{ij}), \quad a_{ij} = a(\phi_j, \phi_i) \end{array}$$

Refer [TABATA1994], pp.70–75 for solvability of (5.8). The stability of (5.8) is in [TABATA1994], Theorem 2.13:

Let $\{\mathcal{T}_h\}_{h\downarrow 0}$ be regular triangulations (see *Regular Triangulation*). Then there is a number $c_0 > 0$ independent of h such that,

$$|u_h^n|^2 \le \begin{cases} \frac{1}{\delta}\left\{|u_h^0|^2 + \tau \sum_{k=0}^{n-1} \|f^{k+\theta}\|_{V_h'}^2\right\} & \theta \in [0, 1/2) \\ |u_h^0|^2 + \tau \sum_{k=0}^{n-1} \|f^{k+\theta}\|_{V_h'}^2 & \theta \in [1/2, 1] \end{cases}$$

if the following are satisfied:

1. When $\theta \in [0, 1/2)$, then we can take a time step τ in such a way that

$$\tau < \frac{2(1-\delta)}{(1-2\theta)c_0^2} h^2$$

   for arbitrary $\delta \in (0,1)$.

2. When $1/2 \le \theta \le 1$, we can take τ arbitrary.

Tip: Example

```
// Parameters
real tau = 0.1; real
theta = 0.;

// Mesh
```

(continues on next page)

(continued from previous page)

```
mesh Th = square(12, 12);

// Fespace
fespace Vh(Th, P1);
Vh u, v, oldU;
Vh f1, f0;

fespace Ph(Th, P0);
Ph h = hTriangle; // mesh sizes for each triangle

// Function
func real f (real t){
    return x^2*(x-1)^2 + t*(-2 + 12*x - 11*x^2 - 2*x^3 + x^4);
}

// File
ofstream out("err02.csv"); //file to store calculations
out << "mesh size = " << h[].max << ", time step = " << tau << endl;
for (int n = 0; n < 5/tau; n++)
    out << n*tau << ",";
out << endl;

// Problem
problem aTau (u, v)
    = int2d(Th)(
          u*v
        + theta*tau*(dx(u)*dx(v) + dy(u)*dy(v) + u*v)
    )
    - int2d(Th)(
          oldU*v
        - (1-theta)*tau*(dx(oldU)*dx(v) + dy(oldU)*dy(v) + oldU*v)
    )
    - int2d(Th)(
          tau*(theta*f1 + (1-theta)*f0)*v
    )
    ;

// Theta loop
while (theta <= 1.0){
    real t = 0;
    real T = 3;
    oldU = 0;
    out << theta << ",";
    for (int n = 0; n < T/tau; n++){
        // Update
        t = t + tau;
        f0 = f(n*tau);
        f1 = f((n+1)*tau);

        // Solve
        aTau;
        oldU = u;

        // Plot
        plot(u);

        // Error
```

(continues on next page)

(continued from previous page)

```
        Vh uex = t*x^2*(1-x)^2; //exact solution = tx^2(1-x)^2
        Vh err = u - uex; // err = FE-sol - exact
        out << abs(err[].max)/abs(uex[].max) << ",";
    }
    out << endl;
    theta = theta + 0.1;
}
```

Fig. 5.14: $\max_{x\in\Omega}|u_h^n(\theta)-u_{ex}(n\tau)|\max_{x\in\Omega}|u_{ex}(n\tau)|atn=0,1,\cdots,29$

We can see in Fig. 5.14 that $u_h^n(\theta)$ become unstable at $\theta=0.4$, and figures are omitted in the case $\theta<0.4$.

5.5.2 Convection

The hyperbolic equation

$$\partial_t u+\alpha\cdot\nabla u=f;\ \text{ for a vector-valued function } \alpha \tag{5.8}$$

appears frequently in scientific problems, for example in the Navier-Stokes equations, in the Convection-Diffusion equation, etc.

In the case of 1-dimensional space, we can easily find the general solution $(x,t)\mapsto u(x,t)=u^0(x-\alpha t)$ of the

following equation, if α is constant,

$$\begin{array}{rcl} \partial_t u + \alpha \partial_x u & = & 0 \\ u(x,0) & = & u^0(x), \end{array} \tag{5.9}$$

because $\partial_t u + \alpha \partial_x u = -\alpha \dot{u}^0 + a\dot{u}^0 = 0$, where $\dot{u}^0 = du^0(x)/dx$.

Even if α is not constant, the construction works on similar principles. One begins with the ordinary differential equation (with the convention that α is prolonged by zero apart from $(0,L) \times (0,T)$):

$$\dot{X}(\tau) = +\alpha(X(\tau),\tau),\ \tau \in (0,t) \quad X(t) = x$$

In this equation τ is the variable and x, t are parameters, and we denote the solution by $X_{x,t}(\tau)$. Then it is noticed that $(x,t) \to v(X(\tau),\tau)$ in $\tau = t$ satisfies the equation

$$\partial_t v + \alpha \partial_x v = \partial_t X \dot{v} + a \partial_x X \dot{v} = 0$$

and by the definition $\partial_t X = \dot{X} = +\alpha$ and $\partial_x X = \partial_x x$ in $\tau = t$, because if $\tau = t$ we have $X(\tau) = x$.

The general solution of (5.9) is thus the value of the boundary condition in $X_{x,t}(0)$, that is to say $u(x,t) = u^0(X_{x,t}(0))$ where $X_{x,t}(0)$ is on the x axis, $u(x,t) = u^0(X_{x,t}(0))$ if $X_{x,t}(0)$ is on the axis of t.

In higher dimension $\Omega \subset R^d,\ d = 2,3$, the equation for the convection is written

$$\partial_t u + \alpha \cdot \nabla u = 0 \text{ in } \Omega \times (0,T)$$

where $\mathbf{a}(x,t) \in \mathbb{R}^d$.

FreeFEM implements the Characteristic-Galerkin method for convection operators. Recall that the equation (5.8) can be discretized as

$$\frac{Du}{Dt} = f \ \text{ i.e. } \frac{du}{dt}(X(t),t) = f(X(t),t) \ \text{ where } \frac{dX}{dt}(t) = \alpha(X(t),t)$$

where D is the total derivative operator. So a good scheme is one step of backward convection by the method of Characteristics-Galerkin

$$\frac{1}{\tau}\left(u^{m+1}(x) - u^m(X^m(x))\right) = f^m(x) \tag{5.10}$$

where $X^m(x)$ is an approximation of the solution at $t = m\tau$ of the ordinary differential equation

$$\frac{d\mathbf{X}}{dt}(t) = \alpha^m(\mathbf{X}(t)), \mathbf{X}((m+1)\tau) = x.$$

where $\alpha^m(x) = (\alpha_1(x,m\tau), \alpha_2(x,m\tau))$. Because, by Taylor's expansion, we have

$$\begin{array}{rcl} u^m(\mathbf{X}(m\tau)) & = & u^m(\mathbf{X}((m+1)\tau)) - \tau \sum_{i=1}^d \frac{\partial u^m}{\partial x_i}(\mathbf{X}((m+1)\tau)) \frac{\partial X_i}{\partial t}((m+1)\tau) + o(\tau) \\ & = & u^m(x) - \tau \alpha^m(x) \cdot \nabla u^m(x) + o(\tau) \end{array}$$

where $X_i(t)$ are the i-th component of $\mathbf{X}(t)$, $u^m(x) = u(x,m\tau)$ and we used the chain rule and $x = \mathbf{X}((m+1)\tau)$. From (5.11), it follows that

$$u^m(X^m(x)) = u^m(x) - \tau \alpha^m(x) \cdot \nabla u^m(x) + o(\tau)$$

Also we apply Taylor's expansion for $t \to u^m(x - \alpha^m(x)t), 0 \le t \le \tau$, then

$$u^m(x - \alpha\tau) = u^m(x) - \tau \alpha^m(x) \cdot \nabla u^m(x) + o(\tau).$$

Putting

$\texttt{convect}(\alpha, -\tau, u^m) \approx u^m\left(x - \alpha^m \tau\right)$

we can get the approximation

$u^m\left(X^m(x)\right) \approx \texttt{convect}\left([a_1^m, a_2^m], -\tau, u^m\right)$ by $X^m \approx x \mapsto x - \tau[a_1^m(x), a_2^m(x)]$

A classical convection problem is that of the "rotating bell" (quoted from [LUCQUIN1998], p.16).

Let Ω be the unit disk centered at 0, with its center rotating with speed $\alpha_1 = y$, $\alpha_2 = -x$. We consider the problem (5.8) with $f = 0$ and the initial condition $u(x, 0) = u^0(x)$, that is, from (5.10)

$u^{m+1}(x) = u^m(X^m(x)) \approx \texttt{convect}(\alpha, -\tau, u^m)$

The exact solution is $u(x, t) = u(\mathbf{X}(t))$ where $\mathbf{X}$ equals x rotated around the origin by an angle $\theta = -t$ (rotate in clockwise). So, if u^0 in a 3D perspective looks like a bell, then u will have exactly the same shape, but rotated by the same amount. The program consists in solving the equation until $T = 2\pi$, that is for a full revolution and to compare the final solution with the initial one; they should be equal.

Tip: Convect

```
// Parameters
real dt = 0.17;

// Mesh
border C(t=0, 2*pi){x=cos(t); y=sin(t);}
mesh Th = buildmesh(C(70));

// Fespace
fespace Vh(Th, P1);
Vh u0;
Vh a1 = -y, a2 = x; //rotation velocity
Vh u;

// Initialization
u = exp(-10*((x-0.3)^2 +(y-0.3)^2));

// Time loop
real t = 0.;
for (int m = 0; m < 2*pi/dt; m++){
    // Update
    t += dt;
    u0 = u;

    // Convect
    u = convect([a1, a2], -dt, u0); //u^{m+1}=u^m(X^m(x))

    // Plot
    plot(u, cmm=" t="+t+", min="+u[].min+", max="+u[].max);
}
```

Note: The scheme `convect` is unconditionally stable, then the bell become lower and lower (the maximum of u^{37} is 0.406 as shown in Fig. 5.15a.

(a) $u^0 = e^{-10((x-0.3)^2+(y-0.3)^2)}$ **(b)** The bell at $t = 6.29$

5.5.3 2D Black-Scholes equation for an European Put option

In mathematical finance, an option on two assets is modeled by a Black-Scholes equations in two space variables, (see for example [WILMOTT1995] or [ACHDOU2005]).

$$\begin{aligned}\partial_t u \quad + \quad & \frac{(\sigma_1 x)^2}{2}\frac{\partial^2 u}{\partial x^2} + \frac{(\sigma_2 y)^2}{2}\frac{\partial^2 u}{\partial y^2} \\ & + \rho x y \frac{\partial^2 u}{\partial x \partial y} + r S_1 \frac{\partial u}{\partial x} + r S_2 \frac{\partial u}{\partial y} - rP = 0\end{aligned}$$

which is to be integrated in $(0,T) \times \mathbb{R}^+ \times \mathbb{R}^+$ subject to, in the case of a put

$$u(x,y,T) = (K - \max(x,y))^+$$

Boundary conditions for this problem may not be so easy to device. As in the one dimensional case the PDE contains boundary conditions on the axis $x_1 = 0$ and on the axis $x_2 = 0$, namely two one dimensional Black-Scholes equations driven respectively by the data $u(0,+\infty,T)$ and $u(+\infty,0,T)$. These will be automatically accounted for because they are embedded in the PDE. So if we do nothing in the variational form (i.e. if we take a Neumann boundary condition at these two axis in the strong form) there will be no disturbance to these. At infinity in one of the variable, as in 1D, it makes sense to impose $u = 0$. We take

$$\sigma_1 = 0.3, \quad \sigma_2 = 0.3, \quad \rho = 0.3, \quad r = 0.05, \quad K = 40, \quad T = 0.5$$

An implicit Euler scheme is used and a mesh adaptation is done every 10 time steps. To have an unconditionally stable scheme, the first order terms are treated by the Characteristic Galerkin method, which, roughly, approximates

$$\frac{\partial u}{\partial t} + a_1 \frac{\partial u}{\partial x} + a_2 \frac{\partial u}{\partial y} \approx \frac{1}{\tau}\left(u^{n+1}(x) - u^n(x - \alpha\tau)\right)$$

Tip: Black-Scholes

```
// Parameters
int m = 30; int L = 80; int LL = 80; int j = 100; real sigx = 0.3; real sigy = 0.3;␣
↪real rho = 0.3; real r = 0.05; real K = 40; real dt = 0.01;

// Mesh
mesh th = square(m, m, [L*x, LL*y]);

// Fespace
fespace Vh(th, P1);
```

(continues on next page)

(continued from previous page)

```
Vh u = max(K-max(x,y),0.);
Vh xveloc, yveloc, v, uold;

// Time loop
for (int n = 0; n*dt <= 1.0; n++){
    // Mesh adaptation
    if (j > 20){
        th = adaptmesh(th, u, verbosity=1, abserror=1, nbjacoby=2,
        err=0.001, nbvx=5000, omega=1.8, ratio=1.8, nbsmooth=3,
        splitpbedge=1, maxsubdiv=5, rescaling=1);
        j = 0;
        xveloc = -x*r + x*sigx^2 + x*rho*sigx*sigy/2;
        yveloc = -y*r + y*sigy^2 + y*rho*sigx*sigy/2;
        u = u;
    }

    // Update
    uold = u;

    // Solve
    solve eq1(u, v, init=j, solver=LU)
        = int2d(th)(
              u*v*(r+1/dt)
            + dx(u)*dx(v)*(x*sigx)^2/2
            + dy(u)*dy(v)*(y*sigy)^2/2
            + (dy(u)*dx(v) + dx(u)*dy(v))*rho*sigx*sigy*x*y/2
        )
        - int2d(th)(
              v*convect([xveloc, yveloc], dt, uold)/dt
        )
        + on(2, 3, u=0)
        ;

    // Update
    j = j+1;
};

// Plot
plot(u, wait=true, value=true);
```

Results are shown on Fig. 5.16a and Fig. 5.16b.

5.6 Navier-Stokes equations

The Stokes equations are: for a given $\mathbf{f} \in L^2(\Omega)^2$:

$$\left.\begin{aligned} -\Delta \mathbf{u} + \nabla p &= \mathbf{f} \\ \nabla \cdot \mathbf{u} &= 0 \end{aligned}\right\} \quad \text{in } \Omega \tag{5.11}$$

where $\mathbf{u} = (u_1, u_2)$ is the velocity vector and p the pressure. For simplicity, let us choose Dirichlet boundary conditions on the velocity, $\mathbf{u} = \mathbf{u}_\Gamma$ on Γ.

In [TEMAM1977], Theorem 2.2, there is a weak form of (5.11):

(a) The adapted triangulation

(b) The level line of the European basquet put option

Find $\mathbf{v} = (v_1, v_2) \in \mathbf{V}(\Omega)$:

$$\mathbf{V}(\Omega) = \{\mathbf{w} \in H_0^1(\Omega)^2 |\ \mathrm{div}\mathbf{w} = 0\}$$

which satisfy:

$$\sum_{i=1}^{2} \int_\Omega \nabla u_i \cdot \nabla v_i = \int_\Omega \mathbf{f} \cdot \mathbf{w} \quad \text{for all } v \in V$$

Here it is used the existence $p \in H^1(\Omega)$ such that $\mathbf{u} = \nabla p$, if:

$$\int_\Omega \mathbf{u} \cdot \mathbf{v} = 0 \quad \text{for all } \mathbf{v} \in V$$

Another weak form is derived as follows: We put:

$$\mathbf{V} = H_0^1(\Omega)^2; \quad W = \left\{ q \in L^2(\Omega) \,\middle|\, \int_\Omega q = 0 \right\}$$

By multiplying the first equation in (5.11) with $v \in V$ and the second with $q \in W$, subsequent integration over Ω, and an application of Green's formula, we have:

$$\begin{array}{rcl} \int_\Omega \nabla \mathbf{u} \cdot \nabla \mathbf{v} - \int_\Omega \mathrm{div}\mathbf{v}\, p & = & \int_\Omega \mathbf{f} \cdot \mathbf{v} \\ \int_\Omega \mathrm{div}\mathbf{u}\, q & = & 0 \end{array}$$

This yields the weak form of (5.11):

Find $(\mathbf{u}, p) \in \mathbf{V} \times W$ such that:

$$\begin{array}{rcl} a(\mathbf{u}, \mathbf{v}) + b(\mathbf{v}, p) & = & (\mathbf{f}, \mathbf{v}) \\ b(\mathbf{u}, q) & = & 0 \end{array}$$

for all $(\mathbf{v}, q) \in V \times W$, where:

$$\begin{array}{rcl} a(\mathbf{u}, \mathbf{v}) & = & \int_\Omega \nabla \mathbf{u} \cdot \nabla \mathbf{v} = \sum_{i=1}^{2} \int_\Omega \nabla u_i \cdot \nabla v_i \\ b(\mathbf{u}, q) & = & -\int_\Omega \mathrm{div}\mathbf{u}\, q \end{array}$$

Now, we consider finite element spaces $\mathbf{V}_h \subset \mathbf{V}$ and $W_h \subset W$, and we assume the following basis functions:

$$\begin{array}{l} \mathbf{V}_h = V_h \times V_h, \quad V_h = \{v_h |\ v_h = v_1\phi_1 + \cdots + v_{M_V}\phi_{M_V}\}, \\ W_h = \{q_h |\ q_h = q_1\varphi_1 + \cdots + q_{M_W}\varphi_{M_W}\} \end{array}$$

The discrete weak form is: Find $(\mathbf{u}_h, p_h) \in \mathbf{V}_h \times W_h$ such that:

$$\begin{array}{lll} a(\mathbf{u}_h, \mathbf{v}_h) + b(\mathbf{v}_h, p) & = (\mathbf{f}, \mathbf{v}_h), & \forall \mathbf{v}_h \in \mathbf{V}_h \\ b(\mathbf{u}_h, q_h) & = 0, & \forall q_h \in W_h \end{array} \tag{5.12}$$

Note: Assume that:

1. There is a constant $\alpha_h > 0$ such that:

$$a(\mathbf{v}_h, \mathbf{v}_h) \geq \alpha \|\mathbf{v}_h\|_{1,\Omega}^2 \quad \text{for all } \mathbf{v}_h \in Z_h$$

   where:

$$Z_h = \{\mathbf{v}_h \in \mathbf{V}_h |\ b(\mathbf{w}_h, q_h) = 0 \quad \text{for all } q_h \in W_h\}$$

2. There is a constant $\beta_h > 0$ such that:

$$\sup_{\mathbf{v}_h \in \mathbf{V}_h} \frac{b(\mathbf{v}_h, q_h)}{\|\mathbf{v}_h\|_{1,\Omega}} \geq \beta_h \|q_h\|_{0,\Omega} \quad \text{for all } q_h \in W_h$$

   Then we have an unique solution $(\mathbf{u}_h, p_h)$ of (5.12) satisfying:

$$\|\mathbf{u} - \mathbf{u}_h\|_{1,\Omega} + \|p - p_h\|_{0,\Omega} \leq C \left(\inf_{\mathbf{v}_h \in \mathbf{V}_h} \|u - v_h\|_{1,\Omega} + \inf_{q_h \in W_h} \|p - q_h\|_{0,\Omega} \right)$$

   with a constant $C > 0$ (see e.g. [ROBERTS1993], Theorem 10.4).

Let us denote that:

$$\begin{array}{rcl} A & = & (A_{ij}),\ A_{ij} = \int_\Omega \nabla\phi_j \cdot \nabla\phi_i \qquad i,j = 1, \cdots, M_{\mathbf{V}} \\ \mathbf{B} & = & (Bx_{ij}, By_{ij}),\ Bx_{ij} = -\int_\Omega \partial\phi_j/\partial x\, \varphi_i \qquad By_{ij} = -\int_\Omega \partial\phi_j/\partial y\, \varphi_i \\ & & \qquad i = 1, \cdots, M_W; j = 1, \cdots, M_V \end{array}$$

then (5.12) is written by:

$$\begin{pmatrix} \mathbf{A} & \mathbf{B}^* \\ \mathbf{B} & 0 \end{pmatrix} \begin{pmatrix} \mathbf{U}_h \\ \{p_h\} \end{pmatrix} = \begin{pmatrix} \mathbf{F}_h \\ 0 \end{pmatrix}$$

where:

$$\mathbf{A} = \begin{pmatrix} A & 0 \\ 0 & A \end{pmatrix} \qquad \mathbf{B}^* = \begin{Bmatrix} Bx^T \\ By^T \end{Bmatrix} \qquad \mathbf{U}_h = \begin{Bmatrix} \{u_{1,h}\} \\ \{u_{2,h}\} \end{Bmatrix} \qquad \mathbf{F}_h = \begin{Bmatrix} \{\int_\Omega f_1\phi_i\} \\ \{\int_\Omega f_2\phi_i\} \end{Bmatrix}$$

Penalty method: This method consists of replacing (5.12) by a more regular problem:

Find $(\mathbf{v}_h^\epsilon, p_h^\epsilon) \in \mathbf{V}_h \times \tilde{W}_h$ satisfying:

$$\begin{array}{lll} a(\mathbf{u}_h^\epsilon, \mathbf{v}_h) + b(\mathbf{v}_h, p_h^\epsilon) & = (\mathbf{f}, \mathbf{v}_h), & \forall \mathbf{v}_h \in \mathbf{V}_h \\ b(\mathbf{u}_h^\epsilon, q_h) - \epsilon(p_h^\epsilon, q_h) & = 0, & \forall q_h \in \tilde{W}_h \end{array} \tag{5.13}$$

where $\tilde{W}_h \subset L^2(\Omega)$. Formally, we have:

$$\text{div}\mathbf{u}_h^\epsilon = \epsilon p_h^\epsilon$$

and the corresponding algebraic problem:

$$\begin{pmatrix} \mathbf{A} & B^* \\ B & -\epsilon I \end{pmatrix} \begin{pmatrix} \mathbf{U}_h^\epsilon \\ \{p_h^\epsilon\} \end{pmatrix} = \begin{pmatrix} \mathbf{F}_h \\ 0 \end{pmatrix}$$

Note: We can eliminate $p_h^\epsilon = (1/\epsilon)BU_h^\epsilon$ to obtain:

$$(A + (1/\epsilon)B^*B)\mathbf{U}_h^\epsilon = \mathbf{F}_h^\epsilon \tag{5.14}$$

Since the matrix $A + (1/\epsilon)B^*B$ is symmetric, positive-definite, and sparse, (5.14) can be solved by known technique. There is a constant $C > 0$ independent of ϵ such that:

$$\|\mathbf{u}_h - \mathbf{u}_h^\epsilon\|_{1,\Omega} + \|p_h - p_h^\epsilon\|_{0,\Omega} \leq C\epsilon$$

(see e.g. [ROBERTS1993], 17.2)

Tip: Cavity

The driven cavity flow problem is solved first at zero Reynolds number (Stokes flow) and then at Reynolds 100. The velocity pressure formulation is used first and then the calculation is repeated with the stream function vorticity formulation.

We solve the driven cavity problem by the penalty method (5.13) where $\mathbf{u}_\Gamma \cdot \mathbf{n} = 0$ and $\mathbf{u}_\Gamma \cdot \mathbf{s} = 1$ on the top boundary and zero elsewhere ($\mathbf{n}$ is the unit normal to Γ, and $\mathbf{s}$ the unit tangent to Γ).

The mesh is constructed by:

```
mesh Th = square(8, 8);
```

We use a classical Taylor-Hood element technique to solve the problem:

The velocity is approximated with the P_2 FE (X_h space), and the pressure is approximated with the P_1 FE (M_h space), where:

$$X_h = \left\{ \mathbf{v} \in H^1(]0,1[^2) \,\middle|\, \forall K \in \mathcal{T}_h \quad v_{|K} \in P_2 \right\}$$

and:

$$M_h = \left\{ v \in H^1(]0,1[^2) \,\middle|\, \forall K \in \mathcal{T}_h \quad v_{|K} \in P_1 \right\}$$

The FE spaces and functions are constructed by:

```
fespace Xh(Th, P2); //definition of the velocity component space
fespace Mh(Th, P1); //definition of the pressure space
Xh u2, v2;
Xh u1, v1;
Mh p, q;
```

The Stokes operator is implemented as a system-solve for the velocity $(u1, u2)$ and the pressure p. The test function for the velocity is $(v1, v2)$ and q for the pressure, so the variational form (5.12) in freefem language is:

```
solve Stokes (u1, u2, p, v1, v2, q, solver=Crout)
    = int2d(Th)(
        (
            dx(u1)*dx(v1)
            + dy(u1)*dy(v1)
            + dx(u2)*dx(v2)
            + dy(u2)*dy(v2)
        )
        - p*q*(0.000001)
        - p*dx(v1) - p*dy(v2)
        - dx(u1)*q - dy(u2)*q
    )
    + on(3, u1=1, u2=0)
    + on(1, 2, 4, u1=0, u2=0)
    ;
```

Each unknown has its own boundary conditions.

If the streamlines are required, they can be computed by finding ψ such that $rot\psi = u$ or better:

$$-\Delta\psi = \nabla \times u$$

```
Xh psi, phi;

solve streamlines (psi, phi)
    = int2d(Th)(
          dx(psi)*dx(phi)
        + dy(psi)*dy(phi)
    )
    + int2d(Th)(
        - phi*(dy(u1) - dx(u2))
    )
    + on(1, 2, 3, 4, psi=0)
    ;
```

Now the Navier-Stokes equations are solved:

$$\frac{\partial u}{\partial t} + u \cdot \nabla u - \nu\Delta u + \nabla p = 0, \nabla \cdot u = 0$$

with the same boundary conditions and with initial conditions $u = 0$.

This is implemented by using the convection operator `convect` for the term $\frac{\partial u}{\partial t} + u \cdot \nabla u$, giving a discretization in time

$$\begin{aligned} \tfrac{1}{\tau}(u^{n+1} - u^n \circ X^n) - \nu\Delta u^{n+1} + \nabla p^{n+1} &= 0, \\ \nabla \cdot u^{n+1} &= 0 \end{aligned}$$

The term $u^n \circ X^n(x) \approx u^n(x - u^n(x)\tau)$ will be computed by the operator `convect`, so we obtain:

```
int i=0;
real alpha=1/dt;
problem NS (u1, u2, p, v1, v2, q, solver=Crout, init=i)
    = int2d(Th)(
          alpha*(u1*v1 + u2*v2)
        + nu * (
              dx(u1)*dx(v1) + dy(u1)*dy(v1)
            + dx(u2)*dx(v2) + dy(u2)*dy(v2)
```

(continues on next page)

(continued from previous page)

```
        )
        - p*q*(0.000001)
        - p*dx(v1) - p*dy(v2)
        - dx(u1)*q - dy(u2)*q
    )
    + int2d(Th)(
        - alpha*convect([up1,up2],-dt,up1)*v1
        - alpha*convect([up1,up2],-dt,up2)*v2
    )
    + on(3, u1=1, u2=0)
    + on(1, 2, 4,u1=0, u2=0)
    ;

// Time loop
for (i = 0; i <= 10; i++){
    // Update
    up1 = u1;
    up2 = u2;

    // Solve
    NS;

    // Plot
    if (!(i % 10))
    plot(coef=0.2, cmm="[u1,u2] and p", p, [u1, u2]);
}
```

Notice that the stiffness matrices are reused (keyword `init=i`)

The complete script is available in *cavity example*.

5.6.1 Uzawa Algorithm and Conjugate Gradients

We solve Stokes problem without penalty. The classical iterative method of Uzawa is described by the algorithm (see e.g. [ROBERTS1993], 17.3, [GLOWINSKI1979], 13 or [GLOWINSKI1985], 13):

- **Initialize:** Let p_h^0 be an arbitrary chosen element of $L^2(\Omega)$.
- **Calculate :math:‘mathbf{u}_h‘:** Once p_h^n is known, $\mathbf{v}_h^n$ is the solution of:

$$\mathbf{u}_h^n = A^{-1}(\mathbf{f}_h - \mathbf{B}^* p_h^n)$$

- **Advance :math:‘p_h‘:** Let p_h^{n+1} be defined by;

$$p_h^{n+1} = p_h^n + \rho_n \mathbf{B}\mathbf{u}_h^n$$

There is a constant $\alpha > 0$ such that $\alpha \leq \rho_n \leq 2$ for each n, then $\mathbf{u}_h^n$ converges to the solution $\mathbf{u}_h$, and then $B\mathbf{v}_h^n \to 0$ as $n \to \infty$ from the *Advance* p_h. This method in general converges quite slowly.

First we define mesh, and the Taylor-Hood approximation. So X_h is the velocity space, and M_h is the pressure space.

Tip: Stokes Uzawa

```
// Mesh
mesh Th = square(10, 10);

// Fespace
fespace Xh(Th, P2);
Xh u1, u2;
Xh bc1, bc2;
Xh b;

fespace Mh(Th, P1);
Mh p;
Mh ppp; //ppp is a working pressure

// Problem
varf bx (u1, q) = int2d(Th)(-(dx(u1)*q));
varf by (u1, q) = int2d(Th)(-(dy(u1)*q));
varf a (u1, u2)
= int2d(Th)(
          dx(u1)*dx(u2)
        + dy(u1)*dy(u2)
    )
    + on(3, u1=1)
    + on(1, 2, 4, u1=0) ;
//remark: put the on(3,u1=1) before on(1,2,4,u1=0)
//because we want zero on intersection

matrix A = a(Xh, Xh, solver=CG);
matrix Bx = bx(Xh, Mh); //B=(Bx, By)
matrix By = by(Xh, Mh);

bc1[] = a(0,Xh); //boundary condition contribution on u1
bc2 = 0; //no boundary condition contribution on u2

//p_h^n -> B A^-1 - B^* p_h^n = -div u_h
//is realized as the function divup
func real[int] divup (real[int] & pp){
    //compute u1(pp)
    b[] = Bx'*pp;
    b[] *= -1;
    b[] += bc1[];
    u1[] = A^-1*b[];
    //compute u2(pp)
    b[] = By'*pp;
    b[] *= -1;
    b[] += bc2[];
    u2[] = A^-1*b[];
    //u^n = (A^-1 Bx^T p^n, By^T p^n)^T
    ppp[] = Bx*u1[]; //ppp = Bx u_1
    ppp[] += By*u2[]; //+ By u_2

    return ppp[] ;
}

// Initialization
p=0; //p_h^0 = 0
LinearCG(divup, p[], eps=1.e-6, nbiter=50); //p_h^{n+1} = p_h^n + B u_h^n
// if n> 50 or |p_h^{n+1} - p_h^n| <= 10^-6, then the loop end
```

(continues on next page)

(continued from previous page)

```
divup(p[]); //compute the final solution

plot([u1, u2], p, wait=1, value=true, coef=0.1);
```

5.6.2 NSUzawaCahouetChabart.edp

In this example we solve the Navier-Stokes equation past a cylinder with the Uzawa algorithm preconditioned by the Cahouet-Chabart method (see [GLOWINSKI2003] for all the details).

The idea of the preconditioner is that in a periodic domain, all differential operators commute and the Uzawa algorithm comes to solving the linear operator $\nabla.((\alpha Id + \nu\Delta)^{-1}\nabla$, where Id is the identity operator. So the preconditioner suggested is $\alpha\Delta^{-1} + \nu Id$.

To implement this, we do:

Tip: NS Uzawa Cahouet Chabart

```
// Parameters
verbosity = 0;
real D = 0.1;
real H = 0.41;
real cx0 = 0.2;
real cy0 = 0.2; //center of cylinder
real xa = 0.15;
real ya = 0.2;
real xe = 0.25;
real ye = 0.2;
int nn = 15;

//TODO
real Um = 1.5; //max velocity (Rey 100)
real nu = 1e-3;

func U1 = 4.*Um*y*(H-y)/(H*H); //Boundary condition
func U2 = 0.;
real T=2;
real dt = D/nn/Um; //CFL = 1
real epspq = 1e-10;
real eps = 1e-6;

// Variables
func Ub = Um*2./3.;
real alpha = 1/dt;
real Rey = Ub*D/nu;
real t = 0.;

// Mesh
border fr1(t=0, 2.2){x=t; y=0; label=1;}
border fr2(t=0, H){x=2.2; y=t; label=2;}
border fr3(t=2.2, 0){x=t; y=H; label=1;}
border fr4(t=H, 0){x=0; y=t; label=1;}
border fr5(t=2*pi, 0){x=cx0+D*sin(t)/2; y=cy0+D*cos(t)/2; label=3;}
mesh Th = buildmesh(fr1(5*nn) + fr2(nn) + fr3(5*nn) + fr4(nn) + fr5(-nn*3));

```

(continues on next page)

(continued from previous page)

```
// Fespace
fespace Mh(Th, [P1]);
Mh p;

fespace Xh(Th, [P2]);
Xh u1, u2;

fespace Wh(Th, [P1dc]);
Wh w; //vorticity

// Macro
macro grad(u) [dx(u), dy(u)] //
macro div(u1, u2) (dx(u1) + dy(u2)) //

// Problem
varf von1 ([u1, u2, p], [v1, v2, q])
    = on(3, u1=0, u2=0)
    + on(1, u1=U1, u2=U2)
    ;

//remark : the value 100 in next varf is manualy fitted, because free outlet.
varf vA (p, q) =
    int2d(Th)(
          grad(p)' * grad(q)
    )
    + int1d(Th, 2)(
          100*p*q
    )
    ;

varf vM (p, q)
    = int2d(Th, qft=qf2pT)(
          p*q
    )
    + on(2, p=0)
    ;

varf vu ([u1], [v1])
    = int2d(Th)(
          alpha*(u1*v1)
        + nu*(grad(u1)' * grad(v1))
    )
    + on(1, 3, u1=0)
    ;

varf vu1 ([p], [v1]) = int2d(Th)(p*dx(v1));
varf vu2 ([p], [v1]) = int2d(Th)(p*dy(v1));

varf vonu1 ([u1], [v1]) = on(1, u1=U1) + on(3, u1=0);
varf vonu2 ([u1], [v1]) = on(1, u1=U2) + on(3, u1=0);

matrix pAM = vM(Mh, Mh, solver=UMFPACK);
matrix pAA = vA(Mh, Mh, solver=UMFPACK);
matrix AU = vu(Xh, Xh, solver=UMFPACK);
matrix B1 = vu1(Mh, Xh);
matrix B2 = vu2(Mh, Xh);

```

(continues on next page)

(continued from previous page)

```
real[int] brhs1 = vonu1(0, Xh);
real[int] brhs2 = vonu2(0, Xh);

varf vrhs1(uu, vv) = int2d(Th)(convect([u1, u2], -dt, u1)*vv*alpha) + vonu1;
varf vrhs2(v2, v1) = int2d(Th)(convect([u1, u2], -dt, u2)*v1*alpha) + vonu2;

// Uzawa function
func real[int] JUzawa (real[int] & pp){
    real[int] b1 = brhs1; b1 += B1*pp;
    real[int] b2 = brhs2; b2 += B2*pp;
    u1[] = AU^-1 * b1;
    u2[] = AU^-1 * b2;
    pp = B1'*u1[];
    pp += B2'*u2[];
    pp = -pp;
    return pp;
}

// Preconditioner function
func real[int] Precon (real[int] & p){
    real[int] pa = pAA^-1*p;
    real[int] pm = pAM^-1*p;
    real[int] pp = alpha*pa + nu*pm;
    return pp;
}

// Initialization
p = 0;

// Time loop
int ndt = T/dt;
for(int i = 0; i < ndt; ++i){
    // Update
    brhs1 = vrhs1(0, Xh);
    brhs2 = vrhs2(0, Xh);

    // Solve
    int res = LinearCG(JUzawa, p[], precon=Precon, nbiter=100, verbosity=10,
    veps=eps);
    assert(res==1);
    eps = -abs(eps);

    // Vorticity
    w = -dy(u1) + dx(u2);
    plot(w, fill=true, wait=0, nbiso=40);

    // Update
    dt = min(dt, T-t);
    t += dt;
    if(dt < 1e-10*T) break;
}

// Plot
plot(w, fill=true, nbiso=40);

// Display
cout << "u1 max = " << u1[].linfty
```

(continues on next page)

(continued from previous page)

```
    << ", u2 max = " << u2[].linfty
    << ", p max = " << p[].max << endl;
```

Warning: Stop test of the conjugate gradient

Because we start from the previous solution and the end the previous solution is close to the final solution, don't take a relative stop test to the first residual, take an absolute stop test (negative here).

Fig. 5.17: The vorticity at Reynolds number 100 a time 2s with the Cahouet-Chabart method.

5.7 Variational Inequality

We present, a classical example of variational inequality.

Let us denote $\mathcal{C} = \{u \in H_0^1(\Omega), u \leq g\}$

The problem is:

$$u = arg \min_{u \in \mathcal{C}} J(u) = \frac{1}{2} \int_\Omega \nabla u . \nabla u - \int_\Omega f u$$

where f and g are given function.

The solution is a projection on the convex $\mathcal{C}$ of $f^\star$ for the scalar product $((v, w)) = \int_\Omega \nabla v . \nabla w$ of $H_0^1(\Omega)$ where $f^\star$ is solution of:

$$(f^\star, v) = \int_\Omega f v, \forall v \in H_0^1(`\Omega)$$

The projection on a convex satisfy clearly $\forall v \in \mathcal{C}, \quad ((u - v, u - \tilde{f})) \leq 0$, and after expanding, we get the classical inequality:

$$\forall v \in \mathcal{C}, \quad \int_\Omega \nabla(u - v) \nabla u \leq \int_\Omega (u - v) f$$

We can also rewrite the problem as a saddle point problem:

Find λ, u such that:

$$\max_{\lambda \in L^2(\Omega), \lambda \geq 0} \min_{u \in H_0^1(\Omega)} \mathcal{L}(u, \lambda) = \frac{1}{2} \int_\Omega \nabla u . \nabla u - \int_\Omega f u + \int_\Omega \lambda (u - g)^+$$

where $((u - g)^+ = max(0, u - g)$.

This saddle point problem is equivalent to find u, λ such that:

$$\begin{cases} \int_\Omega \nabla u.\nabla v + \lambda v^+ \, d\omega = \int_\Omega f u, & \forall v \in H_0^1(\Omega) \\ \int_\Omega \mu(u-g)^+ = 0, & \forall \mu \in L^2(\Omega), \mu \geq 0, \lambda \geq 0, \end{cases}$$

An algorithm to solve the previous problem is:

1. k=0, and choose λ_0 belong $H^{-1}(\Omega)$
2. Loop on $k = 0, ...$
   - set $\mathcal{I}_k = \{x \in \Omega / \lambda_k + c * (u_{k+1} - g) \leq 0\}$
   - $V_{g,k+1} = \{v \in H_0^1(\Omega) / v = g \text{ on } I_k\}$,
   - $V_{0,k+1} = \{v \in H_0^1(\Omega) / v = 0 \text{ on } I_k\}$,
   - Find $u_{k+1} \in V_{g,k+1}$ and $\lambda_{k+1} \in H^{-1}(\Omega)$ such that

$$\begin{cases} \int_\Omega \nabla u_{k+1}.\nabla v_{k+1} \, d\omega = \int_\Omega f v_{k+1}, & \forall v_{k+1} \in V_{0,k+1} \\ < \lambda_{k+1}, v >= \int_\Omega \nabla u_{k+1}.\nabla v - f v \, d\omega \end{cases}$$

   where $<,>$ is the duality bracket between $H_0^1(\Omega)$ and $H^{-1}(\Omega)$, and c is a penalty constant (large enough).

You can find all the mathematics about this algorithm in [ITO2003].

Now how to do that in **FreeFEM**? The full example is:

Tip: Variational inequality

```
load "medit"

// Parameters
real eps = 1e-5;
real c = 1000; //penalty parameter of the algoritm
real tgv = 1e30; //a huge value for exact penalization
func f = 1; //right hand side function
func fd = 0; //Dirichlet boundary condition function

// Mesh
mesh Th = square(20, 20);

// Fespace
fespace Vh(Th, P1);
int n = Vh.ndof; //number of degree of freedom
Vh uh, uhp; //u^n+1 and u^n
Vh Ik; //to define the set where the containt is reached.
Vh g = 0.05; //discret function g
Vh lambda = 0;

// Problem
varf a (uh, vh)
    = int2d(Th)(
          dx(uh)*dx(vh)
        + dy(uh)*dy(vh)
    )
```

(continues on next page)

(continued from previous page)

```
    - int2d(Th)(
          f*vh
    )
    + on(1, 2, 3, 4, uh=fd)
    ;

//the mass Matrix construction
varf vM (uh, vh) = int2d(Th)(uh*vh);

//two versions of the matrix of the problem
matrix A = a(Vh, Vh, tgv=tgv, solver=CG); //one changing
matrix AA = a(Vh, Vh, solver=CG); //one for computing residual

matrix M = vM(Vh, Vh); //to do a fast computing of L^2 norm : sqrt(u'*(w=M*u))

real[int] Aiin(n);
real[int] Aii = A.diag; //get the diagonal of the matrix
real[int] rhs = a(0, Vh, tgv=tgv);

// Initialization
Ik = 0;
uhp = -tgv;

// Loop
for(int iter = 0; iter < 100; ++iter){
    // Update
    real[int] b = rhs; //get a copy of the Right hand side
    real[int] Ak(n); //the complementary of Ik ( !Ik = (Ik-1))
    Ak = 1.; Ak -= Ik[];
    //adding new locking condition on b and on the diagonal if (Ik ==1 )
    b = Ik[] .* g[]; b *= tgv; b -= Ak .* rhs;
    Aiin = Ik[] * tgv; Aiin += Ak .* Aii; //set Aii= tgv i in Ik
    A.diag = Aiin; //set the matrix diagonal
    set(A, solver=CG); //important to change preconditioning for solving

    // Solve
    uh[] = A^-1* b; //solve the problem with more locking condition

    // Residual
    lambda[] = AA * uh[]; //compute the residual (fast with matrix)
    lambda[] += rhs; //remark rhs = -\int f v

    Ik = (lambda + c*( g- uh)) < 0.; //the new locking value

    // Plot
    plot(Ik, wait=true, cmm=" lock set ", value=true, fill=true);
    plot(uh, wait=true, cmm="uh");

    // Error
    //trick to compute L^2 norm of the variation (fast method)
    real[int] diff(n), Mdiff(n);
    diff = uh[] - uhp[];
    Mdiff = M*diff;
    real err = sqrt(Mdiff'*diff);
    cout << "|| u_{k=1} - u_{k} ||_2 = " << err << endl;

    // Stop test
```

(continues on next page)

(continued from previous page)

```
    if(err < eps) break;

    // Update
    uhp[] = uh[];
}

// Plot
medit("uh", Th, uh);
```

Note: As you can see on this example, some vector, or matrix operator are not implemented so a way is to skip the expression and we use operator +=, −= to merge the result.

5.8 Domain decomposition

We present three classic examples of domain decomposition technique: first, Schwarz algorithm with overlapping, second Schwarz algorithm without overlapping (also call Shur complement), and last we show to use the conjugate gradient to solve the boundary problem of the Shur complement.

5.8.1 Schwarz overlapping

To solve:

$$-\Delta u = f, \text{ in } \Omega = \Omega_1 \cup \Omega_2 \quad u|_\Gamma = 0$$

the Schwarz algorithm runs like this:

$$\begin{array}{rcll} -\Delta u_1^{n+1} & = & f \text{ in } \Omega_1 & u_1^{n+1}|_{\Gamma_1} = u_2^n \\ -\Delta u_2^{n+1} & = & f \text{ in } \Omega_2 & u_2^{n+1}|_{\Gamma_2} = u_1^n \end{array}$$

where Γ_i is the boundary of Ω_i and on the condition that $\Omega_1 \cap \Omega_2 \neq \emptyset$ and that u_i are zero at iteration 1.

Here we take Ω_1 to be a quadrangle, Ω_2 a disk and we apply the algorithm starting from zero.

Tip: Schwarz overlapping

```
// Parameters
int inside =2; //inside boundary
int outside = 1; //outside boundary
int n = 4;

// Mesh
border a(t=1, 2){x=t; y=0; label=outside;}
border b(t=0, 1){x=2; y=t; label=outside;}
border c(t=2, 0){x=t; y=1; label=outside;}
border d(t=1, 0){x=1-t; y=t; label=inside;}
border e(t=0, pi/2){x=cos(t); y=sin(t); label=inside;}
border e1(t=pi/2, 2*pi){x=cos(t); y=sin(t); label=outside;}
mesh th = buildmesh(a(5*n) + b(5*n) + c(10*n) + d(5*n));
mesh TH = buildmesh(e(5*n) + e1(25*n));
```

(continues on next page)

Fig. 5.18: The 2 overlapping mesh `TH` and `th`

(continued from previous page)

```
plot(th, TH, wait=true); //to see the 2 meshes

// Fespace
fespace vh(th, P1);
vh u=0, v;

fespace VH(TH, P1);
VH U, V;

// Problem
int i = 0;
problem PB (U, V, init=i, solver=Cholesky)
   = int2d(TH)(
         dx(U)*dx(V)
       + dy(U)*dy(V)
   )
   + int2d(TH)(
       - V
   )
   + on(inside, U=u)
   + on(outside, U=0)
   ;

problem pb (u, v, init=i, solver=Cholesky)
   = int2d(th)(
         dx(u)*dx(v)
       + dy(u)*dy(v)
   )
   + int2d(th)(
       - v
   )
```

(continues on next page)

(a) Isovalues of the solution at iteration 0 **(b)** Isovalues of the solution at iteration 0

Fig. 5.19: Schwarz overlapping

(continued from previous page)

```
   + on(inside, u=U)
   + on(outside, u=0)
   ;

// Calculation loop
for (i = 0 ; i < 10; i++){
   // Solve
   PB;
   pb;

   // Plot
   plot(U, u, wait=true);
}
```

5.8.2 Schwarz non overlapping Scheme

To solve:

$$-\Delta u = f \text{ in } \Omega = \Omega_1 \cup \Omega_2 \quad u|_\Gamma = 0$$

the Schwarz algorithm for domain decomposition without overlapping runs like this

Let introduce Γ_i is common the boundary of Ω_1 and Ω_2 and $\Gamma_e^i = \partial\Omega_i \setminus \Gamma_i$.

The problem find λ such that $(u_1|_{\Gamma_i} = u_2|_{\Gamma_i})$ where u_i is solution of the following Laplace problem:

$$-\Delta u_i = f \text{ in } \Omega_i \quad u_i|_{\Gamma_i} = \lambda \quad u_i|_{\Gamma_e^i} = 0$$

To solve this problem we just make a loop with upgrading λ with

$$\lambda = \lambda \pm \frac{(u_1 - u_2)}{2}$$

Fig. 5.20: The two none overlapping mesh `TH` and `th`

where the sign $+$ or $-$ of $\pm$ is choose to have convergence.

Tip: Schwarz non-overlapping

```
// Parameters
int inside = 2; int outside = 1; int n = 4;

// Mesh
border a(t=1, 2){x=t; y=0; label=outside;};
border b(t=0, 1){x=2; y=t; label=outside;};
border c(t=2, 0){x=t; y=1; label=outside;};
border d(t=1, 0){x=1-t; y=t; label=inside;};
border e(t=0, 1){x=1-t; y=t; label=inside;};
border e1(t=pi/2, 2*pi){x=cos(t); y=sin(t); label=outside;};
mesh th = buildmesh(a(5*n) + b(5*n) + c(10*n) + d(5*n));
mesh TH = buildmesh(e(5*n) + e1(25*n));
plot(th, TH, wait=true);

// Fespace
fespace vh(th, P1);
vh u=0, v;
vh lambda=0;

fespace VH(TH, P1);
VH U, V;

// Problem
int i = 0;
problem PB (U, V, init=i, solver=Cholesky)
    = int2d(TH)(
          dx(U)*dx(V)
        + dy(U)*dy(V)
```

(continues on next page)

(continued from previous page)

```
    )
    + int2d(TH)(
        - V
    )
    + int1d(TH, inside)(
          lambda*V
    )
    + on(outside, U= 0 )
    ;

problem pb (u, v, init=i, solver=Cholesky)
    = int2d(th)(
          dx(u)*dx(v)
        + dy(u)*dy(v)
    )
    + int2d(th)(
        - v
    )
    + int1d(th, inside)(
        - lambda*v
    )
    + on(outside, u=0)
    ;

for (i = 0; i < 10; i++){
    // Solve
    PB;
    pb;
    lambda = lambda - (u-U)/2;

    // Plot
    plot(U,u,wait=true);
}

// Plot
plot(U, u);
```

5.8.3 Schwarz conjuguate gradient

To solve $-\Delta u = f$ in $\Omega = \Omega_1 \cup \Omega_2 \quad u|_\Gamma = 0$ the Schwarz algorithm for domain decomposition without overlapping runs like this

Let introduce Γ_i is common the boundary of Ω_1 and Ω_2 and $\Gamma_e^i = \partial\Omega_i \setminus \Gamma_i$.

The problem find λ such that $(u_1|_{\Gamma_i} = u_2|_{\Gamma_i})$ where u_i is solution of the following Laplace problem:

$$-\Delta u_i = f \text{ in } \Omega_i \quad u_i|_{\Gamma_i} = \lambda \quad u_i|_{\Gamma_e^i} = 0$$

The version of this example uses the Shur complement. The problem on the border is solved by a conjugate gradient method.

Tip: Schwarz conjugate gradient

First, we construct the two domains:

(a) Isovalues of the solution at iteration 0 without overlapping

(b) Isovalues of the solution at iteration 9 without overlapping

```
// Parameters
int inside = 2; int outside = 1; int n = 4;

// Mesh
border Gamma1(t=1, 2){x=t; y=0; label=outside;}
border Gamma2(t=0, 1){x=2; y=t; label=outside;}
border Gamma3(t=2, 0){x=t; y=1; label=outside;}
border GammaInside(t=1, 0){x=1-t; y=t; label=inside;}
border GammaArc(t=pi/2, 2*pi){x=cos(t); y=sin(t); label=outside;}
mesh Th1 = buildmesh(Gamma1(5*n) + Gamma2(5*n) + GammaInside(5*n) + Gamma3(5*n));
mesh Th2 = buildmesh(GammaInside(-5*n) + GammaArc(25*n));
plot(Th1, Th2);
```

Now, define the finite element spaces:

```
// Fespace
fespace Vh1(Th1, P1);
Vh1 u1, v1;
Vh1 lambda;
Vh1 p=0;

fespace Vh2(Th2,P1);
Vh2 u2, v2;
```

Note: It is impossible to define a function just on a part of boundary, so the λ function must be defined on the all domain Ω_1 such as:

```
Vh1 lambda;
```

The two Poisson's problems:

```
problem Pb1 (u1, v1, init=i, solver=Cholesky)
    = int2d(Th1)(
          dx(u1)*dx(v1)
        + dy(u1)*dy(v1)
    )
    + int2d(Th1)(
        - v1
    )
    + int1d(Th1, inside)(
          lambda*v1
    )
    + on(outside, u1=0)
    ;

problem Pb2 (u2, v2, init=i, solver=Cholesky)
    = int2d(Th2)(
          dx(u2)*dx(v2)
        + dy(u2)*dy(v2)
    )
    + int2d(Th2)(
        - v2
    )
    + int1d(Th2, inside)(
        - lambda*v2
    )
    + on(outside, u2=0)
    ;
```

And, we define a border matrix, because the λ function is none zero inside the domain Ω_1:

```
varf b(u2, v2, solver=CG) = int1d(Th1, inside)(u2*v2);
matrix B = b(Vh1, Vh1, solver=CG);
```

The boundary problem function,

$$\lambda \longrightarrow \int_{\Gamma_i} (u_1 - u_2) v_1$$

```
// Boundary problem function
func real[int] BoundaryProblem (real[int] &l){
   lambda[] = l; //make FE function form l
   Pb1;
   Pb2;
   i++; //no refactorization i != 0
   v1 = -(u1-u2);
   lambda[] = B*v1[];
   return lambda[];
}
```

Note: The difference between the two notations `v1` and `v1[]` is: `v1` is the finite element function and `v1[]` is the vector in the canonical basis of the finite element function `v1`.

```
// Solve
real cpu=clock();
```

(continues on next page)

(continued from previous page)

```
LinearCG(BoundaryProblem, p[], eps=1.e-6, nbiter=100);
//compute the final solution, because CG works with increment
BoundaryProblem(p[]); //solve again to have right u1, u2

// Display & Plot
cout << " -- CPU time schwarz-gc:" << clock()-cpu << endl;
plot(u1, u2);
```

5.9 Fluid-structure coupled problem

This problem involves the Lamé system of elasticity and the Stokes system for viscous fluids with velocity $\mathbf{u}$ and pressure p:

$$\begin{array}{rcll} -\Delta\mathbf{u}+\nabla p & = & 0 & \text{in } \Omega \\ \nabla\cdot\mathbf{u} & = & 0 & \text{in } \Omega \\ \mathbf{u} & = & \mathbf{u}_\Gamma & \text{on } \Gamma=\partial\Omega \end{array}$$

where u_Γ is the velocity of the boundaries. The force that the fluid applies to the boundaries is the normal stress

$$\mathbf{h}=(\nabla\mathbf{u}+\nabla\mathbf{u}^T)\mathbf{n}-p\mathbf{n}$$

Elastic solids subject to forces deform: a point in the solid at (x,y) goes to (X,Y) after. When the displacement vector $\mathbf{v}=(v_1,v_2)=(X-x,Y-y)$ is small, Hooke's law relates the stress tensor σ inside the solid to the deformation tensor ϵ:

$$\sigma_{ij}=\lambda\delta_{ij}\nabla.\mathbf{v}+2\mu\epsilon_{ij},\ \epsilon_{ij}=\frac{1}{2}(\frac{\partial v_i}{\partial x_j}+\frac{\partial v_j}{\partial x_i})$$

where δ is the Kronecker symbol and where λ, μ are two constants describing the material mechanical properties in terms of the modulus of elasticity, and Young's modulus.

The equations of elasticity are naturally written in variational form for the displacement vector $v(x)\in V$ as:

$$\int_\Omega[2\mu\epsilon_{ij}(\mathbf{v})\epsilon_{ij}(\mathbf{w})+\lambda\epsilon_{ii}(v)\epsilon_{jj}(\mathbf{w})]=\int_\Omega\mathbf{g}\cdot\mathbf{w}+\int_\Gamma\mathbf{h}\cdot\mathbf{w},\forall\mathbf{w}\in V$$

The data are the gravity force $\mathbf{g}$ and the boundary stress $\mathbf{h}$.

Tip: Fluide-structure In our example, the Lamé system and the Stokes system are coupled by a common boundary on which the fluid stress creates a displacement of the boundary and hence changes the shape of the domain where the Stokes problem is integrated. The geometry is that of a vertical driven cavity with an elastic lid. The lid is a beam with weight so it will be deformed by its own weight and by the normal stress due to the fluid reaction. The cavity is the 10×10 square and the lid is a rectangle of height $l=2$.

A beam sits on a box full of fluid rotating because the left vertical side has velocity one. The beam is bent by its own weight, but the pressure of the fluid modifies the bending.

The bending displacement of the beam is given by (uu,vv) whose solution is given as follows.

```
// Parameters
int bottombeam = 2; //label of bottombeam
real E = 21.5;
real sigma = 0.29;
```

(continues on next page)

(continued from previous page)

```
real gravity = -0.05;
real coef = 0.2;

// Mesh (solid)
border a(t=2, 0){x=0; y=t; label=1;}
border b(t=0, 10){x=t; y=0; label=bottombeam;}
border c(t=0, 2){x=10; y=t; label=1;}
border d(t=0, 10){x=10-t; y=2; label=3;}
mesh th = buildmesh(b(20) + c(5) + d(20) + a(5));

// Fespace (solid)
fespace Vh(th, P1);
Vh uu, w, vv, s;

// Macro
real sqrt2 = sqrt(2.);
macro epsilon(u1, u2) [dx(u1), dy(u2), (dy(u1)+dx(u2))/sqrt2] //
macro div(u1, u2) (dx(u1) + dy(u2)) //

// Problem (solid)
real mu = E/(2*(1+sigma));
real lambda = E*sigma/((1+sigma)*(1-2*sigma));
solve Elasticity([uu, vv], [w, s])
    = int2d(th)(
          lambda*div(w,s)*div(uu,vv)
        + 2.*mu*(epsilon(w,s)'*epsilon(uu,vv))
    )
    + int2d(th)(
        - gravity*s
    )
    + on(1, uu=0, vv=0)
    ;

plot([uu, vv], wait=true);
mesh th1 = movemesh(th, [x+uu, y+vv]);
plot(th1, wait=true);
```

Then Stokes equation for fluids at low speed are solved in the box below the beam, but the beam has deformed the box (see border h):

```
// Mesh (fluid)
border e(t=0, 10){x=t; y=-10; label= 1;}
border f(t=0, 10){x=10; y=-10+t ; label= 1;}
border g(t=0, 10){x=0; y=-t; label= 2;}
border h(t=0, 10){x=t; y=vv(t,0)*( t>=0.001 )*(t <= 9.999); label=3;}
mesh sh = buildmesh(h(-20) + f(10) + e(10) + g(10));
plot(sh, wait=true);
```

We use the Uzawa conjugate gradient to solve the Stokes problem like in *Navier-Stokes equations*.

```
// Fespace (fluid)
fespace Xh(sh, P2);
Xh u1, u2;
Xh bc1;
Xh brhs;
Xh bcx=0, bcy=1;
```

(continues on next page)

(continued from previous page)

```

fespace Mh(sh, P1);
Mh p, ppp;

// Problem (fluid)
varf bx (u1, q) = int2d(sh)(-(dx(u1)*q));
varf by (u1, q) = int2d(sh)(-(dy(u1)*q));
varf Lap (u1, u2)
    = int2d(sh)(
          dx(u1)*dx(u2)
        + dy(u1)*dy(u2)
    )
    + on(2, u1=1)
    + on(1, 3, u1=0)
    ;

bc1[] = Lap(0, Xh);

matrix A = Lap(Xh, Xh, solver=CG);
matrix Bx = bx(Xh, Mh);
matrix By = by(Xh, Mh);


func real[int] divup (real[int] & pp){
    int verb = verbosity;
    verbosity = 0;
    brhs[] = Bx'*pp;
    brhs[] += bc1[] .*bcx[];
    u1[] = A^-1*brhs[];
    brhs[] = By'*pp;
    brhs[] += bc1[] .*bcy[];
    u2[] = A^-1*brhs[];
    ppp[] = Bx*u1[];
    ppp[] += By*u2[];
    verbosity = verb;
    return ppp[];
}
```

do a loop on the two problems

```
// Coupling loop
for(int step = 0; step < 10; ++step){
    // Solve (fluid)
    LinearCG(divup, p[], eps=1.e-3, nbiter=50);
    divup(p[]);
```

Now the beam will feel the stress constraint from the fluid:

```
// Forces
Vh sigma11, sigma22, sigma12;
Vh uu1=uu, vv1=vv;

sigma11([x+uu, y+vv]) = (2*dx(u1) - p);
sigma22([x+uu, y+vv]) = (2*dy(u2) - p);
sigma12([x+uu, y+vv]) = (dx(u1) + dy(u2));
```

which comes as a boundary condition to the PDE of the beam:

```
// Solve (solid)
solve Elasticity2 ([uu, vv], [w, s], init=step)
= int2d(th)(
      lambda*div(w,s)*div(uu,vv)
    + 2.*mu*(epsilon(w,s)'*epsilon(uu,vv))
)
+ int2d(th)(
    - gravity*s
)
+ int1d(th, bottombeam)(
    - coef*(sigma11*N.x*w + sigma22*N.y*s + sigma12*(N.y*w+N.x*s))
)
+ on(1, uu=0, vv=0)
;

// Plot
plot([uu, vv], wait=1);

// Error
real err = sqrt(int2d(th)((uu-uu1)^2 + (vv-vv1)^2));
cout << "Erreur L2 = " << err << endl;
```

Notice that the matrix generated by `Elasticity2` is reused (see *init=i*). Finally we deform the beam:

```
// Movemesh
th1 = movemesh(th, [x+0.2*uu, y+0.2*vv]);
plot(th1, wait=true);
```

Fluid velocity and pressure, displacement vector of the structure and displaced geometry in the fluid-structure interaction of a soft side and a driven cavity are shown Fig. 5.22, Fig. 5.23a and Fig. 5.23b

Fig. 5.22: Velocity and pressure

(a) Displacement

(b) Moved mesh

5.10 Transmission problem

Consider an elastic plate whose displacement change vertically, which is made up of three plates of different materials, welded on each other.

Let Ω_i, $i = 1, 2, 3$ be the domain occupied by i-th material with tension μ_i (see *Soap film*).

The computational domain Ω is the interior of $\overline{\Omega_1} \cup \overline{\Omega_2} \cup \overline{\Omega_3}$. The vertical displacement $u(x, y)$ is obtained from:

$$\begin{array}{rcll} -\mu_i \Delta u & = & f & \text{in } \Omega_i \\ \mu_i \partial_n u|_{\Gamma_i} & = & -\mu_j \partial_n u|_{\Gamma_j} & \text{on } \overline{\Omega_i} \cap \overline{\Omega_j} \text{ if } 1 \le i < j \le 3 \end{array} \tag{5.15}$$

where $\partial_n u|_{\Gamma_i}$ denotes the value of the normal derivative $\partial_n u$ on the boundary Γ_i of the domain Ω_i.

By introducing the characteristic function χ_i of Ω_i, that is:

$$\chi_i(x) = 1 \ \text{ if } x \in \Omega_i; \ \chi_i(x) = 0 \ \text{ if } x \notin \Omega_i$$

we can easily rewrite (5.15) to the weak form. Here we assume that $u = 0$ on $\Gamma = \partial\Omega$.

Transmission problem: For a given function f, find u such that:

$$a(u, v) = \ell(f, v) \text{ for all } v \in H_0^1(\Omega)$$

$$\begin{array}{rcl} a(u, v) & = & \int_\Omega \mu \nabla u \cdot \nabla v \\ \ell(f, v) & = & \int_\Omega f v \end{array}$$

where $\mu = \mu_1\chi_1 + \mu_2\chi_2 + \mu_3\chi_3$. Here we notice that μ become the discontinuous function.

This example explains the definition and manipulation of *region*, i.e. sub-domains of the whole domain. Consider this L-shaped domain with 3 diagonals as internal boundaries, defining 4 sub-domains:

```
// Mesh
border a(t=0, 1){x=t; y=0;};
border b(t=0, 0.5){x=1; y=t;};
border c(t=0, 0.5){x=1-t; y=0.5;};
border d(t=0.5, 1){x=0.5; y=t;};
border e(t=0.5, 1){x=1-t; y=1;};
border f(t=0, 1){x=0; y=1-t;};
border i1(t=0, 0.5){x=t; y=1-t;};
border i2(t=0, 0.5){x=t; y=t;};
border i3(t=0, 0.5){x=1-t; y=t;};
mesh th = buildmesh(a(6) + b(4) + c(4) +d(4) + e(4)
    + f(6) + i1(6) + i2(6) + i3(6));

// Fespace
fespace Ph(th, P0); //constant discontinuous functions / element
Ph reg=region; //defined the P0 function associated to region number

// Plot
plot(reg, fill=true, wait=true, value=true);
```

Fig. 5.24: The function `reg`

`region` is a keyword of **FreeFEM** which is in fact a variable depending of the current position (is not a function today, use `Ph reg=region;` to set a function). This variable value returned is the number of the sub-domain of the current position. This number is defined by `buildmesh` which scans while building the mesh all its connected component.

So to get the number of a region containing a particular point one does:

```
// Characteristic function
int nupper = reg(0.4, 0.9); //get the region number of point (0.4,0.9)
int nlower = reg(0.9, 0.1); //get the region number of point (0.4,0.1)
cout << "nlower = " <<  nlower << ", nupper = " << nupper<< endl;
Ph nu = 1 + 5*(region==nlower) + 10*(region==nupper);

// Plot
plot(nu, fill=true,wait=true);
```

This is particularly useful to define discontinuous functions such as might occur when one part of the domain is copper and the other one is iron, for example.

We this in mind we proceed to solve a Laplace equation with discontinuous coefficients (ν is 1, 6 and 11 below).

```
// Problem
solve lap (u, v)
    = int2d(th)(
          nu*(dx(u)*dx(v) + dy(u)*dy(v))
    )
    + int2d(th)(
        - 1*v
    )
    + on(a, b, c, d, e, f, u=0)
    ;

// Plot
plot(u);
```

Fig. 5.25: The function `nu`

Fig. 5.26: The isovalue of the solution u

5.11 Free boundary problems

The domain Ω is defined with:

```
// Parameters
real L = 10; //length
real hr = 2.1; //left height
real hl = 0.35; //right height
int n = 4;

// Mesh
border a(t=0, L){x=t; y=0;}; //bottom: Gamma_a
border b(t=0, hr){x=L; y=t;}; //right: Gamma_b
border f(t=L, 0){x=t; y=t*(hr-hl)/L+hl;}; //free surface: Gamma_f
border d(t=hl, 0){x=0; y=t;}; // left: Gamma_d
mesh Th = buildmesh(a(10*n) + b(6*n) + f(8*n) + d(3*n));
plot(Th);
```

Fig. 5.27: The mesh of the domain Ω

The free boundary problem is:

Find u and Ω such that:

$$\left\{\begin{array}{rcll} -\Delta u & = & 0 & \text{in } \Omega \\ u & = & y & \text{on } \Gamma_b \\ \frac{\partial u}{\partial n} & = & 0 & \text{on } \Gamma_d \cup \Gamma_a \\ \frac{\partial u}{\partial n} & = & \frac{q}{K} n_x & \text{on } \Gamma_f \\ u & = & y & \text{on } \Gamma_f \end{array}\right.$$

We use a fixed point method;

$\Omega^0 = \Omega$

In two step, fist we solve the classical following problem:

$$\left\{\begin{array}{rcll} -\Delta u & = & 0 & \text{in } \Omega^n \\ u & = & y & \text{on } \Gamma_b^n \\ \frac{\partial u}{\partial n} & = & 0 & \text{on } \Gamma_d^n \cup \Gamma_a^n \\ u & = & y & \text{on } \Gamma_f^n \end{array}\right.$$

The variational formulation is:

Find u on $V = H^1(\Omega^n)$, such than $u = y$ on Γ_b^n and Γ_f^n

$$\int_{\Omega^n} \nabla u \nabla u' = 0, \ \forall u' \in V \text{ with } u' = 0 \text{ on } \Gamma_b^n \cup \Gamma_f^n$$

And secondly to construct a domain deformation $\mathcal{F}(x,y) = [x, y - v(x,y)]$ where v is solution of the following problem:

$$\left\{\begin{array}{rcll} -\Delta v & = & 0 & \text{in } \Omega^n \\ v & = & 0 & \text{on } \Gamma_a^n \\ \frac{\partial v}{\partial n} & = & 0 & \text{on } \Gamma_b^n \cup \Gamma_d^n \\ \frac{\partial v}{\partial n} & = & \frac{\partial u}{\partial n} - \frac{q}{K} n_x & \text{on } \Gamma_f^n \end{array}\right.$$

The variational formulation is:

Find v on V, such than $v = 0$ on Γ_a^n:

$$\int_{\Omega^n} \nabla v \nabla v' = \int_{\Gamma_f^n} (\frac{\partial u}{\partial n} - \frac{q}{K} n_x) v', \quad \forall v' \in V \text{ with } v' = 0 \text{ on } \Gamma_a^n$$

Finally the new domain $\Omega^{n+1} = \mathcal{F}(\Omega^n)$

Tip: Free boundary

The **FreeFEM** implementation is:

```
// Parameters
real L = 10; //length
real hr = 2.1; //left height
real hl = 0.35; //right height
int n = 4;

real q = 0.02; //incoming flow
real K = 0.5; //permeability

// Mesh
border a(t=0, L){x=t; y=0;}; //bottom: Gamma_a
border b(t=0, hr){x=L; y=t;}; //right: Gamma_b
border f(t=L, 0){x=t; y=t*(hr-hl)/L+hl;}; //free surface: Gamma_f
border d(t=hl, 0){x=0; y=t;}; // left: Gamma_d
mesh Th = buildmesh(a(10*n) + b(6*n) + f(8*n) + d(3*n));
plot(Th);

// Fespace
fespace Vh(Th, P1);
Vh u, v;
Vh uu, vv;

// Problem
problem Pu (u, uu, solver=CG)
    = int2d(Th)(
          dx(u)*dx(uu)
        + dy(u)*dy(uu)
    )
    + on(b, f, u=y)
    ;

problem Pv (v, vv, solver=CG)
    = int2d(Th)(
          dx(v)*dx(vv)
        + dy(v)*dy(vv)
    )
    + on(a, v=0)
```

(continues on next page)

(continued from previous page)

```
    + int1d(Th, f)(
          vv*((q/K)*N.y - (dx(u)*N.x + dy(u)*N.y))
    )
    ;

// Loop
int j = 0;
real errv = 1.;
real erradap = 0.001;
while (errv > 1e-6){
    // Update
    j++;

    // Solve
    Pu;
    Pv;

    // Plot
    plot(Th, u, v);

    // Error
    errv = int1d(Th, f)(v*v);

    // Movemesh
    real coef = 1.;
    real mintcc = checkmovemesh(Th, [x, y])/5.;
    real mint = checkmovemesh(Th, [x, y-v*coef]);

    if (mint < mintcc || j%10==0){ //mesh too bad => remeshing
        Th = adaptmesh(Th, u, err=erradap);
        mintcc = checkmovemesh(Th, [x, y])/5.;
    }

    while (1){
        real mint = checkmovemesh(Th, [x, y-v*coef]);

        if (mint > mintcc) break;

        cout << "min |T| = " << mint << endl;
        coef /= 1.5;
    }

    Th=movemesh(Th, [x, y-coef*v]);

    // Display
    cout << endl << j << " - errv = " << errv << endl;
}

// Plot
plot(Th);
plot(u, wait=true);
```

(a) The final solution on the new domain Ω^{72}

(b) The adapted mesh of the domain Ω^{72}

5.12 Non-linear elasticity

The nonlinear elasticity problem is: find the displacement (u_1, u_2) minimizing J:

$$\min J(u_1, u_2) = \int_\Omega f(F2) - \int_{\Gamma_p} P_a\, u_2$$

where $F2(u_1, u_2) = A(E[u_1, u_2], E[u_1, u_2])$ and $A(X, Y)$ is bilinear symmetric positive form with respect two matrix X, Y.

where f is a given $\mathcal{C}^2$ function, and $E[u_1, u_2] = (E_{ij})_{i=1,2,\, j=1,2}$ is the Green-Saint Venant deformation tensor defined with:

$$E_{ij} = 0.5\big((\partial_i u_j + \partial_j u_i) + \sum_k \partial_i u_k \times \partial_j u_k\big)$$

Denote $\mathbf{u} = (u_1, u_2)$, $\mathbf{v} = (v_1, v_2)$, $\mathbf{w} = (w_1, w_2)$. So, the differential of J is:

$$DJ(\mathbf{u})(\mathbf{v}) = \int DF2(\mathbf{u})(\mathbf{v})\ f'(F2(\mathbf{u}))) - \int_{\Gamma_p} P_a v_2$$

where $DF2(\mathbf{u})(\mathbf{v}) = 2\ A(DE[\mathbf{u}](\mathbf{v}), E[\mathbf{u}])$ and DE is the first differential of E.

The second order differential is:

$$\begin{aligned} D^2J(\mathbf{u})((\mathbf{v}), (\mathbf{w})) &= \int DF2(\mathbf{u})(\mathbf{v})\ DF2(\mathbf{u})(\mathbf{w})\ f''(F2(\mathbf{u}))) \\ &+ \int D^2F2(\mathbf{u})(\mathbf{v}, \mathbf{w})\ f'(F2(\mathbf{u}))) \end{aligned}$$

where:

$$D^2F2(\mathbf{u})(\mathbf{v}, \mathbf{w}) = 2\ A(\ D^2E[\mathbf{u}](\mathbf{v}, \mathbf{w})\ ,\ E[\mathbf{u}]\ ) + 2\ A(\ DE[\mathbf{u}](\mathbf{v})\ , DE[\mathbf{u}](\mathbf{w})\ ).$$

and D^2E is the second differential of E.

So all notations can be define with *macro*:

```
macro EL(u, v) [dx(u), (dx(v)+dy(u)), dy(v)] //is [epsilon_11, 2epsilon_12, epsilon_
↪22]

macro ENL(u, v) [
     (dx(u)*dx(u) + dx(v)*dx(v))*0.5,
     (dx(u)*dy(u) + dx(v)*dy(v)),
     (dy(u)*dy(u) + dy(v)*dy(v))*0.5
] //

macro dENL(u, v, uu, vv) [
     (dx(u)*dx(uu) + dx(v)*dx(vv)),
     (dx(u)*dy(uu) + dx(v)*dy(vv) + dx(uu)*dy(u) + dx(vv)*dy(v)),
     (dy(u)*dy(uu) + dy(v)*dy(vv))
```

(continues on next page)

(continued from previous page)

```
] //

macro E(u, v) (EL(u,v) + ENL(u,v)) //is [E_11, 2E_12, E_22]
macro dE(u, v, uu, vv) (EL(uu, vv) + dENL(u, v, uu, vv)) //
macro ddE(u, v, uu, vv, uuu, vvv) dENL(uuu, vvv, uu, vv) //
macro F2(u, v) (E(u, v)'*A*E(u, v)) //
macro dF2(u, v, uu, vv) (E(u, v)'*A*dE(u, v, uu, vv)*2.) //
macro ddF2(u, v, uu, vv, uuu, vvv) (
    (dE(u, v, uu, vv)'*A*dE(u, v, uuu, vvv))*2.
    + (E(u, v)'*A*ddE(u, v, uu, vv, uuu, vvv))*2.
) //
```

The Newton Method is:

choose $n = 0$,and u_0, v_0 the initial displacement

- loop:
  - find (du, dv) : solution of

$$D^2J(u_n, v_n)((w, s), (du, dv)) = DJ(u_n, v_n)(w, s), \quad \forall w, s$$

  - $un = un - du, \quad vn = vn - dv$
  - until (du, dv) small is enough

The way to implement this algorithm in **FreeFEM** is use a macro tool to implement A and $F2$, f, f',f''.

A macro is like in `ccp` preprocessor of C++, but this begin by `macro` and the end of the macro definition is before the comment `//`. In this case the macro is very useful because the type of parameter can be change. And it is easy to make automatic differentiation.

Fig. 5.29: The deformed domain

```
// Macro
macro EL(u, v) [dx(u), (dx(v)+dy(u)), dy(v)] //is [epsilon_11, 2epsilon_12, epsilon_
↪22]

macro ENL(u, v) [
    (dx(u)*dx(u) + dx(v)*dx(v))*0.5,
    (dx(u)*dy(u) + dx(v)*dy(v)),
```

(continues on next page)

(continued from previous page)

```
    (dy(u)*dy(u) + dy(v)*dy(v))*0.5
    ] //

macro dENL(u, v, uu, vv) [
    (dx(u)*dx(uu) + dx(v)*dx(vv)),
    (dx(u)*dy(uu) + dx(v)*dy(vv) + dx(uu)*dy(u) + dx(vv)*dy(v)),
    (dy(u)*dy(uu) + dy(v)*dy(vv))
    ] //

macro E(u, v) (EL(u,v) + ENL(u,v)) //is [E_11, 2E_12, E_22]
macro dE(u, v, uu, vv) (EL(uu, vv) + dENL(u, v, uu, vv)) //
macro ddE(u, v, uu, vv, uuu, vvv) dENL(uuu, vvv, uu, vv) //
macro F2(u, v) (E(u, v)'*A*E(u, v)) //
macro dF2(u, v, uu, vv) (E(u, v)'*A*dE(u, v, uu, vv)*2.) //
macro ddF2(u, v, uu, vv, uuu, vvv) (
      (dE(u, v, uu, vv)'*A*dE(u, v, uuu, vvv))*2.
    + (E(u, v)'*A*ddE(u, v, uu, vv, uuu, vvv))*2.
    ) //

macro f(u) ((u)*(u)*0.25) //
macro df(u) ((u)*0.5) //
macro ddf(u) (0.5) //

// Parameters
real mu = 0.012e5; //kg/cm^2
real lambda = 0.4e5; //kg/cm^2
real Pa = 1e2;

// sigma = 2 mu E + lambda tr(E) Id
// A(u,v) = sigma(u):E(v)
//
// ( a b )
// ( b c )
//
// tr*Id : (a,b,c) -> (a+c,0,a+c)
// so the associed matrix is:
// ( 1 0 1 )
// ( 0 0 0 )
// ( 1 0 1 )

real a11 = 2*mu + lambda;
real a22 = mu; //because [0, 2*t12, 0]' A [0, 2*s12,0] = 2*mu*(t12*s12 + t21*s21) =
↪4*mu*t12*s12
real a33 = 2*mu + lambda;
real a12 = 0;
real a13 = lambda;
real a23 = 0;
// symetric part
real a21 = a12;
real a31 = a13;
real a32 = a23;

//the matrix A
func A = [[a11, a12, a13], [a21, a22, a23], [a31, a32, a33]];

// Mesh
int n = 30;
```

(continues on next page)

(continued from previous page)

```
int m = 10;
mesh Th = square(n, m, [x, .3*y]); //label: 1 bottom, 2 right, 3 up, 4 left;
int bottom = 1, right = 2, upper = 3, left = 4;
plot(Th);

// Fespace
fespace Wh(Th, P1dc);
Wh e2, fe2, dfe2, ddfe2;

fespace Vh(Th, [P1, P1]);
Vh [uu, vv] = [0, 0], [w, s], [un, vn] = [0, 0];

fespace Sh(Th, P1);
Sh u1, v1;

// Problem
varf vmass ([uu, vv], [w, s], solver=CG) = int2d(Th)(uu*w + vv*s);
matrix M = vmass(Vh, Vh);
problem NonLin([uu, vv], [w, s], solver=LU)
    = int2d(Th, qforder=1)( //(D^2 J(un))
           dF2(un, vn, uu, vv)*dF2(un, vn, w, s)*ddfe2
        + ddF2(un, vn, uu, vv, w, s)*ddfe2
    )
    - int1d(Th, upper)(
          Pa*s
    )
    - int2d(Th, qforder=1)( //(D J(un))
          dF2(un, vn, w, s)*dfe2
    )
    + on(right, left, uu=0, vv=0)
    ;

// Newton's method
for (int i = 0; i < 10; i++){
    cout << "Loop " << i << endl;

    // Update
    e2 = F2(un, vn);
    dfe2 = df(e2) ;
    ddfe2 = ddf(e2);
    cout << "e2 max = " <<e2[].max << ", min = " << e2[].min << endl;
    cout << "de2 max = "<< dfe2[].max << ", min = " << dfe2[].min << endl;
    cout << "dde2 max = "<< ddfe2[].max << ", min = " << ddfe2[].min << endl;

    // Solve
    NonLin;
    w[]  = M*uu[];

    // Residual
    real res = sqrt(w[]' * uu[]); //L^2 norm of [uu, vv]
    cout << " L^2 residual = " << res << endl;

    // Update
    v1 = vv;
    u1 = uu;
    cout << "u1 min = " <<u1[].min << ", u1 max = " << u1[].max << endl;
    cout << "v1 min = " <<v1[].min << ", v2 max = " << v1[].max << endl;
```

(continues on next page)

(continued from previous page)

```
    // Plot
    plot([uu, vv], wait=true, cmm="uu, vv");

    // Update
    un[] -= uu[];
    plot([un, vn], wait=true, cmm="displacement");

    if (res < 1e-5) break;
}

// Plot
plot([un, vn], wait=true);

// Movemesh
mesh th1 = movemesh(Th, [x+un, y+vn]);

// Plot
plot(th1, wait=true);
```

5.13 Compressible Neo-Hookean materials

Author: *Alex Sadovsky*

5.13.1 Notation

In what follows, the symbols $\mathbf{u}, \mathbf{F}, \mathbf{B}, \mathbf{C}, \underline{\sigma}$ denote, respectively, the displacement field, the deformation gradient, the left Cauchy-Green strain tensor $\mathbf{B} = \mathbf{F}\mathbf{F}^T$, the right Cauchy-Green strain tensor $\mathbf{C} = \mathbf{F}^T\mathbf{F}$, and the Cauchy stress tensor.

We also introduce the symbols $I_1 := \mathbf{tr}\ \mathbf{C}$ and $J := \det \mathbf{F}$. Use will be made of the identity:

$$\frac{\partial J}{\partial \mathbf{C}} = J\mathbf{C}^{-1}$$

The symbol $\mathbf{I}$ denotes the identity tensor. The symbol Ω_0 denotes the reference configuration of the body to be deformed. The unit volume in the reference (resp., deformed) configuration is denoted dV (resp., dV_0); these two are related by:

$$dV = JdV_0,$$

which allows an integral over Ω involving the Cauchy stress $\mathbf{T}$ to be rewritten as an integral of the Kirchhoff stress $\kappa = J\mathbf{T}$ over Ω_0.

5.13.2 Recommended References

For an exposition of nonlinear elasticity and of the underlying linear and tensor algebra, see [OGDEN1984]. For an advanced mathematical analysis of the Finite Element Method, see [RAVIART1998].

5.13.3 A Neo-Hookean Compressible Material

Constitutive Theory and Tangent Stress Measures

The strain energy density function is given by:

$$W = \frac{\mu}{2}(I_1 - \mathbf{tr}\ \mathbf{I} - 2\ln J)$$

(see [HORGAN2004], formula (12)).

The corresponding 2nd Piola-Kirchoff stress tensor is given by:

$$\mathbf{S}_n := \frac{\partial W}{\partial \mathbf{E}}(\mathbf{F}_n) = \mu(\mathbf{I} - \mathbf{C}^{-1})$$

The Kirchhoff stress, then, is:

$$\kappa = \mathbf{F}\mathbf{S}\mathbf{F}^T = \mu(\mathbf{B} - \mathbf{I})$$

The tangent Kirchhoff stress tensor at $\mathbf{F}_n$ acting on $\delta\mathbf{F}_{n+1}$ is, consequently:

$$\frac{\partial \kappa}{\partial \mathbf{F}}(\mathbf{F}_n)\delta\mathbf{F}_{n+1} = \mu\left[\mathbf{F}_n(\delta\mathbf{F}_{n+1})^T + \delta\mathbf{F}_{n+1}(\mathbf{F}_n)^T\right]$$

The Weak Form of the BVP in the Absence of Body (External) Forces

The Ω_0 we are considering is an elliptical annulus, whose boundary consists of two concentric ellipses (each allowed to be a circle as a special case), with the major axes parallel. Let P denote the dead stress load (traction) on a portion $\partial\Omega_0^t$ (= the inner ellipse) of the boundary $\partial\Omega_0$. On the rest of the boundary, we prescribe zero displacement.

The weak formulation of the boundary value problem is:

$$\left.\begin{array}{rcl} 0 & = & \int_{\Omega_0} \kappa[\mathbf{F}] \ : \ \left\{(\nabla \otimes \mathbf{w})(\mathbf{F})^{-1}\right\} \\ & - & \int_{\partial\Omega_0^t} P \cdot \hat{N}_0 \end{array}\right\}$$

For brevity, in the rest of this section we assume $P = 0$. The provided **FreeFEM** code, however, does not rely on this assumption and allows for a general value and direction of P.

Given a Newton approximation $\mathbf{u}_n$ of the displacement field $\mathbf{u}$ satisfying the BVP, we seek the correction $\delta\mathbf{u}_{n+1}$ to obtain a better approximation:

$$\mathbf{u}_{n+1} = \mathbf{u}_n + \delta\mathbf{u}_{n+1}$$

by solving the weak formulation:

$$\left.\begin{array}{rcl} 0 & = & \int_{\Omega_0} \kappa[\mathbf{F}_n + \delta\mathbf{F}_{n+1}] \ : \ \left\{(\nabla \otimes \mathbf{w})(\mathbf{F}_n + \delta\mathbf{F}_{n+1})^{-1}\right\} - \int_{\partial\Omega_0} P \cdot \hat{N}_0 \\ & = & \int_{\Omega_0} \left\{\kappa[\mathbf{F}_n] + \frac{\partial\kappa}{\partial\mathbf{F}}[\mathbf{F}_n]\delta\mathbf{F}_{n+1}\right\} \ : \ \left\{(\nabla \otimes \mathbf{w})(\mathbf{F}_n + \delta\mathbf{F}_{n+1})^{-1}\right\} \\ & = & \int_{\Omega_0} \left\{\kappa[\mathbf{F}_n] + \frac{\partial\kappa}{\partial\mathbf{F}}[\mathbf{F}_n]\delta\mathbf{F}_{n+1}\right\} \ : \ \left\{(\nabla \otimes \mathbf{w})(\mathbf{F}_n^{-1} + \mathbf{F}_n^{-2}\delta\mathbf{F}_{n+1})\right\} \\ & = & \int_{\Omega_0} \kappa[\mathbf{F}_n] \ : \ \left\{(\nabla \otimes \mathbf{w})\mathbf{F}_n^{-1}\right\} \\ & - & \int_{\Omega_0} \kappa[\mathbf{F}_n] \ : \ \left\{(\nabla \otimes \mathbf{w})(\mathbf{F}_n^{-2}\delta\mathbf{F}_{n+1})\right\} \\ & + & \int_{\Omega_0} \left\{\frac{\partial\kappa}{\partial\mathbf{F}}[\mathbf{F}_n]\delta\mathbf{F}_{n+1}\right\} \ : \ \left\{(\nabla \otimes \mathbf{w})\mathbf{F}_n^{-1}\right\} \end{array}\right\} \text{for all test functions} \mathbf{w},$$

where we have taken:

$$\delta\mathbf{F}_{n+1} = \nabla \otimes \delta\mathbf{u}_{n+1}$$

Note: Contrary to standard notational use, the symbol δ here bears no variational context. By δ we mean simply an increment in the sense of Newton's Method. The role of a variational virtual displacement here is played by $\mathbf{w}$.

5.13.4 An Approach to Implementation in FreeFEM

Introducing the code-like notation, where a string in <>'s is to be read as one symbol, the individual components of the tensor:

$$< TanK >:= \frac{\partial \kappa}{\partial \mathbf{F}}[\mathbf{F}_n]\delta\mathbf{F}_{n+1}$$

will be implemented as the macros $< TanK11 >, < TanK12 >, \ldots$

The individual components of the tensor quantities:

$$\mathbf{D}_1 := \mathbf{F}_n(\delta\mathbf{F}_{n+1})^T + \delta\mathbf{F}_{n+1}(\mathbf{F}_n)^T,$$

$$\mathbf{D}_2 := \mathbf{F}_n^{-T}\delta\mathbf{F}_{n+1},$$

$$\mathbf{D}_3 := (\nabla \otimes \mathbf{w})\mathbf{F}_n^{-2}\delta\mathbf{F}_{n+1},$$

and

$$\mathbf{D}_4 := (\nabla \otimes \mathbf{w})\mathbf{F}_n^{-1},$$

will be implemented as the macros:

$$\left.\begin{array}{llll} < d1Aux11 >, & < d1Aux12 >, & \ldots & , < d1Aux22 >, \\ < d2Aux11 >, & < d2Aux12 >, & \ldots & , < d2Aux22 > \\ < d3Aux11 >, & < d3Aux12 >, & \ldots & , < d3Aux22 > \\ < d4Aux11 >, & < d4Aux12 >, & \ldots & , < d4Aux22 > \end{array}\right\},$$

respectively.

In the above notation, the tangent Kirchhoff stress term becomes

$$\frac{\partial \kappa}{\partial \mathbf{F}}(\mathbf{F}_n)\,\delta\mathbf{F}_{n+1} = \mu\,\mathbf{D}_1$$

while the weak BVP formulation acquires the form:

$$\left.\begin{array}{rcl} 0 & = & \int_{\Omega_0} \kappa[\mathbf{F}_n] \,:\, \mathbf{D}_4 \\ & - & \int_{\Omega_0} \kappa[\mathbf{F}_n] \,:\, \mathbf{D}_3 \\ & + & \int_{\Omega_0} \left\{\frac{\partial \kappa}{\partial \mathbf{F}}[\mathbf{F}_n]\delta\mathbf{F}_{n+1}\right\} \,:\, \mathbf{D}_4 \end{array}\right\} \quad \text{for all test functions} \mathbf{w}$$

```
// Macro
//Macros for the gradient of a vector field (u1, u2)
macro grad11(u1, u2) (dx(u1)) //
macro grad21(u1, u2) (dy(u1)) //
macro grad12(u1, u2) (dx(u2)) //
macro grad22(u1, u2) (dy(u2)) //

//Macros for the deformation gradient
macro F11(u1, u2) (1.0 + grad11(u1, u2)) //
macro F12(u1, u2) (0.0 + grad12(u1, u2)) //
macro F21(u1, u2) (0.0 + grad21(u1, u2)) //
macro F22(u1, u2) (1.0 + grad22(u1, u2)) //

//Macros for the incremental deformation gradient
macro dF11(varu1, varu2) (grad11(varu1, varu2)) //
macro dF12(varu1, varu2) (grad12(varu1, varu2)) //
```

(continues on next page)

(continued from previous page)

```
macro dF21(varu1, varu2) (grad21(varu1, varu2)) //
macro dF22(varu1, varu2) (grad22(varu1, varu2)) //

//Macro for the determinant of the deformation gradient
macro J(u1, u2) (
      F11(u1, u2)*F22(u1, u2)
    - F12(u1, u2)*F21(u1, u2)
) //

//Macros for the inverse of the deformation gradient
macro Finv11 (u1, u2) (
      F22(u1, u2) / J(u1, u2)
) //
macro Finv22 (u1, u2) (
      F11(u1, u2) / J(u1, u2)
) //
macro Finv12 (u1, u2) (
    - F12(u1, u2) / J(u1, u2)
) //
macro Finv21 (u1, u2) (
    - F21(u1, u2) / J(u1, u2)
) //

//Macros for the square of the inverse of the deformation gradient
macro FFinv11 (u1, u2) (
      Finv11(u1, u2)^2
    + Finv12(u1, u2)*Finv21(u1, u2)
) //

macro FFinv12 (u1, u2) (
      Finv12(u1, u2)*(Finv11(u1, u2)
    + Finv22(u1, u2))
) //

macro FFinv21 (u1, u2) (
      Finv21(u1, u2)*(Finv11(u1, u2)
    + Finv22(u1, u2))
) //

macro FFinv22 (u1, u2) (
      Finv12(u1, u2)*Finv21(u1, u2)
    + Finv22(u1, u2)^2
) //

//Macros for the inverse of the transpose of the deformation gradient
macro FinvT11(u1, u2) (Finv11(u1, u2)) //
macro FinvT12(u1, u2) (Finv21(u1, u2)) //
macro FinvT21(u1, u2) (Finv12(u1, u2)) //
macro FinvT22(u1, u2) (Finv22(u1, u2)) //

//The left Cauchy-Green strain tensor
macro B11(u1, u2) (
      F11(u1, u2)^2 + F12(u1, u2)^2
)//

macro B12(u1, u2) (
      F11(u1, u2)*F21(u1, u2)
```

(continues on next page)

(continued from previous page)

```
    + F12(u1, u2)*F22(u1, u2)
)//

macro B21(u1, u2) (
      F11(u1, u2)*F21(u1, u2)
    + F12(u1, u2)*F22(u1, u2)
)//

macro B22(u1, u2)(
      F21(u1, u2)^2 + F22(u1, u2)^2
)//

//The macros for the auxiliary tensors (D0, D1, D2, ...): Begin
//The tensor quantity D0 = F{n} (delta F{n+1})^T
macro d0Aux11 (u1, u2, varu1, varu2) (
      dF11(varu1, varu2) * F11(u1, u2)
    + dF12(varu1, varu2) * F12(u1, u2)
)//

macro d0Aux12 (u1, u2, varu1, varu2) (
      dF21(varu1, varu2) * F11(u1, u2)
    + dF22(varu1, varu2) * F12(u1, u2)
)//

macro d0Aux21 (u1, u2, varu1, varu2) (
      dF11(varu1, varu2) * F21(u1, u2)
    + dF12(varu1, varu2) * F22(u1, u2)
)//

macro d0Aux22 (u1, u2, varu1, varu2) (
      dF21(varu1, varu2) * F21(u1, u2)
    + dF22(varu1, varu2) * F22(u1, u2)
)//

///The tensor quantity D1 = D0 + D0^T
macro d1Aux11 (u1, u2, varu1, varu2) (
      2.0 * d0Aux11 (u1, u2, varu1, varu2)
)//

macro d1Aux12 (u1, u2, varu1, varu2) (
      d0Aux12 (u1, u2, varu1, varu2)
    + d0Aux21 (u1, u2, varu1, varu2)
)//

macro d1Aux21 (u1, u2, varu1, varu2) (
      d1Aux12 (u1, u2, varu1, varu2)
)//

macro d1Aux22 (u1, u2, varu1, varu2)(
      2.0 * d0Aux22 (u1, u2, varu1, varu2)
)//

///The tensor quantity D2 = F^{-T}_{n} dF_{n+1}
macro d2Aux11 (u1, u2, varu1, varu2) (
      dF11(varu1, varu2) * FinvT11(u1, u2)
    + dF21(varu1, varu2) * FinvT12(u1, u2)
)//
```

(continues on next page)

(continued from previous page)

```

macro d2Aux12 (u1, u2, varu1, varu2) (
      dF12(varu1, varu2) * FinvT11(u1, u2)
    + dF22(varu1, varu2) * FinvT12(u1, u2)
)//

macro d2Aux21 (u1, u2, varu1, varu2) (
      dF11(varu1, varu2) * FinvT21(u1, u2)
    + dF21(varu1, varu2) * FinvT22(u1, u2)
)//

macro d2Aux22 (u1, u2, varu1, varu2) (
      dF12(varu1, varu2) * FinvT21(u1, u2)
    + dF22(varu1, varu2) * FinvT22(u1, u2)
)//

///The tensor quantity D3 = F^{-2}_{n} dF_{n+1}
macro d3Aux11 (u1, u2, varu1, varu2, w1, w2) (
      dF11(varu1, varu2) * FFinv11(u1, u2) * grad11(w1, w2)
    + dF21(varu1, varu2) * FFinv12(u1, u2) * grad11(w1, w2)
    + dF11(varu1, varu2) * FFinv21(u1, u2) * grad12(w1, w2)
    + dF21(varu1, varu2) * FFinv22(u1, u2) * grad12(w1, w2)
)//

macro d3Aux12 (u1, u2, varu1, varu2, w1, w2) (
      dF12(varu1, varu2) * FFinv11(u1, u2) * grad11(w1, w2)
    + dF22(varu1, varu2) * FFinv12(u1, u2) * grad11(w1, w2)
    + dF12(varu1, varu2) * FFinv21(u1, u2) * grad12(w1, w2)
    + dF22(varu1, varu2) * FFinv22(u1, u2) * grad12(w1, w2)
)//

macro d3Aux21 (u1, u2, varu1, varu2, w1, w2) (
      dF11(varu1, varu2) * FFinv11(u1, u2) * grad21(w1, w2)
    + dF21(varu1, varu2) * FFinv12(u1, u2) * grad21(w1, w2)
    + dF11(varu1, varu2) * FFinv21(u1, u2) * grad22(w1, w2)
    + dF21(varu1, varu2) * FFinv22(u1, u2) * grad22(w1, w2)
)//

macro d3Aux22 (u1, u2, varu1, varu2, w1, w2) (
      dF12(varu1, varu2) * FFinv11(u1, u2) * grad21(w1, w2)
    + dF22(varu1, varu2) * FFinv12(u1, u2) * grad21(w1, w2)
    + dF12(varu1, varu2) * FFinv21(u1, u2) * grad22(w1, w2)
    + dF22(varu1, varu2) * FFinv22(u1, u2) * grad22(w1, w2)
)//

///The tensor quantity D4 = (grad w) * Finv
macro d4Aux11 (w1, w2, u1, u2) (
      Finv11(u1, u2)*grad11(w1, w2)
    + Finv21(u1, u2)*grad12(w1, w2)
)//

macro d4Aux12 (w1, w2, u1, u2) (
      Finv12(u1, u2)*grad11(w1, w2)
    + Finv22(u1, u2)*grad12(w1, w2)
)//

macro d4Aux21 (w1, w2, u1, u2) (
```

(continues on next page)

(continued from previous page)

```
      Finv11(u1, u2)*grad21(w1, w2)
    + Finv21(u1, u2)*grad22(w1, w2)
)//

macro d4Aux22 (w1, w2, u1, u2) (
      Finv12(u1, u2)*grad21(w1, w2)
    + Finv22(u1, u2)*grad22(w1, w2)
)//
//The macros for the auxiliary tensors (D0, D1, D2, ...): End

//The macros for the various stress measures: BEGIN
//The Kirchhoff stress tensor
macro StressK11(u1, u2) (
      mu * (B11(u1, u2) - 1.0)
)//

//The Kirchhoff stress tensor
macro StressK12(u1, u2) (
      mu * B12(u1, u2)
)//

//The Kirchhoff stress tensor
macro StressK21(u1, u2) (
      mu * B21(u1, u2)
)//

//The Kirchhoff stress tensor
macro StressK22(u1, u2) (
      mu * (B22(u1, u2) - 1.0)
)//

//The tangent Kirchhoff stress tensor
macro TanK11(u1, u2, varu1, varu2) (
      mu * d1Aux11(u1, u2, varu1, varu2)
)//

macro TanK12(u1, u2, varu1, varu2) (
      mu * d1Aux12(u1, u2, varu1, varu2)
)//

macro TanK21(u1, u2, varu1, varu2) (
      mu * d1Aux21(u1, u2, varu1, varu2)
)//

macro TanK22(u1, u2, varu1, varu2) (
      mu * d1Aux22(u1, u2, varu1, varu2)
)//
//The macros for the stress tensor components: END

// Parameters
real mu = 5.e2; //Elastic coefficients (kg/cm^2)
real D = 1.e3; //(1 / compressibility)
real Pa = -3.e2; //Stress loads

real InnerRadius = 1.e0; //The wound radius
real OuterRadius = 4.e0; //The outer (truncated) radius
real tol = 1.e-4; //Tolerance (L^2)
```

(continues on next page)

(continued from previous page)

```
real InnerEllipseExtension = 1.e0; //Extension of the inner ellipse ((major axis) - (minor axis))

int m = 40, n = 20;

// Mesh
border InnerEdge(t=0, 2.*pi){x=(1.0 + InnerEllipseExtension)*InnerRadius*cos(t); y=InnerRadius*sin(t); label=1;}
border OuterEdge(t=0, 2.*pi){x=(1.0 + 0.0*InnerEllipseExtension)*OuterRadius*cos(t); y=OuterRadius*sin(t); label=2;}
mesh Th = buildmesh(InnerEdge(-m) + OuterEdge(n));
int bottom = 1, right = 2, upper = 3, left = 4;
plot(Th);

// Fespace
fespace Wh(Th, P1dc);
fespace Vh(Th, [P1, P1]);
Vh [w1, w2], [u1n,u2n], [varu1, varu2];
Vh [ehat1x, ehat1y], [ehat2x, ehat2y];
Vh [auxVec1, auxVec2]; //The individual elements of the total 1st Piola-Kirchoff stress
Vh [ef1, ef2];

fespace Sh(Th, P1);
Sh p, ppp;
Sh StrK11, StrK12, StrK21, StrK22;
Sh u1, u2;

// Problem
varf vfMass1D(p, q) = int2d(Th)(p*q);
matrix Mass1D = vfMass1D(Sh, Sh, solver=CG);

p[] = 1;
ppp[] = Mass1D * p[];

real DomainMass = ppp[].sum;
cout << "DomainMass = " << DomainMass << endl;

varf vmass ([u1, u2], [v1, v2], solver=CG)
    = int2d(Th)( (u1*v1 + u2*v2) / DomainMass );

matrix Mass = vmass(Vh, Vh);

matrix Id = vmass(Vh, Vh);

//Define the standard Euclidean basis functions
[ehat1x, ehat1y] = [1.0, 0.0];
[ehat2x, ehat2y] = [0.0, 1.0];

real ContParam, dContParam;

problem neoHookeanInc ([varu1, varu2], [w1, w2], solver=LU)
    = int2d(Th, qforder=1)(
        - (
              StressK11 (u1n, u2n) * d3Aux11(u1n, u2n, varu1, varu2, w1, w2)
            + StressK12 (u1n, u2n) * d3Aux12(u1n, u2n, varu1, varu2, w1, w2)
            + StressK21 (u1n, u2n) * d3Aux21(u1n, u2n, varu1, varu2, w1, w2)
```

(continues on next page)

(continued from previous page)

```
            + StressK22 (u1n, u2n) * d3Aux22(u1n, u2n, varu1, varu2, w1, w2)
        )
        + TanK11 (u1n, u2n, varu1, varu2) * d4Aux11(w1, w2, u1n, u2n)
        + TanK12 (u1n, u2n, varu1, varu2) * d4Aux12(w1, w2, u1n, u2n)
        + TanK21 (u1n, u2n, varu1, varu2) * d4Aux21(w1, w2, u1n, u2n)
        + TanK22 (u1n, u2n, varu1, varu2) * d4Aux22(w1, w2, u1n, u2n)
    )
    + int2d(Th, qforder=1)(
          StressK11 (u1n, u2n) * d4Aux11(w1, w2, u1n, u2n)
        + StressK12 (u1n, u2n) * d4Aux12(w1, w2, u1n, u2n)
        + StressK21 (u1n, u2n) * d4Aux21(w1, w2, u1n, u2n)
        + StressK22 (u1n, u2n) * d4Aux22(w1, w2, u1n, u2n)
    )
    //Choose one of the following two boundary conditions involving Pa:
    // Load vectors normal to the boundary:
    - int1d(Th, 1)(
          Pa * (w1*N.x + w2*N.y)
    )
    //Load vectors tangential to the boundary:
    //- int1d(Th, 1)(
    //    Pa * (w1*N.y - w2*N.x)
    //)
    + on(2, varu1=0, varu2=0)
    ;

//Auxiliary variables
matrix auxMat;

// Newton's method
ContParam = 0.;
dContParam = 0.01;

//Initialization:
[varu1, varu2] = [0., 0.];
[u1n, u2n] = [0., 0.];
real res = 2. * tol;
real eforceres;
int loopcount = 0;
int loopmax = 45;

// Iterations
while (loopcount <= loopmax && res >= tol){
    loopcount ++;
    cout << "Loop " << loopcount << endl;

    // Solve
    neoHookeanInc;

    // Update
    u1 = varu1;
    u2 = varu2;

    // Residual
    w1[] = Mass*varu1[];
    res = sqrt(w1[]' * varu1[]); //L^2 norm of [varu1, varu2]
    cout << " L^2 residual = " << res << endl;

```

(continues on next page)

(continued from previous page)

```
    // Newton
    u1n[] += varu1[];

    // Plot
    plot([u1n,u2n], cmm="displacement");
}

// Plot
plot(Th, wait=true);

// Movemesh
mesh Th1 = movemesh(Th, [x+u1n, y+u2n]);

// Plot
plot(Th1, wait=true);
plot([u1n,u2n]);
```

5.14 Whispering gallery modes

Author: I. S. Grudinin

In whispering gallery mode (WGM) resonators, which are typically spheres or disks, electromagnetic field is trapped by total internal reflections from the boundary. Modes of such resonators are distinguished by compact volume and record high quality factors (Q) in a broad range of frequencies.

Modern applications of such resonators include microwave and optical cavities for atomic clocks, cavity optomechanics, nonlinear and quantum optics. Analytical solutions for WG modes are only available for a limited number of idealized geometries, such as sphere or ellipsoid. Since resonator dimensions are typically much larger than optical wavelength, direct application of numerical 3D finite difference time domain (FDTD) or finite element methods (FEM) is not practical. It's possible to solve the vectorial wave equation by reducing it to a two dimensional case by taking axial symmetry into account.

Such reduction leads to a system of 3 equations to be solved in a 2D "$\rho - z$" section of a resonator. Please refer to [OXBORROW2007] for a detailed derivation and to [GRUDININ2012] for an example of using **FreeFEM** to compute WGMs.

5.14.1 Wave equation for the WGMs

Since electric field is discontinuous on the surface of a dielectric and magnetic field is typically not, we derive our equations for the magnetic field. The electric field can be easily derived at a later stage from $\vec{E} = \frac{i}{\omega\epsilon_0}\hat{\epsilon}^{-1}\nabla \times \vec{H}$. Following a standard procedure starting with Maxwell equations we derive a wave equation in a single-axis anisotropic medium such as an optical crystal:

$$\nabla \times \left(\hat{\epsilon}^{-1}\nabla \times \vec{H}\right) - k_0^2\vec{H} - \alpha\nabla\left(\nabla \cdot \vec{H}\right) = 0 \tag{5.16}$$

Here $k_0 = \omega/c$ is the wavenumber, α is the penalty term added to fight spurious FEM solutions. For anisotropic single-axis medium with $\partial\hat{\epsilon}/\partial\phi = 0$ in cylindrical system of coordinates we have:

$$\hat{\epsilon} = \begin{pmatrix} \epsilon_\rho & 0 & 0 \\ 0 & \epsilon_\rho & 0 \\ 0 & 0 & \epsilon_z \end{pmatrix}.$$

We now assume axial symmetry of our electromagnetic fields and insert an imaginary unity in front of the H_ϕ to allow all field components to be real numbers and also to account for the phase shift of this component $\vec{H}(\rho,\phi,z) = \{H_\rho(\rho,z), iH_\phi(\rho,z), H_z(\rho,z)\} \times e^{im\phi}$.

We write the wave equation (5.16) explicitly in cylindrical coordinates, thus obtaining a set of three differential equations for the domain Ω given by the resonator's cross section and some space outside:

$$\begin{aligned} A_1\{H^t_\rho, H^t_\phi, H^t_z\} &= 0 \\ A_2\{H^t_\rho, H^t_\phi, H^t_z\} &= 0 \\ A_3\{H^t_\rho, H^t_\phi, H^t_z\} &= 0 \end{aligned}$$

The numerical solutions of these equations and boundary conditions can be found with **FreeFEM** if we write the system in the weak, or integral form.

5.14.2 Weak formulation

In general, to obtain the integral or "weak" statements equivalent to system (5.17) and boundary conditions we form a scalar dot product between an arbitrary magnetic field test function $\mathbf{H}^t = \{H^t_\rho, H^t_\phi, H^t_z\}$ and the components of our vectorial equation A_1, A_2, A_3, and integrate over the resonator's cross section domain Ω (and its boundary for the boundary conditions):

$$\int_\Omega (H^t_\rho A_1 + H^t_\phi A_2 + H^t_z A_3) d\Omega$$

We can reduce the order of partial derivatives in this integral by using the Green's formula for integration by parts. For example:

$$\int_\Omega H^t_z \frac{\partial^2 H_z}{\partial \rho^2} d\Omega = -\int_\Omega \frac{\partial H^t_z}{\partial \rho}\frac{\partial H_z}{\partial \rho} d\Omega + \oint H^t_z \frac{\partial H_z}{\partial \rho} n_\rho d\Gamma$$

Thus converting equations (5.17) we obtain a large expression for the weak form.

5.14.3 A dielectric sphere example with FreeFEM

We now compute the fundamental mode frequency for a fused silica sphere. The sphere is 36 micrometer in diameter, the refractive index is 1.46, the boundary condition is the magnetic wall (which can actually be omitted as it holds automatically).

```
// Parameters
real radius = 36; //approximate radius of the cavity
real yb = -10, yt = -yb; //window yb=bottom and yt=top coordinates
real xl = radius-5, xr = radius+3; //window xl=left and xr=right coordinates
real angle = asin((yt)/radius); //angle of the sphere segment to model in radians
int Nm = 60; //number of mesh vertices per border
real ne = 1.46; //n_e-extraordinary refractive index (root of permittivity parallel␣
↪to z-axis, epara)
real no = 1.46; //n_o-ordinary refractive index (root of permittivity orthogonal to z-
↪axis, eorto)
real nm = 1; //refractive index of surrounding medium (air)

int nev = 4; // number of eigen values to find

int M = 213; //azimuthal mode order ~ 2Pi*n*R/lambda
real alpha = 1; //penalty term
```

(continues on next page)

(continued from previous page)

```

// Mesh
border W1l(t=0, 1){x=xl+(radius*cos(angle)-xl)*(1-t); y=yt; label=1;}
border W1r(t=0, 1){x=xr-(xr-radius*cos(angle))*(t); y=yt; label=1;}
border W2(t=0, 1){x=xr; y=yb+(yt-yb)*t; label=1;}
border W3l(t=0, 1){x=xl+(radius*cos(angle)-xl)*(t); y=yb; label=1;}
border W3r(t=0, 1){x=xr-(xr-radius*cos(angle))*(1-t); y=yb; label=1;}
border W4(t=0, 1){x=xl; y=yt-(yt-yb)*t; label=1;}
border S(t=0, 1){x=radius*cos((t-0.5)*2*angle); y=radius*sin((t-0.5)*2*angle); label=2;}
mesh Th = buildmesh(W1r(Nm/4) + W1l(Nm/4) + W4(Nm) + W3l(Nm/4) + W3r(Nm/4) + W2(Nm) + S(Nm));
plot(Th, WindowIndex=0);

// Fespace
fespace Ph(Th, P0);
Ph reg = region;

int ncav = reg(xl+1, 0); // cavity
int nair = reg(xr-1, 0); //air
Ph eorto = no^2*(region==ncav) + nm^2*(region==nair); //subdomains for epsilon values inside and outside the resonators
Ph epara = ne^2*(region==ncav) + nm^2*(region==nair); //subdomains for epsilon values inside and outside the resonators

//supplementary variables to store eigenvectors, defined on mesh Th with P2 elements - Largange quadratic.
fespace Supp(Th, P2);
Supp eHsqr;

//3d vector FE space
fespace Vh(Th, [P2, P2, P2]);
Vh [Hr, Hphi, Hz], [vHr, vHphi, vHz]; //magnetic field components on Vh space and test functions vH

// Macro
//boundary condition macros
macro EWall(Hr, Hphi, Hz) (
      dy(Hr) - dx(Hz) + Hr*N.x + Hz*N.y
    - epara*(Hz*M - dy(Hphi)*x)*N.y
    + eorto*(Hphi - Hr*M+dx(Hphi)*x)*N.x) //
macro MWall(Hr, Hphi, Hz) (
      Hphi + Hz*N.x - Hr*N.y
    + epara*(Hz*M - dy(Hphi)*x)*N.x
    + eorto*(Hphi - Hr*M+dx(Hphi)*x)*N.y ) //

// Problem
real sigma =(M/(ne*radius))^2+2; // value of the shift (k^2), where the modes will be found
varf b ([Hr, Hphi, Hz], [vHr, vHphi, vHz])
    = int2d(Th)(
          x*(Hr*vHr+Hphi*vHphi+Hz*vHz)
    )
    ;
// OP = A - sigma B ; // the shifted matrix
varf op ([Hr, Hphi, Hz], [vHr, vHphi, vHz])=
    int2d(Th)(
```

(continues on next page)

(continued from previous page)

```
            (
                (eorto*(vHphi*Hphi - M*(vHphi*Hr + Hphi*vHr) + M^2*vHr*Hr) + epara*M^2*vHz*Hz)/x //A/r
              + eorto*(dx(vHphi)*(Hphi - M*Hr) + dx(Hphi)*(vHphi - M*vHr)) - epara*M*(vHz*dy(Hphi) + Hz*dy(vHphi)) //B
              + x*(eorto*dx(vHphi)*dx(Hphi) + epara*((dx(vHz) - dy(vHr))*(dx(Hz) - dy(Hr)) + dy(vHphi)*dy(Hphi))) //C
          )/(eorto*epara)
          + alpha*(
                (vHr*Hr - M*(vHphi*Hr + Hphi*vHr) + M^2*vHphi*Hphi)/x //D/r
              + (dx(vHr) + dy(vHz))*(Hr - M*Hphi) + (vHr - M*vHphi)*(dx(Hr) + dy(Hz)) //E
              + x*(dx(vHr) + dy(vHz))*(dx(Hr) + dy(Hz)) //F
          )
          -sigma*x*(vHr*Hr + vHphi*Hphi + vHz*Hz)
      )
      //electric wall boundary condition on the boundary of computation domain
      +int1d(Th, 1)(
            EWall(Hr, Hphi, Hz)*EWall(vHr, vHphi, vHz)
      )
      ;
//setting sparce matrices and assigning the solver UMFPACK to solve eigenvalue problem
matrix B = b(Vh, Vh, solver=UMFPACK);
matrix OP = op(Vh, Vh, solver=UMFPACK);

// Solve
real[int] ev(nev); //to store the nev eigenvalue
Vh[int] [eHr, eHphi, eHz](nev); //to store the nev eigenvector
//calling ARPACK on sparce matrices with the assigned solver UMFPACK:
int k = EigenValue(OP, B, sym=true, sigma=sigma, value=ev, vector=eHr, tol=1e-10, maxit=0, ncv=0);

k = min(k, nev); //sometimes the number of converged eigen values
                 //can be greater than nev

//file to output mode values
ofstream f("modes.txt");
//setting number of digits in the file output
int nold = f.precision(11);

// Plot & Save
for (int i = 0; i < k; i++){
    real lambda = 2*pi/sqrt(ev[i]);
    eHsqr = (sqrt(eHr[i]^2 + eHphi[i]^2 + eHz[i]^2)); //intensity from magnetic field components
    plot(eHsqr, WindowIndex=i, value=1, nbiso=20,LabelColors=1, aspectratio=1, cmm="Mode "+i+", lambda="+lambda+", F="+(299792.458/lambda));
    f << "Mode "<< i << ", ka=" << sqrt(ev[i])*radius << endl;
}
```

CHAPTER

SIX

EXAMPLES

6.1 Misc

6.1.1 Poisson's Equation

```
// Parameters
int nn = 20;
real L = 1.;
real H = 1.;
real l = 0.5;
real h = 0.5;

func f = 1.;
func g = 0.;

int NAdapt = 10;

// Mesh
border b1(t=0, L){x=t; y=0;};
border b2(t=0, h){x=L; y=t;};
border b3(t=L, l){x=t; y=h;};
border b4(t=h, H){x=l; y=t;};
border b5(t=l, 0){x=t; y=H;};
border b6(t=H, 0){x=0; y=t;};

mesh Th = buildmesh(b1(nn*L) + b2(nn*h) + b3(nn*(L-l)) + b4(nn*(H-h)) + b5(nn*l) +
↪b6(nn*H));

// Fespace
fespace Vh(Th, P1); // Change P1 to P2 to test P2 finite element
Vh u, v;

// Macro
macro grad(u) [dx(u), dy(u)] //

// Problem
problem Poisson (u, v, solver=CG, eps=-1.e-6)
   = int2d(Th)(
        grad(u)' * grad(v)
   )
   + int2d(Th)(
        f * v
   )
```

(continues on next page)

(a) Adapted mesh **(b)** Solution on adapted mesh

Fig. 6.1: Poisson

(continued from previous page)

```
   + on(b1, b2, b3, b4, b5, b6, u=g)
   ;

// Mesh adaptation iterations
real error = 0.1;
real coef = 0.1^(1./5.);
for (int i = 0; i < NAdapt; i++){
   // Solve
   Poisson;

   // Plot
   plot(Th, u);

   // Adaptmesh
   Th = adaptmesh(Th, u, inquire=1, err=error);
   error = error * coef;
}
```

6.1.2 Poisson's equation 3D

```
load "tetgen"

// Parameters
real hh = 0.1;
func ue = 2.*x*x + 3.*y*y + 4.*z*z + 5.*x*y + 6.*x*z + 1.;
func f= -18.;

// Mesh
mesh Th = square(10, 20, [x*pi-pi/2, 2*y*pi]); // ]-pi/2, pi/2[X]0,2pi[
func f1 = cos(x)*cos(y);
func f2 = cos(x)*sin(y);
```

(continues on next page)

(continued from previous page)

```
func f3 = sin(x);
func f1x = sin(x)*cos(y);
func f1y = -cos(x)*sin(y);
func f2x = -sin(x)*sin(y);
func f2y = cos(x)*cos(y);
func f3x = cos(x);
func f3y = 0;
func m11 = f1x^2 + f2x^2 + f3x^2;
func m21 = f1x*f1y + f2x*f2y + f3x*f3y;
func m22 = f1y^2 + f2y^2 + f3y^2;
func perio = [[4, y], [2, y], [1, x], [3, x]];
real vv = 1/square(hh);
Th = adaptmesh(Th, m11*vv, m21*vv, m22*vv, IsMetric=1, periodic=perio);
Th = adaptmesh(Th, m11*vv, m21*vv, m22*vv, IsMetric=1, periodic=perio);
plot(Th);

real[int] domain = [0., 0., 0., 1, 0.01];
mesh3 Th3 = tetgtransfo(Th, transfo=[f1, f2, f3], nbofregions=1, regionlist=domain);
plot(Th3);

border cc(t=0, 2*pi){x=cos(t); y=sin(t); label=1;}
mesh Th2 = buildmesh(cc(50));

// Fespace
fespace Vh(Th3, P23d);
Vh u, v;
Vh uhe = ue;
cout << "uhe min: " << uhe[].min << " - max: " << uhe[].max << endl;
cout << uhe(0.,0.,0.) << endl;

fespace Vh2(Th2, P2);
Vh2 u2, u2e;

// Macro
macro Grad3(u) [dx(u), dy(u), dz(u)] //

// Problem
problem Lap3d (u, v, solver=CG)
    = int3d(Th3)(
          Grad3(v)' * Grad3(u)
    )
    - int3d(Th3)(
          f * v
    )
    + on(0, 1, u=ue)
    ;

// Solve
Lap3d;
cout << "u min: " << u[]. min << " - max: " << u[].max << endl;

// Error
real err = int3d(Th3)(square(u-ue));
cout << int3d(Th3)(1.) << " = " << Th3.measure << endl;
Vh d = ue - u;
cout << " err = " << err << " - diff l^intfy = " << d[].linfty << endl;

```

(continues on next page)

(continued from previous page)

```
// Plot
u2 = u;
u2e = ue;
plot(u2, wait=true);
plot(u2, u2e,wait=true);
```

Fig. 6.2: Iso-surfaces of the solution

6.1.3 Stokes Equation on a cube

```
load "msh3"
load "medit" // Dynamically loaded tools for 3D

// Parameters
int nn = 8;

// Mesh
mesh Th0 = square(nn, nn);
int[int] rup = [0, 2];
int[int] rdown = [0, 1];
int[int] rmid = [1, 1, 2, 1, 3, 1, 4, 1];
real zmin = 0, zmax = 1;
mesh3 Th = buildlayers(Th0, nn, zbound=[zmin, zmax],
    reffacemid=rmid, reffaceup=rup, reffacelow=rdown);

medit("c8x8x8", Th); // 3D mesh visualization with medit

// Fespaces
fespace Vh2(Th0, P2);
Vh2 ux, uz, p2;

fespace VVh(Th, [P2, P2, P2, P1]);
VVh [u1, u2, u3, p];
VVh [v1, v2, v3, q];

// Macro
macro Grad(u) [dx(u), dy(u), dz(u)] //
```

(continues on next page)

(a) Solution **(b)** Associated mesh

Fig. 6.3: Stokes

(continued from previous page)

```
macro div(u1,u2,u3) (dx(u1) + dy(u2) + dz(u3)) //

// Problem (directly solved)
solve vStokes ([u1, u2, u3, p], [v1, v2, v3, q])
    = int3d(Th, qforder=3)(
          Grad(u1)' * Grad(v1)
        + Grad(u2)' * Grad(v2)
        + Grad(u3)' * Grad(v3)
        - div(u1, u2, u3) * q
        - div(v1, v2, v3) * p
        + 1e-10 * q * p
    )
    + on(2, u1=1., u2=0, u3=0)
    + on(1, u1=0, u2=0, u3=0)
    ;

// Plot
plot(p, wait=1, nbiso=5); // 3D visualization of pressure isolines

// See 10 plan of the velocity in 2D
for(int i = 1; i < 10; i++){
    // Cut plane
    real yy = i/10.;
    // 3D to 2D interpolation
    ux = u1(x,yy,y);
    uz = u3(x,yy,y);
    p2 = p(x,yy,y);
    // Plot
    plot([ux, uz], p2, cmm="cut y = "+yy, wait= 1);
}
```

6.1.4 Cavity

```
//Parameters
int m = 300;
real L = 1;
real rho = 500.;
real mu = 0.1;

real uin = 1;
func fx = 0;
func fy = 0;
int[int] noslip = [1, 2, 4];
int[int] inflow = [3];

real dt = 0.1;
real T = 50;

real eps = 1e-3;

//Macros
macro div(u) (dx(u#x) + dy(u#y))//
macro grad(u) [dx(u), dy(u)]//
macro Grad(u) [grad(u#x), grad(u#y)]//

//Time
real cpu;
real tabcpu;

//mesh
border C1(t = 0, L){ x = t; y = 0; label = 1; }
border C2(t = 0, L){ x = L; y = t; label = 2; }
border C3(t = 0, L){ x = L-t; y = L; label = 3; }
border C4(t = 0, L){ x = 0; y = L-t; label = 4; }
mesh th = buildmesh( C1(m) + C2(m) + C3(m) + C4(m) );

fespace UPh(th, [P2,P2,P1]);
UPh [ux, uy, p];
UPh [uhx, uhy, ph];
UPh [upx, upy, pp];

//Solve
varf navierstokes([ux, uy, p], [uhx, uhy, ph])
  = int2d(th)(
      rho/dt* [ux, uy]'* [uhx, uhy]
    + mu* (Grad(u):Grad(uh))
    - p* div(uh)
    - ph* div(u)
    - 1e-10 *p*ph
    )

  + int2d(th) (
      [fx, fy]' * [uhx, uhy]
    + rho/dt* [convect([upx, upy], -dt, upx), convect([upx, upy], -dt, upy)]'* [uhx, uhy]
    )

  + on(noslip, ux=0, uy=0)
```

(continues on next page)

(continued from previous page)

```
  + on(inflow, ux=uin, uy=0)
  ;

//Initialization
[ux, uy, p]=[0, 0, 0];

matrix<real> NS = navierstokes(UPh, UPh, solver=sparsesolver);
real[int] NSrhs = navierstokes(0, UPh);

//Time loop
for(int j = 0; j < T/dt; j++){
  [upx, upy, pp]=[ux, uy, p];

  NSrhs = navierstokes(0, UPh);
  ux[] = NS^-1 * NSrhs;

  plot( [ux,uy], p, wait=0, cmm=j);
}

//CPU
cout << " CPU = " << clock()-cpu << endl ;
tabcpu = clock()-cpu;
```

6.2 Mesh Generation

6.2.1 Square mesh

```
mesh Th0 = square(10 ,10);

mesh Th1 = square(4, 5);

real x0 = 1.2;
real x1 = 1.8;
real y0 = 0;
real y1 = 1;
int n = 5;
real m = 20;
mesh Th2 = square(n, m, [x0+(x1-x0)*x, y0+(y1-y0)*y]);

for (int i = 0; i < 5; ++i){
   int[int] labs = [11, 12, 13, 14];
   mesh Th3 = square(3, 3, flags=i, label=labs, region=10);
   plot(Th3, wait=1, cmm="square flags = "+i );
}
```

6.2.2 Mesh adaptation

```
// Parameters
real eps = 0.0001;
real h = 1;
real hmin = 0.05;
```

(continues on next page)

(a) Initial mesh

(b) Adapted mesh

Fig. 6.4: Mesh adaptation

(continued from previous page)

```
func f = 10.0*x^3 + y^3 + h*atan2(eps, sin(5.0*y)-2.0*x);

// Mesh
mesh Th = square(5, 5, [-1+2*x, -1+2*y]);

// Fespace
fespace Vh(Th,P1);
Vh fh = f;
plot(fh);

// Adaptmesh
for (int i = 0; i < 2; i++){
   Th = adaptmesh(Th, fh);
   fh = f; //old mesh is deleted
   plot(Th, fh, wait=true);
}
```

6.2.3 Mesh adaptation for the Poisson's problem

```
// Parameters
real error = 0.1;

// Mesh
border ba(t=0, 1){x=t; y=0; label=1;}
border bb(t=0, 0.5){x=1; y=t; label=1;}
border bc(t=0, 0.5){x=1-t; y=0.5; label=1;}
border bd(t=0.5, 1){x=0.5; y=t; label=1;}
border be(t=0.5, 1){x=1-t; y=1; label=1;}
border bf(t=0, 1){x=0; y=1-t; label=1;}
mesh Th = buildmesh(ba(6) + bb(4) + bc(4) + bd(4) + be(4) + bf(6));

// Fespace
fespace Vh(Th, P1);
Vh u, v;

```

(continues on next page)

(continued from previous page)

```
// Function
func f = 1;

// Problem
problem Poisson(u, v, solver=CG, eps=1.e-6)
   = int2d(Th)(
        dx(u)*dx(v)
      + dy(u)*dy(v)
   )
   - int2d(Th)(
        f*v
   )
   + on(1, u=0);

// Adaptmesh loop
for (int i = 0; i < 4; i++){
   Poisson;
   Th = adaptmesh(Th, u, err=error);
   error = error/2;
}

// Plot
plot(u);
```

6.2.4 Uniform mesh adaptation

```
mesh Th = square(2, 2); // The initial mesh
plot(Th, wait=true);

Th = adaptmesh(Th, 1./30., IsMetric=1, nbvx=10000);
plot(Th, wait=true);

Th = adaptmesh(Th, 1./30., IsMetric=1, nbvx=10000); // More than one time due to the
Th = adaptmesh(Th, 1./30., IsMetric=1, nbvx=10000); // adaptation bound `maxsubdiv=`
plot(Th, wait=true);
```

6.2.5 Borders

```
{
   int upper = 1;
   int others = 2;
   int inner = 3;

   border C01(t=0, 1){x=0; y=-1+t; label=upper;}
   border C02(t=0, 1){x=1.5-1.5*t; y=-1; label=upper;}
   border C03(t=0, 1){x=1.5; y=-t; label=upper;}
   border C04(t=0, 1){x=1+0.5*t; y=0; label=others;}
   border C05(t=0, 1){x=0.5+0.5*t; y=0; label=others;}
   border C06(t=0, 1){x=0.5*t; y=0; label=others;}
   border C11(t=0, 1){x=0.5; y=-0.5*t; label=inner;}
   border C12(t=0, 1){x=0.5+0.5*t; y=-0.5; label=inner;}
   border C13(t=0, 1){x=1; y=-0.5+0.5*t; label=inner;}
```

(continues on next page)

(a) Initial mesh

(b) Adapted mesh

(c) Solution on adapted mesh

Fig. 6.5: Mesh adaptation (Poisson)

(a) Initial mesh

(b) Adapted mesh

Fig. 6.6: Uniform mesh adaptation

(continued from previous page)

```

   int n = 10;
   plot(C01(-n) + C02(-n) + C03(-n) + C04(-n) + C05(-n)
      + C06(-n) + C11(n) + C12(n) + C13(n), wait=true);

   mesh Th = buildmesh(C01(-n) + C02(-n) + C03(-n) + C04(-n) + C05(-n)
      + C06(-n) + C11(n) + C12(n) + C13(n));

   plot(Th, wait=true);

   cout << "Part 1 has region number " << Th(0.75, -0.25).region << endl;
   cout << "Part 2 has redion number " << Th(0.25, -0.25).region << endl;
}

{
   border a(t=0, 2*pi){x=cos(t); y=sin(t); label=1;}
   border b(t=0, 2*pi){x=0.3+0.3*cos(t); y=0.3*sin(t); label=2;}
   plot(a(50) + b(30)); //to see a plot of the border mesh
   mesh Thwithouthole = buildmesh(a(50) + b(30));
   mesh Thwithhole = buildmesh(a(50) + b(-30));
   plot(Thwithouthole);
   plot(Thwithhole);
}

{
   real r=1;
   border a(t=0, 2*pi){x=r*cos(t); y=r*sin(t); label=1;}
   r=0.3;
   border b(t=0, 2*pi){x=r*cos(t); y=r*sin(t); label=1;}
   //  mesh Thwithhole = buildmesh(a(50) + b(-30)); // do not do this because the two
   // circles have the same radius = $0.3$
}
```

6.2.6 Change

```
verbosity=3;

// Mesh
mesh Th1 = square(10, 10);
mesh Th2 = square(20, 10, [x+1, y]);

int[int] r1=[2, 0];
plot(Th1, wait=true);

Th1 = change(Th1, label=r1); // Change edges' label from 2 to 0
plot(Th1, wait=true);

int[int] r2=[4, 0];
Th2 = change(Th2, label=r2); // Change edges' label from 4 to 0
plot(Th2, wait=true);

mesh Th = Th1 + Th2; // 'gluing together' Th1 and Th2 meshes
cout << "nb lab = " << int1d(Th1,1,3,4)(1./lenEdge)+int1d(Th2,1,2,3)(1./lenEdge)
   << " == " << int1d(Th,1,2,3,4)(1./lenEdge) << " == " << ((10+20)+10)*2 << endl;
plot(Th, wait=true);
```

(continues on next page)

(a) Mesh with two regions

(b) Mesh without a hole

(c) Mesh with a hole

Fig. 6.7: Borders

(continued from previous page)

```

fespace Vh(Th, P1);
Vh u, v;

macro Grad(u) [dx(u),dy(u)] // Definition of a macro

solve P(u, v)
   = int2d(Th)(
        Grad(u)'*Grad(v)
   )
   -int2d(Th)(
        v
   )
   + on(1, 3, u=0)
   ;

plot(u, wait=1);
```

Fig. 6.8: Result

6.2.7 Cube

```
load "msh3"

int[int] l6 = [37, 42, 45, 40, 25, 57];
int r11 = 11;
mesh3 Th = cube(4, 5, 6, [x*2-1, y*2-1, z*2-1], label=l6, flags =3, region=r11);

cout << "Volume = " << Th.measure << ", border area = " << Th.bordermeasure << endl;

int err = 0;
for(int i = 0; i < 100; ++i){
   real s = int2d(Th,i)(1.);
   real sx = int2d(Th,i)(x);
   real sy = int2d(Th,i)(y);
   real sz = int2d(Th,i)(z);

```

(continues on next page)

(continued from previous page)

```
   if(s){
      int ix = (sx/s+1.5);
      int iy = (sy/s+1.5);
      int iz = (sz/s+1.5);
      int ii = (ix + 4*(iy+1) + 16*(iz+1) );
      //value of ix,iy,iz => face min 0, face max 2, no face 1
      cout << "Label = " << i << ", s = " << s << " " << ix << iy << iz << " : " << ii << endl;
      if( i != ii ) err++;
   }
}
real volr11 = int3d(Th,r11)(1.);
cout << "Volume region = " << 11 << ": " << volr11 << endl;
if((volr11 - Th.measure )>1e-8) err++;
plot(Th, fill=false);
cout << "Nb err = " << err << endl;
assert(err==0);
```

Fig. 6.9: Cube

6.2.8 Empty mesh

```
{
   border a(t=0, 2*pi){x=cos(t); y=sin(t); label=1;}
   mesh Th = buildmesh(a(20));
   Th = emptymesh(Th);
   plot(Th);
}
{
   mesh Th = square(10, 10);
   int[int] ssd(Th.nt);
   // Builds the pseudo region numbering
   for(int i = 0; i < ssd.n; i++){
      int iq = i/2; // Because we have 2 triangles per quad
      int ix = iq%10;
      int iy = iq/10;
      ssd[i] = 1 + (ix>=5) + (iy>=5)*2;
```

(continues on next page)

(a) Empty square **(b)** Empty diamond

Fig. 6.10: Empty mesh

(continued from previous page)

```
    }
    // Builds an emtpy mesh with all edges that satisfy e=T1 cap T2 and ssd[T1] !=
↪ssd[T2]
    Th = emptymesh(Th, ssd);
    // Plot
    plot(Th);
}
```

6.2.9 3 points

```
// Square for Three-Point Bend Specimens fixed on Fix1, Fix2
// It will be loaded on Load
real a = 1, b = 5, c = 0.1;
int n = 5, m = b*n;
border Left(t=0, 2*a){x=-b; y=a-t;}
border Bot1(t=0, b/2-c){x=-b+t; y=-a;}
border Fix1(t=0, 2*c){x=-b/2-c+t; y=-a;}
border Bot2(t=0, b-2*c){x=-b/2+c+t; y=-a;}
border Fix2(t=0, 2*c){x=b/2-c+t; y=-a;}
border Bot3(t=0, b/2-c){x=b/2+c+t; y=-a;}
border Right(t=0, 2*a){x=b; y=-a+t;}
border Top1(t=0, b-c){x=b-t; y=a;}
border Load(t=0, 2*c){x=c-t; y=a;}
border Top2(t=0, b-c){x=-c-t; y=a;}

mesh Th = buildmesh(Left(n) + Bot1(m/4) + Fix1(5) + Bot2(m/2)
    + Fix2(5) + Bot3(m/4) + Right(n) + Top1(m/2) + Load(10) + Top2(m/2));
plot(Th, bw=true);
```

6.2.10 Bezier

```
// A cubic Bezier curve connecting two points with two control points
func real bzi(real p0, real p1, real q1, real q2, real t){
```

(continues on next page)

Fig. 6.11: 3 Points

(continued from previous page)

```
   return p0*(1-t)^3 + q1*3*(1-t)^2*t + q2*3*(1-t)*t^2 + p1*t^3;
}

real[int] p00 = [0, 1], p01 = [0, -1], q00 = [-2, 0.1], q01 = [-2, -0.5];
real[int] p11 = [1,-0.9], q10 = [0.1, -0.95], q11=[0.5, -1];
real[int] p21 = [2, 0.7], q20 = [3, -0.4], q21 = [4, 0.5];
real[int] q30 = [0.5, 1.1], q31 = [1.5, 1.2];
border G1(t=0, 1){
   x=bzi(p00[0], p01[0], q00[0], q01[0], t);
   y=bzi(p00[1], p01[1], q00[1], q01[1], t);
}
border G2(t=0, 1){
   x=bzi(p01[0], p11[0], q10[0], q11[0], t);
   y=bzi(p01[1], p11[1], q10[1], q11[1], t);
}
border G3(t=0, 1){
   x=bzi(p11[0], p21[0], q20[0], q21[0], t);
   y=bzi(p11[1], p21[1], q20[1], q21[1], t);
}
border G4(t=0, 1){
   x=bzi(p21[0], p00[0], q30[0], q31[0], t);
   y=bzi(p21[1], p00[1], q30[1], q31[1], t);
}
int m = 5;
mesh Th = buildmesh(G1(2*m) + G2(m) + G3(3*m) + G4(m));
plot(Th, bw=true);
```

6.2.11 Build layer mesh

```
load "msh3"
load "tetgen"
load "medit"

// Parameters
int C1 = 99;
```

(continues on next page)

Fig. 6.12: Bezier

(continued from previous page)

```
int C2 = 98;

// 2D mesh
border C01(t=0, pi){x=t; y=0; label=1;}
border C02(t=0, 2*pi){ x=pi; y=t; label=1;}
border C03(t=0, pi){ x=pi-t; y=2*pi; label=1;}
border C04(t=0, 2*pi){ x=0; y=2*pi-t; label=1;}

border C11(t=0, 0.7){x=0.5+t; y=2.5; label=C1;}
border C12(t=0, 2){x=1.2; y=2.5+t; label=C1;}
border C13(t=0, 0.7){x=1.2-t; y=4.5; label=C1;}
border C14(t=0, 2){x=0.5; y=4.5-t; label=C1;}

border C21(t=0, 0.7){x=2.3+t; y=2.5; label=C2;}
border C22(t=0, 2){x=3; y=2.5+t; label=C2;}
border C23(t=0, 0.7){x=3-t; y=4.5; label=C2;}
border C24(t=0, 2){x=2.3; y=4.5-t; label=C2;}

mesh Th = buildmesh(C01(10) + C02(10) + C03(10) + C04(10)
   + C11(5) + C12(5) + C13(5) + C14(5)
   + C21(-5) + C22(-5) + C23(-5) + C24(-5));

mesh Ths = buildmesh(C01(10) + C02(10) + C03(10) + C04(10)
   + C11(5) + C12(5) + C13(5) + C14(5));

// Construction of a box with one hole and two regions
func zmin = 0.;
func zmax = 1.;
int MaxLayer = 10;

func XX = x*cos(y);
func YY = x*sin(y);
func ZZ = z;

int[int] r1 = [0, 41], r2 = [98, 98, 99, 99, 1, 56];
int[int] r3 = [4, 12]; // Change upper surface mesh's triangles labels
// generated by the 2D mesh's triangles Th
```

(continues on next page)

(a) Box with a hole

(b) Sphere

Fig. 6.13: Build layer mesh

(continued from previous page)

```
// from label 4 to label 12
int[int] r4 = [4, 45]; // Change lower surface mesh's triangles labels
// generated by the 2D mesh's triangles Th
// from label 4 to label 45

mesh3 Th3 = buildlayers(Th, MaxLayer, zbound=[zmin, zmax], region=r1,
   labelmid=r2, labelup=r3, labeldown=r4);
medit("box 2 regions 1 hole", Th3);

// Construction of a sphere with TetGen
func XX1 = cos(y)*sin(x);
func YY1 = sin(y)*sin(x);
func ZZ1 = cos(x);

real[int] domain = [0., 0., 0., 0, 0.001];
string test = "paACQ";
cout << "test = " << test << endl;
mesh3 Th3sph = tetgtransfo(Ths, transfo=[XX1, YY1, ZZ1],
   switch=test, nbofregions=1, regionlist=domain);
medit("sphere 2 regions", Th3sph);
```

6.2.12 Sphere

```
// Parameter
real hh = 0.1;

// Mesh 2D
mesh Th = square(10, 20, [x*pi-pi/2, 2*y*pi]); // ]-pi/2, pi/2[X]0, 2pi[
// A parametrization of a sphere
func f1 = cos(x)*cos(y);
func f2 = cos(x)*sin(y);
func f3 = sin(x);
// Partial derivative of the parametrization DF
func f1x = sin(x)*cos(y);
func f1y = -cos(x)*sin(y);
```

(continues on next page)

(a) Initial mesh

(b) Sphere

Fig. 6.14: Sphere

(continued from previous page)

```
func f2x = -sin(x)*sin(y);
func f2y = cos(x)*cos(y);
func f3x = cos(x);
func f3y = 0;
//M = DF^t DF
func m11 = f1x^2 + f2x^2 + f3x^2;
func m21 = f1x*f1y + f2x*f2y + f3x*f3y;
func m22 = f1y^2 + f2y^2 + f3y^2;

// Periodic condition
func perio = [[4, y], [2, y], [1, x], [3, x]];

// Mesh adaptation
real vv = 1/square(hh);
Th = adaptmesh(Th, m11*vv, m21*vv, m22*vv, IsMetric=1, inquire=1, periodic=perio);
Th = adaptmesh(Th, m11*vv, m21*vv, m22*vv, IsMetric=1, periodic=perio);
Th = adaptmesh(Th, m11*vv, m21*vv, m22*vv, IsMetric=1, periodic=perio);
Th = adaptmesh(Th, m11*vv, m21*vv, m22*vv, IsMetric=1, periodic=perio);

// Sphere
mesh3 Th3 = movemesh23(Th, transfo=[f1, f2, f3]);
plot(Th3);
```

6.3 Finite Element

6.3.1 Periodic 3D

```
load "msh3"
load "medit"

// Parameters
searchMethod=1; // More safe seach algo
real a = 1, d = 0.5, h = 0.5;
int nnb = 7, nni = 10;
```

(continues on next page)

(continued from previous page)

```
int nz = 3;
func zmin = 0;
func zmax = h;

// Mesh 2D
border b1(t=0.5, -0.5){x=a*t; y=-a/2; label=1;}
border b2(t=0.5, -0.5){x=a/2; y=a*t; label=2;}
border b3(t=0.5, -0.5){x=a*t; y=a/2; label=3;}
border b4(t=0.5, -0.5){x=-a/2; y=a*t; label=4;}
border i1(t=0, 2.*pi){x=d/2*cos(t); y=-d/2*sin(t); label=7;}
mesh Th = buildmesh(b1(-nnb) + b3(nnb) + b2(-nnb) + b4(nnb) + i1(nni));

{ // Cleaning the memory correctly
    int[int] old2new(0:Th.nv-1);
    fespace Vh2(Th, P1);
    Vh2 sorder = x + y;
    sort(sorder[], old2new);
    int[int] new2old = old2new^-1; // Inverse permutation
    Th = change(Th, renumv=new2old);
    sorder[] = 0:Th.nv-1;
}
{
    fespace Vh2(Th, P1);
    Vh2 nu;
    nu[] = 0:Th.nv-1;
    plot(nu, cmm="nu=", wait=true);
}

// Mesh 3D
int[int] rup = [0, 5], rlow = [0, 6], rmid = [1, 1, 2, 2, 3, 3, 4, 4, 7, 7], rtet = [0, 41];
mesh3 Th3 = buildlayers(Th, nz, zbound=[zmin, zmax],
    reftet=rtet, reffacemid=rmid, reffaceup=rup, reffacelow=rlow);
for(int i = 1; i <= 6; ++i)
    cout << " int " << i << " : " << int2d(Th3,i)(1.) << " " << int2d(Th3,i)(1./area) << endl;

plot(Th3, wait=true);
medit("Th3", Th3);

fespace Vh(Th3, P2, periodic=[[1, x, z], [3, x, z], [2, y, z], [4, y, z], [5, x, y], [6, x, y]]);
```

6.3.2 Lagrange multipliers

```
// Parameters
func f = 1 + x - y;

// Mesh
mesh Th = square(10, 10);

// Fespace
fespace Vh(Th, P1);
int n = Vh.ndof;
int n1 = n+1;
```

(continues on next page)

Fig. 6.15: Periodic mesh

(continued from previous page)

```
Vh uh, vh;

// Problem
varf va (uh, vh)
    = int2d(Th)(
          dx(uh)*dx(vh)
        + dy(uh)*dy(vh)
    )
    ;

varf vL (uh, vh) = int2d(Th)(f*vh);
varf vb (uh, vh) = int2d(Th)(1.*vh);

matrix A = va(Vh, Vh);
real[int] b = vL(0, Vh);
real[int] B = vb(0, Vh);

// Block matrix
matrix AA = [ [ A, B ], [ B', 0 ] ];
set(AA, solver=sparsesolver);

real[int] bb(n+1), xx(n+1), b1(1), l(1);
b1 = 0;
// Builds the right hand side block
bb = [b, b1];

// Solve
xx = AA^-1 * bb;

// Set values
[uh[],l] = xx;

// Display
cout << " l = " << l(0) << " , b(u, 1) =" << B'*uh[] << endl;

// Plot
plot(uh);
```

Fig. 6.16: Result

6.4 Visualization

6.4.1 Plot

```
mesh Th = square(5,5);
fespace Vh(Th, P1);

// Plot scalar and vectorial FE function
Vh uh=x*x+y*y, vh=-y^2+x^2;
plot(Th, uh, [uh, vh], value=true, wait=true);

// Zoom on box defined by the two corner points [0.1,0.2] and [0.5,0.6]
plot(uh, [uh, vh], bb=[[0.1, 0.2], [0.5, 0.6]],
    wait=true, grey=true, fill=true, value=true);

// Compute a cut
int n = 10;
real[int] xx(10), yy(10);
for (int i = 0; i < n; i++){
    x = i/real(n);
    y = i/real(n);
    xx[i] = i;
    yy[i] = uh; // Value of uh at point (i/10., i/10.)
}
plot([xx, yy], wait=true);

{ // File for gnuplot
    ofstream gnu("plot.gp");
    for (int i = 0; i < n; i++)
        gnu << xx[i] << " " << yy[i] << endl;
}

// Calls the gnuplot command, waits 5 seconds and generates a postscript plot (UNIX␣
↪ONLY)
exec("echo 'plot \"plot.gp\" w l \n pause 5 \n set term postscript \n set output \
↪"gnuplot.eps\" \n replot \n quit' | gnuplot");
```

(a) First plot

(b) Second plot

(c) Gnuplot

Fig. 6.17: Plot

6.4.2 HSV

```
// From: http://en.wikipedia.org/wiki/HSV_color_space
// The HSV (Hue, Saturation, Value) model defines a color space
// in terms of three constituent components:
// HSV color space as a color wheel
// Hue, the color type (such as red, blue, or yellow):
// Ranges from 0-360 (but normalized to 0-100% in some applications like here)
// Saturation, the "vibrancy" of the color: Ranges from 0-100%
// The lower the saturation of a color, the more "grayness" is present
// and the more faded the color will appear.
// Value, the brightness of the color: Ranges from 0-100%

mesh Th = square(10, 10, [2*x-1, 2*y-1]);

fespace Vh(Th, P1);
Vh uh=2-x*x-y*y;

real[int] colorhsv=[ // Color hsv model
    4./6., 1 , 0.5, // Dark blue
    4./6., 1 , 1, // Blue
    5./6., 1 , 1, // Magenta
    1, 1. , 1, // Red
    1, 0.5 , 1 // Light red
    ];
 real[int] viso(31);

 for (int i = 0; i < viso.n; i++)
    viso[i] = i*0.1;

 plot(uh, viso=viso(0:viso.n-1), value=true, fill=true, wait=true, hsv=colorhsv);
```

Fig. 6.18: Result

6.4.3 Medit

```
load "medit"

mesh Th = square(10, 10, [2*x-1, 2*y-1]);
```

(continues on next page)

(a) 2D plot

(b) Plot with elevation

Fig. 6.19: Medit

(continued from previous page)

```

fespace Vh(Th, P1);
Vh u=2-x*x-y*y;

medit("u", Th, u);

// Old way
savemesh(Th, "u", [x, y, u*.5]); // Saves u.points and u.faces file
// build a u.bb file for medit
{
    ofstream file("u.bb");
    file << "2 1 1 " << u[].n << " 2 \n";
    for (int j = 0; j < u[].n; j++)
        file << u[][j] << endl;
}
// Calls medit command
exec("ffmedit u");
// Cleans files on unix-like OS
exec("rm u.bb u.faces u.points");
```

6.4.4 Paraview

```
load "iovtk"

mesh Th = square(10, 10, [2*x-1, 2*y-1]);

fespace Vh(Th, P1);
Vh u=2-x*x-y*y;

int[int] Order = [1];
string DataName = "u";
savevtk("u.vtu", Th, u, dataname=DataName, order=Order);
```

Fig. 6.20: Result

6.5 Algorithms & Optimizations

6.5.1 Algorithms

```
// Parameters
int nerr = 0;
int debugJ = 0;
int debugdJ = 0;
real umax = 0;

// Algorithms tests
{
    func bool stop (int iter, real[int] u, real[int] g){
        cout << " stop = " << iter << " " << u.linfty << " " << g.linfty << endl;
        return g.linfty < 1e-5 || iter > 15;
    }
    // minimization of J(u) = 1./2 * sum (i+1) u_i^2 - b_i
    real[int] b(10), u(10);

    //J
    func real J (real[int] & u){
        real s = 0;
        for (int i = 0; i < u.n; i++)
            s += (i+1)*u[i]*u[i]*0.5 - b[i]*u[i];
        if (debugJ)
            cout << "J = " << s << ", u = " << u[0] << " " << u[1] << endl;
        return s;
    }

    //the gradiant of J (this is a affine version (the RHS is in)
    func real[int] DJ (real[int] &u){
        for (int i = 0; i < u.n; i++)
            u[i] = (i+1)*u[i];
        if (debugdJ)
            cout << "dJ: u = " << u[0] << " " << u[1] << " " << u[2] << endl;
        u -= b;
        if (debugdJ)
```

(continues on next page)

(continued from previous page)

```
            cout << "dJ-b: u = " << u[0] << " " << u[1] << " " << u[2] << endl;
        return u; //return of global variable ok
    }

    //the gradiant of the bilinear part of J (the RHS is remove)
    func real[int] DJ0 (real[int] &u){
        for (int i = 0 ; i < u.n; i++)
            u[i] = (i+1)*u[i];
        if(debugdJ)
            cout << "dJ0: u =" << u[0] << " " << u[1] << " " << u[2] << endl;
        return u; //return of global variable ok
    }

    //erro calculation
    func real error (real[int] & u, real[int] & b){
        real s = 0;
        for (int i = 0; i < u.n; i++)
            s += abs((i+1)*u[i] - b[i]);
        return s;
    }

    func real[int] matId (real[int] &u){ return u; }

    int verb=5; //verbosity
    b = 1.; //set right hand side
    u = 0.; //set initial gest

    LinearCG(DJ, u, eps=1.e-6, nbiter=20, precon=matId, verbosity=verb);
    cout << "LinearGC (Affine) : J(u) = " << J(u) << ", err = " << error(u, b) << endl;
    nerr += !(error(u,b) < 1e-5);
    if(nerr) cout << "sol: u = " << u[0] << " " << u[1] << " " << u[2] << endl;

    b = 1;
    u = 0;
    LinearCG(DJ, u, eps=1.e-15, nbiter=20, precon=matId, verbosity=verb, stop=stop);
    cout << "LinearGC (Affine with stop) : J(u) = " << J(u) << ", err = " << error(u, b) << endl;
    nerr += !(error(u,b) < 1e-5);
    if(nerr) cout << "sol: u = " << u[0] << " " << u[1] << " " << u[2] << endl;

    b = 1;
    u = 0;
    LinearCG(DJ0, u, b, eps=1.e-6, nbiter=20, precon=matId, verbosity=verb);
    cout << "LinearGC (Linear) : J(u) = " << J(u) << ", err = " << error(u, b) << endl;
    nerr += !(error(u,b) < 1e-5);
    if(nerr) cout << "sol: u = " << u[0] << " " << u[1] << " " << u[2] << endl;


    b = 1;
    u = 0;
    AffineGMRES(DJ, u, eps=1.e-6, nbiter=20, precon=matId, verbosity=verb);
    cout << "AffineGMRES (Affine) : J(u) = " << J(u) << ", err = " << error(u, b) << endl;
    nerr += !(error(u,b) < 1e-5);
    if(nerr) cout << "sol: u = " << u[0] << " " << u[1] << " " << u[2] << endl;
```

(continues on next page)

(continued from previous page)

```

    b=1;
    u=0;
    LinearGMRES(DJ0, u, b, eps=1.e-6, nbiter=20, precon=matId, verbosity=verb);
    cout << "LinearGMRES (Linear) : J(u) = " << J(u) << ", err = " << error(u, b) << endl;
    nerr += !(error(u,b) < 1e-5);
    if(nerr) cout << "sol: u = " << u[0] << " " << u[1] << " " << u[2] << endl;


    b=1;
    u=0;
    NLCG(DJ, u, eps=1.e-6, nbiter=20, precon=matId, verbosity=verb);
    cout << "NLCG: J(u) = " << J(u) << ", err = " << error(u, b) << endl;
    nerr += !(error(u,b) < 1e-5);
    if(nerr) cout << "sol: u =" << u[0] << " " << u[1] << " " << u[2] << endl;


    //warning: BFGS use a full matrix of size nxn (where n=u.n)
    b=1;
    u=2;
    BFGS(J, DJ, u, eps=1.e-6, nbiter=20, nbiterline=20);
    cout << "BFGS: J(u) = " << J(u) << ", err = " << error(u, b) << endl;
    assert(error(u,b) < 1e-5);
    if(nerr) cout << "sol: u =" << u[0] << " " << u[1] << " " << u[2] << endl;

    assert(nerr==0);
}

{ // A real non linear test
    // Parameters
    real a = 0.001;
    real eps = 1e-6;
    //f(u) = a*u + u-ln(1+u), f'(u) = a+ u/(1+u), f''(u) = 1/(1+u)^2
    func real f(real u) { return u*a+u-log(1+u); }
    func real df(real u) { return a+u/(1+u); }
    func real ddf(real u) { return 1/((1+u)*(1+u)); }

    // Mesh
    mesh Th = square(20, 20);

    // Fespace
    fespace Vh(Th, P1);
    Vh b = 1;
    Vh u = 0;

    fespace Ph(Th, P0);
    Ph alpha; //store df(|nabla u|^2)

    // The functionnal J
    //J(u) = 1/2 int_Omega f(|nabla u|^2) - int_Omega u b
    func real J (real[int] & u){
        Vh w;
        w[] = u;
        real r = int2d(Th)(0.5*f(dx(w)*dx(w) + dy(w)*dy(w)) - b*w);
        cout << "J(u) = " << r << " " << u.min << " " << u.max << endl;
        return r;
```

(continues on next page)

(continued from previous page)

```
    }

    // The gradiant of J
    func real[int] dJ (real[int] & u){
        Vh w;
        w[] = u;
        alpha = df(dx(w)*dx(w) + dy(w)*dy(w));
        varf au (uh, vh)
            = int2d(Th)(
                  alpha*(dx(w)*dx(vh) + dy(w)*dy(vh))
                - b*vh
            )
            + on(1, 2, 3, 4, uh=0)
            ;

        u = au(0, Vh);
        return u; //warning: no return of local array
    }

    // Problem
    alpha = df(dx(u)*dx(u) + dy(u)*dy(u));
    varf alap (uh, vh)
        = int2d(Th)(
              alpha*(dx(uh)*dx(vh) + dy(uh)*dy(vh))
        )
        + on(1, 2, 3, 4, uh=0)
        ;

    varf amass(uh, vh)
        = int2d(Th)(
              uh*vh
        )
        + on(1, 2, 3, 4, uh=0)
        ;

    matrix Amass = amass(Vh, Vh, solver=CG);
    matrix Alap= alap(Vh, Vh, solver=Cholesky, factorize=1);

    // Preconditionner
    func real[int] C(real[int] & u){
        real[int] w = u;
        u = Alap^-1*w;
        return u; //warning: no return of local array variable
    }

    // Solve
    int conv=0;
    for(int i = 0; i < 20; i++){
        conv = NLCG(dJ, u[], nbiter=10, precon=C, veps=eps, verbosity=5);
        if (conv) break;

        alpha = df(dx(u)*dx(u) + dy(u)*dy(u));
        Alap = alap(Vh, Vh, solver=Cholesky, factorize=1);
        cout << "Restart with new preconditionner " << conv << ", eps =" << eps << endl;
    }

```

(continues on next page)

(a) Result `u` **(b)** `df(dx(u)*dx(u) + dy(u)*dy(u))`

Fig. 6.21: Algorithms

(continued from previous page)

```
    // Plot
    plot (u, wait=true, cmm="solution with NLCG");
    umax = u[].max;

    Vh sss= df(dx(u)*dx(u) + dy(u)*dy(u));
    plot (sss, fill=true, value=true);
}

assert(nerr==0);
```

6.5.2 CMAES variational inequality

```
load "ff-cmaes"

// Parameters
int NN = 7;
func f1 = 1.;
func f2 = -1.;
func g1 = 0.;
func g2 = 0.1;
int iter = 0;
int nadapt = 1;
real starttol = 1e-10;
real bctol = 6.e-12;
real pena = 1000.;

// Mesh
mesh Th = square(NN, NN);

// Fespace
fespace Vh(Th, P1);
Vh ou1, ou2;

// Mesh adaptation loops
for (int al = 0; al < nadapt; ++al){
```

(continues on next page)

(continued from previous page)

```
    // Problem
    varf BVF (v, w)
        = int2d(Th)(
              0.5*dx(v)*dx(w)
            + 0.5*dy(v)*dy(w)
        )
        ;
    varf LVF1 (v, w) = int2d(Th)(f1*w);
    varf LVF2 (v, w) = int2d(Th)(f2*w);

    matrix A =  BVF(Vh, Vh);
    real[int] b1 = LVF1(0, Vh);
    real[int] b2 = LVF2(0, Vh);

    varf Vbord (v, w) = on(1, 2, 3, 4, v=1);

    Vh In, Bord;
    Bord[] = Vbord(0, Vh, tgv=1);
    In[] = Bord[] ? 0:1;
    Vh gh1 = Bord*g1;
    Vh gh2 = Bord*g2;

    // Function which creates a vector of the search space type from
    // two finite element functions
    func int FEFToSSP (real[int] &fef1, real[int] &fef2, real[int] &ssp){
        int kX = 0;
        for (int i = 0; i < Vh.ndof; ++i){
            if (In[][i]){
                ssp[kX] = fef1[i];
                ssp[kX+In[].sum] = fef2[i];
                ++kX;
            }
        }
        return 1;
    }

    // Splits a vector from the search space and fills
    // two finite element functions with it
    func int SSPToFEF (real[int] &fef1, real[int] &fef2, real[int] &ssp){
        int kX = 0;
        for (int i = 0; i < Vh.ndof; ++i){
            if (In[][i]){
                fef1[i] = ssp[kX];
                fef2[i] = ssp[kX+In[].sum];
                ++kX;
            }
            else{
                fef1[i] = gh1[][i];
                fef2[i] = gh2[][i];
            }
        }
        return 1;
    }

    func real IneqC (real[int] &X){
        real[int] constraints(In[].sum);
        for (int i = 0; i < In[].sum; ++i){
```

(continues on next page)

(continued from previous page)

```
            constraints[i] = X[i] - X[i+In[].sum];
            constraints[i] = constraints[i] <= 0 ? 0. : constraints[i];
        }
        return constraints.l2;
    }

    func real J (real[int] &X){
        Vh u1, u2;
        SSPToFEF(u1[], u2[], X);
        iter++;
        real[int] Au1 = A*u1[], Au2 = A*u2[];
        Au1 -= b1;
        Au2 -= b2;
        real val = u1[]'*Au1 + u2[]'*Au2;
        val +=  pena * IneqC(X);
        if (iter%200 == 199)
            plot(u1, u2, nbiso=30, fill=1, dim=3, cmm="adapt level "+al+" - iteration
↪"+iter+" - J = "+val, value=1);
        return val ;
    }

    // Solve
    real[int] start(2*In[].sum);

    if (al == 0){
        start(0:In[].sum-1) = 0.;
        start(In[].sum:2*In[].sum-1) = 0.1;
    }
    else
        FEFToSSP(ou1[], ou2[], start);

    real mini = cmaes(J, start, stopMaxFunEval=10000*(al+1), stopTolX=1.e-3/
↪(10*(al+1)), initialStdDev=(0.025/(pow(100.,al))));
    Vh best1, best2;
    SSPToFEF(best1[], best2[], start);

    // Mesh adaptation
    Th = adaptmesh(Th, best1, best2);
    ou1 = best1;
    ou2 = best2;
}
```

6.5.3 IPOPT minimal surface & volume

```
load "msh3";
load "medit";
load "ff-Ipopt";

// Parameters
int nadapt = 3;
real alpha = 0.9;
int np = 30;
real regtest;
int shapeswitch = 1;
real sigma = 2*pi/40.;
```

(continues on next page)

Fig. 6.22: Results

(continued from previous page)

```
real treshold = 0.1;
real e = 0.1;
real r0 = 0.25;
real rr = 2-r0;
real E = 1./(e*e);
real RR = 1./(rr*rr);

// Mesh
mesh Th = square(2*np, np, [2*pi*x, pi*y]);

// Fespace
fespace Vh(Th, P1, periodic=[[2, y], [4, y]]);
//Initial shape definition
//outside of the mesh adaptation loop to initialize with the previous optimial shape found on further iterations
Vh startshape = 5;
Vh uz = 1., lz = 1.;

// Mesh adaptation loop
real[int] lm = [1];
for(int kkk = 0; kkk < nadapt; ++kkk){
    int iter=0;
    func sin2 = square(sin(y));

    // A function which transform Th in 3d mesh (r=rho)
    //a point (theta,phi) of Th becomes ( r(theta,phi)*cos(theta)*sin(phi) , r(theta,phi)*sin(theta)*sin(phi) , r(theta,phi)*cos(phi) )
    //then displays the resulting mesh with medit
    func int Plot3D (real[int] &rho, string cmm, bool ffplot){
        Vh rhoo;
        rhoo[] = rho;
        //mesh sTh = square(np, np/2, [2*pi*x, pi*y]);
        //fespace sVh(sTh, P1);
```

(continues on next page)

(continued from previous page)

```
        //Vh rhoplot = rhoo;
        try{
            mesh3 Sphere = movemesh23(Th, transfo=[rhoo(x,y)*cos(x)*sin(y), rhoo(x,y)*sin(x)*sin(y), rhoo(x,y)*cos(y)]);
            if(ffplot)
                plot(Sphere);
            else
                medit(cmm, Sphere);
        }
        catch(...){
            cout << "PLOT ERROR" << endl;
        }
        return 1;
    }

    // Surface computation
    //Maybe is it possible to use movemesh23 to have the surface function less complicated
    //However, it would not simplify the gradient and the hessian
    func real Area (real[int] &X){
        Vh rho;
        rho[] = X;
        Vh rho2 = square(rho);
        Vh rho4 = square(rho2);
        real res = int2d(Th)(sqrt(rho4*sin2 + rho2*square(dx(rho)) + rho2*sin2*square(dy(rho))));
        ++iter;
        if(1)
            plot(rho, value=true, fill=true, cmm="rho(theta,phi) on [0,2pi]x[0,pi] - S="+res, dim=3);
        else
            Plot3D(rho[], "shape_evolution", 1);
        return res;
    }

    func real[int] GradArea (real[int] &X){
        Vh rho, rho2;
        rho[] = X;
        rho2[] = square(X);
        Vh sqrtPsi, alpha;
        {
            Vh dxrho2 = dx(rho)*dx(rho), dyrho2 = dy(rho)*dy(rho);
            sqrtPsi = sqrt(rho2*rho2*sin2 + rho2*dxrho2 + rho2*dyrho2*sin2);
            alpha = 2.*rho2*rho*sin2 + rho*dxrho2 + rho*dyrho2*sin2;
        }
        varf dArea (u, v)
            = int2d(Th)(
                1./sqrtPsi * (alpha*v + rho2*dx(rho)*dx(v) + rho2*dy(rho)*sin2*dy(v))
            )
            ;

        real[int] grad = dArea(0, Vh);
        return grad;
    }

    matrix hessianA;
    func matrix HessianArea (real[int] &X){
```

(continues on next page)

(continued from previous page)

```
        Vh rho, rho2;
        rho[] = X;
        rho2 = square(rho);
        Vh sqrtPsi, sqrtPsi3, C00, C01, C02, C11, C12, C22, A;
        {
            Vh C0, C1, C2;
            Vh dxrho2 = dx(rho)*dx(rho), dyrho2 = dy(rho)*dy(rho);
            sqrtPsi = sqrt( rho2*rho2*sin2 + rho2*dxrho2 + rho2*dyrho2*sin2);
            sqrtPsi3 = (rho2*rho2*sin2 + rho2*dxrho2 + rho2*dyrho2*sin2)*sqrtPsi;
            C0 = 2*rho2*rho*sin2 + rho*dxrho2 + rho*dyrho2*sin2;
            C1 = rho2*dx(rho);
            C2 = rho2*sin2*dy(rho);
            C00 = square(C0);
            C01 = C0*C1;
            C02 = C0*C2;
            C11 = square(C1);
            C12 = C1*C2;
            C22 = square(C2);
            A = 6.*rho2*sin2 + dxrho2 + dyrho2*sin2;
        }
        varf d2Area (w, v)
            =int2d(Th)(
                1./sqrtPsi * (
                      A*w*v
                    + 2*rho*dx(rho)*dx(w)*v
                    + 2*rho*dx(rho)*w*dx(v)
                    + 2*rho*dy(rho)*sin2*dy(w)*v
                    + 2*rho*dy(rho)*sin2*w*dy(v)
                    + rho2*dx(w)*dx(v)
                    + rho2*sin2*dy(w)*dy(v)
                )
                + 1./sqrtPsi3 * (
                      C00*w*v
                    + C01*dx(w)*v
                    + C01*w*dx(v)
                    + C02*dy(w)*v
                    + C02*w*dy(v)
                    + C11*dx(w)*dx(v)
                    + C12*dx(w)*dy(v)
                    + C12*dy(w)*dx(v)
                    + C22*dy(w)*dy(v)
                )
            )
            ;
        hessianA = d2Area(Vh, Vh);
        return hessianA;
    }

    // Volume computation
    func real Volume (real[int] &X){
        Vh rho;
        rho[] = X;
        Vh rho3 = rho*rho*rho;
        real res = 1./3.*int2d(Th)(rho3*sin(y));
        return res;
    }

```

(continues on next page)

(continued from previous page)

```
    func real[int] GradVolume (real[int] &X){
        Vh rho;
        rho[] = X;
        varf dVolume(u, v) = int2d(Th)(rho*rho*sin(y)*v);
        real[int] grad = dVolume(0, Vh);
        return grad;
    }
    matrix hessianV;
    func matrix HessianVolume(real[int] &X){
        Vh rho;
        rho[] = X;
        varf d2Volume(w, v) = int2d(Th)(2*rho*sin(y)*v*w);
        hessianV = d2Volume(Vh, Vh);
        return hessianV;
    }

    //if we want to use the volume as a constraint function
    //we must wrap it in some freefem functions returning the appropriate type
    //The lagrangian hessian also have to be wrapped since the Volume is not linear␣
↪with
    //respect to rho, it will constribbute to the hessian.
    func real[int] ipVolume (real[int] &X){ real[int] vol = [Volume(X)]; return vol; }
    matrix mdV;
    func matrix ipGradVolume (real[int] &X) { real[int,int] dvol(1,Vh.ndof); dvol(0,
↪:) = GradVolume(X); mdV = dvol; return mdV; }
    matrix HLagrangian;
    func matrix ipHessianLag (real[int] &X, real objfact, real[int] &lambda){
        HLagrangian = objfact*HessianArea(X) + lambda[0]*HessianVolume(X);
        return HLagrangian;
    }

    //building struct for GradVolume
    int[int] gvi(Vh.ndof), gvj=0:Vh.ndof-1;
    gvi = 0;

    Vh rc = startshape; //the starting value
    Vh ub = 1.e19; //bounds definition
    Vh lb = 0;

    func real Gaussian (real X, real Y, real theta, real phi){
        real deltax2 = square((X-theta)*sin(Y)), deltay2 = square(Y-phi);
        return exp(-0.5 * (deltax2 + deltay2) / (sigma*sigma));
    }

    func disc1 = sqrt(1./(RR+(E-RR)*cos(y)*cos(y)))*(1+0.1*cos(7*x));
    func disc2 = sqrt(1./(RR+(E-RR)*cos(x)*cos(x)*sin2));

    if(1){
        lb = r0;
        for (int q = 0; q < 5; ++q){
            func f = rr*Gaussian(x, y, 2*q*pi/5., pi/3.);
            func g = rr*Gaussian(x, y, 2*q*pi/5.+pi/5., 2.*pi/3.);
            lb = max(max(lb, f), g);
        }
        lb = max(lb, rr*Gaussian(x, y, 2*pi, pi/3));
    }
    lb = max(lb, max(disc1, disc2));
```

(continues on next page)

(continued from previous page)

```
    real Vobj = Volume(lb[]);
    real Vnvc = 4./3.*pi*pow(lb[].linfty,3);

    if(1)
        Plot3D(lb[], "object_inside", 1);
    real[int] clb = 0., cub = [(1-alpha)*Vobj + alpha*Vnvc];

    // Call IPOPT
    int res = IPOPT(Area, GradArea, ipHessianLag, ipVolume, ipGradVolume,
            rc[], ub=ub[], lb=lb[], clb=clb, cub=cub, checkindex=1, maxiter=kkk
↪<nadapt-1 ? 40:150,
            warmstart=kkk, lm=lm, uz=uz[], lz=lz[], tol=0.00001, structjacc=[gvi,
↪gvj]);
    cout << "IPOPT: res =" << res << endl ;

    // Plot
    Plot3D(rc[], "Shape_at_"+kkk, 1);
    Plot3D(GradArea(rc[]), "ShapeGradient", 1);

    // Mesh adaptation
    if (kkk < nadapt-1){
        Th = adaptmesh(Th, rc*cos(x)*sin(y), rc*sin(x)*sin(y), rc*cos(y),
            nbvx=50000, periodic=[[2, y], [4, y]]);
        plot(Th, wait=true);
        startshape = rc;
        uz = uz;
        lz = lz;
    }

    regtest = rc[]'*rc[];
}
```

Fig. 6.23: Mesh

6.5.4 CMAES MPI variational inequality

Command:

```
ff-mpirun -np 4 CMAESMPIVariationalInequality.edp -glut ffglut
```

```
load "mpi-cmaes"

// Parameters
int NN = 10;
func f1 = 1.;
func f2 = -1.;
func g1 = 0.;
func g2 = 0.1;
int iter = 0;
int nadapt = 1;
real starttol = 1e-10;
real bctol = 6.e-12;
real pena = 1000;

// Mesh
mesh Th = square(NN, NN);

// Fespace
fespace Vh(Th, P1);
Vh ou1, ou2;

// Mehs adaptation loop
for (int al = 0; al < nadapt; ++al){
    // Problem
    varf BVF (v, w)
        = int2d(Th)(
              0.5*dx(v)*dx(w)
            + 0.5*dy(v)*dy(w)
        )
        ;
    varf LVF1 (v, w) = int2d(Th)(f1*w);
    varf LVF2 (v, w) = int2d(Th)(f2*w);
    matrix A = BVF(Vh, Vh);
    real[int] b1 = LVF1(0, Vh);
    real[int] b2 = LVF2(0, Vh);

    varf Vbord (v, w) = on(1, 2, 3, 4, v=1);

    Vh In, Bord;
    Bord[] = Vbord(0, Vh, tgv=1);
    In[] = Bord[] ? 0:1;
    Vh gh1 = Bord*g1, gh2 = Bord*g2;

    //Function which create a vector of the search space type from
    //two finite element functions
    func int FEFToSSP (real[int] &fef1, real[int] &fef2, real[int] &ssp){
        int kX = 0;
        for (int i = 0; i < Vh.ndof; ++i){
            if (In[][i]){
                ssp[kX] = fef1[i];
                ssp[kX+In[].sum] = fef2[i];
```

(continues on next page)

(continued from previous page)

```
                ++kX;
            }
        }
        return 1;
    }

    //Function spliting a vector from the search space and fills
    //two finite element functions with it
    func int SSPToFEF (real[int] &fef1, real[int] &fef2, real[int] &ssp){
        int kX = 0;
        for (int i = 0; i < Vh.ndof; ++i){
            if (In[][i]){
                fef1[i] = ssp[kX];
                fef2[i] = ssp[kX+In[].sum];
                ++kX;
            }
            else{
                fef1[i] = gh1[][i];
                fef2[i] = gh2[][i];
            }
        }
        return 1;
    }

    func real IneqC (real[int] &X){
        real[int] constraints(In[].sum);
        for (int i = 0; i < In[].sum; ++i){
            constraints[i] = X[i] - X[i+In[].sum];
            constraints[i] = constraints[i] <= 0 ? 0. : constraints[i];
        }
        return constraints.l2;
    }

    func real J (real[int] &X){
        Vh u1, u2;
        SSPToFEF(u1[], u2[], X);
        iter++;
        real[int] Au1 = A*u1[], Au2 = A*u2[];
        Au1 -= b1;
        Au2 -= b2;
        real val = u1[]'*Au1 + u2[]'*Au2;
        val +=  pena * IneqC(X);
        plot(u1, u2, nbiso=30, fill=1, dim=3, cmm="adapt level "+al+" - iteration
↪"+iter+" - J = "+val, value=1);
        return val ;
    }

    // Solve
    real[int] start(2*In[].sum);

    if (al==0){
        start(0:In[].sum-1) = 0.;
        start(In[].sum:2*In[].sum-1) = 0.1;
    }
    else
        FEFToSSP(ou1[], ou2[], start);

```

(continues on next page)

(continued from previous page)

```
    real mini = cmaesMPI(J, start, stopMaxFunEval=10000*(al+1), stopTolX=1.e-4/
↪(10*(al+1)), initialStdDev=(0.025/(pow(100.,al))));
    Vh best1, best2;
    SSPToFEF(best1[], best2[], start);

    // Mesh adaptation
    Th = adaptmesh(Th, best1, best2);
    ou1 = best1;
    ou2 = best2;
}
```

Fig. 6.24: Result

6.6 Parallelization

6.6.1 MPI-GMRES 2D

To launch this script, use for example:

```
ff-mpirun -np 12 MPIGMRES2D.edp -d 1 -k 1 -gmres 2 -n 50
```

```
//usage :
//ff-mpirun [mpi parameter] MPIGMRES2d.edp [-glut ffglut] [-n N] [-k K] [-d D] [-ns]␣
↪[-gmres [0|1]
//arguments:
//-glut ffglut : to see graphicaly the process
//-n N: set the mesh cube split NxNxN
//-d D: set debug flag D must be one for mpiplot
//-k K: to refined by K all element
//-ns: remove script dump
//-gmres
//0: use iterative schwarz algo.
```

(continues on next page)

(continued from previous page)

```
//1: Algo GMRES on residu of schwarz algo
//2: DDM GMRES
//3: DDM GMRES with coarse grid preconditionner (Good one)

load "MPICG"
load "medit"
load "metis"
include "getARGV.idp"
include "MPIplot.idp"
include "MPIGMRESmacro.idp"

searchMethod = 0; //more safe seach algo (warning can be very expensive in case of␣
↪lot of ouside point)
assert(version >= 3.11); //need at least v3.11
real[int] ttt(10);
int ittt=0;
macro settt {ttt[ittt++] = mpiWtime();}//

// Arguments
verbosity = getARGV("-vv", 0);
int vdebug = getARGV("-d", 1);
int ksplit = getARGV("-k", 3);
int nloc = getARGV("-n", 10);
string sff = getARGV("-p", "");
int gmres = getARGV("-gmres", 2);
bool dplot = getARGV("-dp", 0);
int nC = getARGV("-N", max(nloc/10, 4));

if (mpirank==0 && verbosity){
    cout << "ARGV: ";
    for (int i = 0; i < ARGV.n; ++i)
        cout << ARGV[i] << " ";
    cout << endl;
}

if(mpirank==0 && verbosity)
cout << " vdebug: " << vdebug << ", kspilt "<< ksplit << ", nloc "<< nloc << ", sff "<
↪< sff << "." << endl;

// Parameters
int withplot = 0;
bool withmetis = 1;
bool RAS = 1;
string sPk = "P2-2gd";
func Pk = P2;
int sizeoverlaps = 1; //size of overlap
int[int] l111 = [1, 1, 1, 1]; //mesh labels

// MPI function
func bool plotMPIall(mesh &Th, real[int] &u, string cm){
    if(vdebug)
        PLOTMPIALL(mesh, Pk, Th, u, {cmm=cm, nbiso=20, fill=1, dim=3, value=1});
    return 1;
}

// MPI
mpiComm comm(mpiCommWorld,0,0); //trick : make a no split mpiWorld
```

(continues on next page)

(continued from previous page)

```

int npart = mpiSize(comm); //total number of partion
int ipart = mpiRank(comm); //current partition number

int njpart = 0; //Number of part with intersection (a jpart) with ipart without ipart
int[int] jpart(npart); //list of jpart
if(ipart==0)
    cout << " Final N = " << ksplit*nloc << ", nloc = " << nloc << ", split = " <<␣
↪ksplit << endl;
settt

// Mesh
mesh Thg = square(nloc, nloc, label=1111);
mesh ThC = square(nC, nC, label=1111);// Coarse mesh

mesh Thi, Thin; //with overlap, without olverlap

// Fespace
fespace Phg(Thg, P0);
Phg part;

fespace Vhg(Thg, P1);
Vhg unssd; //boolean function: 1 in the subdomain, 0 elswhere

fespace VhC(ThC, P1); // of the coarse problem

// Partitioning
{
    int[int] nupart(Thg.nt);
    nupart = 0;
    if (npart > 1 && ipart == 0)
        metisdual(nupart, Thg, npart);

    broadcast(processor(0, comm), nupart);
    for(int i = 0; i < nupart.n; ++i)
        part[][i] = nupart[i];
}

if (withplot > 1)
    plot(part, fill=1, cmm="dual", wait=1);

// Overlapping partition
Phg suppi = abs(part-ipart) < 0.1;

Thin = trunc(Thg, suppi>0, label=10); // non-overlapping mesh, interfaces have label␣
↪10
int nnn = sizeoverlaps*2;// to be sure
AddLayers(Thg, suppi[], nnn, unssd[]); //see above! suppi and unssd are modified
unssd[] *= nnn; //to put value nnn a 0
real nnn0 = nnn - sizeoverlaps + 0.001;
Thi = trunc(Thg, unssd>nnn0, label=10); //overlapping mesh, interfaces have label 10

settt

// Fespace
fespace Vhi(Thi,P1);
int npij = npart;
```

(continues on next page)

(continued from previous page)

```
Vhi[int] pij(npij); //local partition of unit + pii
Vhi pii;

real nnn1 = +0.001;
{
    /*
    construction of the partition of the unit,
    let phi_i P1 FE function 1 on Thin and zero ouside of Thi and positive
    the partition is build with
    p_i = phi_i/ \sum phi_i

    to build the partition of one domain i
    we nned to find all j such that supp(phi_j) \cap supp(phi_j) is not empty
    <=> int phi_j
    */
    //build a local mesh of thii such that all computation of the unit partition are
    //exact in thii
    mesh Thii = trunc(Thg, unssd>nnn1, label=10); //overlapping mesh, interfaces have␣
↪label 10

    {
        //find all j mes (supp(p_j) cap supp(p_i)) >0
        //compute all phi_j on Thii
        //remark: supp p_i include in Thi

        // Fespace
        fespace Phii(Thii, P0);
        fespace Vhii(Thii, P1);
        Vhi sumphi = 0;
        Vhii phii = 0;

        jpart = 0;
        njpart = 0;
        int nlayer = RAS ? 1 : sizeoverlaps;
        if (ipart == 0)
            cout << "nlayer = " << nlayer << endl;
        pii = max(unssd-nnn+nlayer, 0.)/nlayer;
        if(dplot)
            plot(pii, wait=1, cmm=" 0000");
        sumphi[] += pii[];
        if(dplot)
            plot(sumphi, wait=1, cmm=" summ 0000");

        real epsmes = 1e-10*Thii.area;
        for (int i = 0; i < npart; ++i)
            if (i != ipart){
            Phii suppii = abs(i-part) < 0.2;
            if (suppii[].max > 0.5){
                AddLayers(Thii, suppii[], nlayer, phii[]);
                assert(phii[].min >= 0);
                real interij = int2d(Thi)(phii);
                if (interij > epsmes){
                    pij[njpart] = abs(phii);
                    if(vdebug > 2)
                        cout << " ***** " << int2d(Thi)(real(pij[njpart])<0) << " " <
↪<pij[njpart][].min << " " << phii[].min << endl;
                    assert(int2d(Thi)(real(pij[njpart]) < 0) == 0);
```

(continues on next page)

(continued from previous page)

```
                    if(dplot)
                        plot(pij[njpart], wait=1, cmm=" j = "+ i + " " + njpart);
                    sumphi[] += pij[njpart][];
                    if(dplot)
                        plot(sumphi, wait=1, cmm=" sum j = "+ i + " " + njpart);
                    jpart[njpart++] = i;
                }
            }
        }

        if(dplot)
            plot(sumphi, wait=1, dim=3, cmm="sum ", fill=1);
        pii[] = pii[] ./ sumphi[];
        for (int j = 0; j < njpart; ++j)
            pij[j][] = pij[j][] ./ sumphi[];
        jpart.resize(njpart);
        for (int j = 0; j < njpart; ++j)
            assert(pij[j][].max <= 1);
        {
            cout << ipart << " number of jpart " << njpart << " : ";
            for (int j = 0; j < njpart; ++j)
                cout << jpart[j] << " ";
            cout << endl;
        }
        sumphi[] = pii[];
        for (int j = 0; j < njpart; ++j)
            sumphi[] += pij[j][];
        if(vdebug > 2)
            cout << "sum min " << sumphi[].min << " " << sumphi[].max << endl;
        assert(sumphi[].min > 1.-1e-6 && sumphi[].max < 1.+1e-6);
    }
} //Thii is remove here
// end of the construction of the local partition of the unity ...
// on Thi
if (ipart == 0)
    cout << "End build partition" << endl;

// Computation of number of intersection
//here pii and the pij is the local partition of the unit on
//Thi (mesh with overlap)
if ( dplot){
    plot(Thi, wait=1);
    for(int j = 0; j < njpart; ++j)
        plot(pij[j], cmm=" j="+j, wait=1);
}

//Partition of the unity on Thi
//computation of message
//all j > we have to receive
//data on intersection of the support of pij[0] and pij[j]
settt

if(vdebug)
    plotMPIall(Thi, pii[], "pi_i");

mesh[int] aThij(njpart);
matrix Pii;
```

(continues on next page)

(continued from previous page)

```
matrix[int] sMj(njpart); //M of send to j
matrix[int] rMj(njpart); //M to recv from j
fespace Whi(Thi, Pk);
mesh Thij = Thi;
fespace Whij(Thij, Pk);//

//construction of the mesh intersect i,j part
for(int jp = 0; jp < njpart; ++jp)
    aThij[jp] = trunc(Thi, pij[jp] > 1e-6, label=10); //mesh of the supp of pij

for(int jp = 0; jp < njpart; ++jp)
    aThij[jp] = trunc(aThij[jp], 1, split=ksplit);

Thi = trunc(Thi, 1, split=ksplit);

settt

if (ipart == 0)
    cout << "End build mesh intersection" << endl;

// Construction of transfert matrix
{
    Whi wpii = pii;
    Pii = wpii[];
    for(int jp = 0; jp < njpart; ++jp){
        int j = jpart[jp];
        Thij = aThij[jp];
        matrix I = interpolate(Whij, Whi); //Whji <- Whi
        sMj[jp] = I*Pii; //Whi -> s Whij
        rMj[jp] = interpolate(Whij, Whi, t=1); //Whji -> Whi
        if(vdebug > 10){
            {Whi uuu=1; Whij vvv=-1; vvv[]+=I*uuu[]; cout << jp << " %%% " << vvv[].
↪linfty << endl; assert(vvv[].linfty < 1e-6);}
            {Whi uuu=1; Whij vvv=-1; vvv[]+=rMj[jp]'*uuu[]; cout << jp << " ### " <<␣
↪vvv[].linfty << endl; assert(vvv[].linfty < 1e-6);}
        }
    }
}
if (ipart == 0)
    cout << "End build transfert matrix" << endl;

// Allocate array of send and recv data
InitU(njpart, Whij, Thij, aThij, Usend) //initU(n, Vh, Th, aTh, U)
InitU(njpart, Whij, Thij, aThij, Vrecv)
if (ipart == 0)
    cout << "End init data for send/revc" << endl;

Whi ui, vi;

func bool Update(real[int] &ui, real[int] &vi){
    for(int j = 0; j < njpart; ++j)
        Usend[j][] = sMj[j]*ui;
    SendRecvUV(comm, jpart, Usend, Vrecv)
    vi = Pii*ui;
    for(int j = 0; j < njpart; ++j)
        vi += rMj[j]*Vrecv[j][];
    return true;
```

(continues on next page)

(continued from previous page)

```
}

// Definition of the Problem
func G = x*0.1;
func F = 1.;
macro grad(u) [dx(u),dy(u)] //
varf vBC (U, V) = on(1, U=G);
varf vPb (U, V) = int2d(Thi)(grad(U)'*grad(V)) + int2d(Thi)(F*V) + on(10, U=0) + on(1,
↪ U=G);
varf vPbC (U, V) = int2d(ThC)(grad(U)'*grad(V)) + on(1, U=0);
varf vPbon (U, V) = on(10, U=1) + on(1, U=1);
varf vPbon10only (U, V) = on(10, U=1) + on(1, U=0);
//remark the order is important we want 0 part on 10 and 1

matrix Ai = vPb(Whi, Whi, solver=sparsesolver);
matrix AC, Rci, Pci;

if (mpiRank(comm) == 0)
    AC = vPbC(VhC, VhC, solver=sparsesolver);

Pci = interpolate(Whi, VhC);
Rci = Pci'*Pii;

real[int] onG10 = vPbon10only(0, Whi);
real[int] onG = vPbon(0, Whi);
real[int] Bi=vPb(0, Whi);

int kiter = -1;

func bool CoarseSolve(real[int] &V, real[int] &U, mpiComm &comm){
    //solving the coarse probleme
    real[int] Uc(Rci.n), Bc(Uc.n);
    Uc = Rci*U;
    mpiReduce(Uc, Bc, processor(0, comm), mpiSUM);
    if (mpiRank(comm) == 0)
        Uc = AC^-1*Bc;
    broadcast(processor(0, comm), Uc);
    V = Pci*Uc;
}

func real[int] DJ (real[int] &U){
    ++kiter;
    real[int] V(U.n);
    V = Ai*U;
    V = onG10 ? 0.: V; //remove internal boundary
    return V;
}

func real[int] PDJ (real[int] &U){
    real[int] V(U.n);

    real[int] b = onG10 ? 0. : U;
    V = Ai^-1*b;
    Update(V, U);
    return U;
}

```

(continues on next page)

(continued from previous page)

```
func real[int] PDJC (real[int] &U){
    real[int] V(U.n);
    CoarseSolve(V, U, comm);
    V = -V; //-C2*Uo
    U += Ai*V; //U = (I-A C2) Uo
    real[int] b = onG10 ? 0. : U;
    U = Ai^-1*b; // (C1( I -A C2) Uo
    V = U -V;
    Update(V, U);
    return U;
}

func real[int] DJ0(real[int] &U){
    ++kiter;
    real[int] V(U.n);
    real[int] b = onG .* U;
    b = onG ? b : Bi ;
    V = Ai^-1*b;
    Update(V, U);
    V -= U;
    return V;
}

Whi u = 0, v;
{ //verification
    Whi u = 1, v;
    Update(u[], v[]);
    u[] -= v[];
    assert(u[].linfty < 1e-6);
}

settt
u[] = vBC(0, Whi, tgv=1); //set u with tgv BC value

real epss = 1e-6;
int rgmres = 0;
if (gmres == 1){
    rgmres = MPIAffineGMRES(DJ0, u[], veps=epss, nbiter=300, comm=comm, dimKrylov=100,
↪ verbosity=ipart ? 0: 50);
    real[int] b = onG .* u[];
    b = onG ? b : Bi;
    v[] = Ai^-1*b;
    Update(v[], u[]);
}
else if (gmres == 2)
    rgmres = MPILinearGMRES(DJ, precon=PDJ, u[], Bi, veps=epss, nbiter=300, comm=comm,
↪ dimKrylov=100, verbosity=ipart ? 0: 50);
else if (gmres == 3)
    rgmres = MPILinearGMRES(DJ, precon=PDJC, u[], Bi, veps=epss, nbiter=300,␣
↪comm=comm, dimKrylov=100, verbosity=ipart ? 0: 50);
else //algo Shwarz for demo
    for(int iter = 0; iter < 10; ++iter){
        real[int] b = onG .* u[];
        b = onG ? b : Bi ;
        v[] = Ai^-1*b;

        Update(v[], u[]);
```

(continues on next page)

(continued from previous page)

```
        if(vdebug)
            plotMPIall(Thi, u[], "u-"+iter);
        v[] -= u[];

        real err = v[].linfty;
        real umax = u[].max;
        real[int] aa = [err, umax], bb(2);
        mpiAllReduce(aa, bb, comm, mpiMAX);
        real errg = bb[0];
        real umaxg = bb[1];

        if (ipart == 0)
            cout << ipart << " err = " << errg << " u. max " << umaxg << endl;
        if (errg < 1e-5) break;
    }

if (vdebug)
    plotMPIall(Thi, u[], "u-final");

settt

real errg = 1, umaxg;
{
    real umax = u[].max, umaxg;
    real[int] aa = [umax], bb(1);
    mpiAllReduce(aa, bb, comm, mpiMAX);
    errg = bb[0];
    if (ipart == 0)
        cout << "umax global = " << bb[0] << " Wtime = " << (ttt[ittt-1]-ttt[ittt-2])
↪<< " s " << " " << kiter << endl;
}

if (sff != ""){
    ofstream ff(sff+".txt", append);
    cout << " ++++ ";
    cout << mpirank << "/" << mpisize << " k=" << ksplit << " n= " << nloc << " " <<␣
↪sizeoverlaps << " it= " << kiter;
    for (int i = 1; i < ittt; ++i)
        cout << " " << ttt[i]-ttt[i-1] << " ";
    cout << epss << " " << Ai.nbcoef << " " << Ai.n << endl;

    /*
    1 mpirank
    2 mpisize
    3 ksplit
    4 nloc
    5 sizeoverlaps
    6 kiter
    7 mesh & part build
    8 build the partion
    9 build mesh, transfere , and the fine mesh ..
    10 build the matrix, the trans matrix, factorizatioon
    11 GMRES
    */

    ff << mpirank << " " << mpisize << " " << sPk << " ";
    ff << ksplit << " " << nloc << " " << sizeoverlaps << " " << kiter;
```

(continues on next page)

(continued from previous page)

```
    for (int i = 1; i < ittt; ++i)
        ff << " " << ttt[i]-ttt[i-1] << " ";
    ff << epss << " " << Ai.nbcoef << " " << Ai.n << " " << gmres << endl;
}
```

Fig. 6.25: Results

6.6.2 MPI-GMRES 3D

Todo: todo

6.6.3 Direct solvers

```
load "MUMPS_FreeFem"
//default solver: real-> MUMPS, complex -> MUMPS
load "real_SuperLU_DIST_FreeFem"
default solver: real-> SuperLU_DIST, complex -> MUMPS
load "real_pastix_FreeFem"
//default solver: real-> pastix, complex -> MUMPS

// Solving with pastix
{
    matrix A =
        [[1, 2, 2, 1, 1],
        [ 2, 12, 0, 10 , 10],
        [ 2, 0, 1, 0, 2],
        [ 1, 10, 0, 22, 0.],
        [ 1, 10, 2, 0., 22]];

    real[int] xx = [1, 32, 45, 7, 2], x(5), b(5), di(5);
    b = A*xx;
    cout << "b = " << b << endl;
    cout << "xx = " << xx << endl;

```

(continues on next page)

(continued from previous page)

```
    set(A, solver=sparsesolver, datafilename="ffpastix_iparm_dparm.txt");
    cout << "solve" << endl;
    x = A^-1*b;
    cout << "b = " << b << endl;
    cout << "x = " << endl;
    cout << x << endl;
    di = xx - x;
    if (mpirank == 0){
        cout << "x-xx = " << endl;
        cout << "Linf = " << di.linfty << ", L2 = " << di.l2 << endl;
    }
}

// Solving with SuperLU_DIST
realdefaulttoSuperLUdist();
//default solver: real-> SuperLU_DIST, complex -> MUMPS
{
    matrix A =
        [[1, 2, 2, 1, 1],
        [ 2, 12, 0, 10 , 10],
        [ 2, 0, 1, 0, 2],
        [ 1, 10, 0, 22, 0.],
        [ 1, 10, 2, 0., 22]];

    real[int] xx = [1, 32, 45, 7, 2], x(5), b(5), di(5);
    b = A*xx;
    cout << "b = " << b << endl;
    cout << "xx = " << xx << endl;

    set(A, solver=sparsesolver, datafilename="ffsuperlu_dist_fileparam.txt");
    cout << "solve" << endl;
    x = A^-1*b;
    cout << "b = " << b << endl;
    cout << "x = " << endl;
    cout << x << endl;
    di = xx - x;
    if (mpirank == 0){
        cout << "x-xx = " << endl;
        cout << "Linf = " << di.linfty << ", L2 = " << di.l2 << endl;
    }
}

// Solving with MUMPS
defaulttoMUMPS();
//default solver: real-> MUMPS, complex -> MUMPS
{
    matrix A =
        [[1, 2, 2, 1, 1],
        [ 2, 12, 0, 10 , 10],
        [ 2, 0, 1, 0, 2],
        [ 1, 10, 0, 22, 0.],
        [ 1, 10, 2, 0., 22]];

    real[int] xx = [1, 32, 45, 7, 2], x(5), b(5), di(5);
    b = A*xx;
    cout << "b = " << b << endl;
    cout << "xx = " << xx << endl;
```

(continues on next page)

(continued from previous page)

```

    set(A, solver=sparsesolver, datafilename="ffmumps_fileparam.txt");
    cout << "solving solution" << endl;
    x = A^-1*b;
    cout << "b = " << b << endl;
    cout << "x = " << endl;
    cout << x << endl;
    di = xx - x;
    if (mpirank == 0){
        cout << "x-xx = " << endl;
        cout << "Linf = " << di.linfty << ", L2 " << di.l2 << endl;
    }
}
```

6.6.4 Solver MUMPS

```
load "MUMPS_FreeFem"

// Parameters
int[int] ICNTL(40); //declaration of ICNTL parameter for MUMPS

//get value of ICNTL from file
if (mpirank == 0){
    ifstream ff("ffmumps_fileparam.txt");
    string line;
    getline(ff, line);
    getline(ff, line);
    for (int iii = 0; iii < 40; iii++){
        ff >> ICNTL[iii];
        getline(ff, line);
    }
}

broadcast(processor(0), ICNTL);

// Given data of MUMPS solver in array lparams(SYM, PAR, ICNTL)
// There is no symmetric storage for a matrix associated with a sparse solver.
// Therefore, the matrix will be considered unsymmetric for parallel sparse solver␣
↪even if symmetric.
{
    // Problem
    int SYM = 0;
    int PAR = 1;
    matrix A =
        [
            [40, 0, 45, 0, 0],
            [0, 12, 0, 0, 0],
            [0, 0, 40, 0, 0],
            [12, 0, 0, 22, 0],
            [0, 0, 20, 0, 22]
        ];

    // Construction of integer parameter for MUMPS
    int[int] MumpsLParams(42);
    MumpsLParams[0] = SYM;
```

(continues on next page)

(continued from previous page)

```
    MumpsLParams[1] = PAR;
    for (int ii = 0; ii < 40; ii++)
        MumpsLParams[ii+2] = ICNTL[ii]; //ICNTL begin with index 0 here

    real[int] xx = [1, 32, 45, 7, 2], x(5), b(5), di(5);
    b = A*xx;
    if (mpirank == 0)
        cout << "xx = " << xx << endl;

    set(A, solver=sparsesolver, lparams=MumpsLParams); //we take the default value␣
↪for CNTL MUMPS parameter

    // Solve
    if (mpirank == 0)
        cout << "Solve" << endl;
    x = A^-1*b;
    if (mpirank == 0)
        cout << "b = " << b << endl;
    if (mpirank == 0)
        cout << "x = " << endl; cout << x << endl;
    di = xx-x;
    if (mpirank == 0){
        cout << "x-xx = " << endl;
        cout << "Linf = " << di.linfty << ", L2 = " << di.l2 << endl;
    }
}

// Read parameter of MUMPS solver in file ffmumps_fileparam.txt
{
    // Problem
    matrix A =
        [
            [40, 0, 45, 0, 0],
            [0, 12, 0, 0 , 0],
            [0, 0, 40, 0, 0],
            [12, 0, 0, 22, 0],
            [0, 0, 20, 0, 22]
        ];

    real[int] xx = [1, 32, 45, 7000, 2], x(5), b(5), di(5);
    b = A*xx;
    if (mpirank == 0){
        cout << "b = " << b << endl;
        cout << "xx = " << xx << endl;
    }

    set(A, solver=sparsesolver, datafilename="ffmumps_fileparam.txt");

    // Solve
    if (mpirank == 0)
        cout << "Solve" << endl;
    x = A^-1*b;

    if (mpirank == 0){
        cout << "b = " << b << endl;
        cout << "x = " << x << endl;
    }
```

(continues on next page)

(continued from previous page)

```
    di = xx-x;
    if (mpirank == 0){
        cout << "x-xx = " << endl;
        cout << "Linf = " << di.linfty << ", L2 = " << di.l2 << endl;
    }
}
```

6.6.5 Solver superLU_DIST

Todo: write code (SuperLU_DIST seems to have a bug)

6.6.6 Solver PaStiX

Todo: write code (PaStiX seems to have a bug)

6.7 Developers

6.7.1 FFT

```
load "dfft"

// Parameters
int nx = 32;
real ny = 16;
real N = nx*ny;
func f1 = cos(2*x*2*pi)*cos(3*y*2*pi);

// Mesh
//warning: the fourier space is not exactly the unit square due to periodic condition
mesh Th = square(nx-1, ny-1, [(nx-1)*x/nx, (ny-1)*y/ny]);
//warning: the numbering of the vertices (x,y) is
//given by i = x/nx + nx*y/ny

// Fespace
fespace Vh(Th,P1);
Vh<complex> u = f1, v;
Vh w = f1;
Vh ur, ui;

// FFT
//in dfft the matrix n, m is in row-major order and array n, m is
//store j + m*i (the transpose of the square numbering)
v[] = dfft(u[], ny, -1);
u[] = dfft(v[], ny, +1);
cout << "||u||_\infty " << u[].linfty << endl;

```

(continues on next page)

(continued from previous page)

```
u[] *= 1./N;
cout << "||u||_\infty " << u[].linfty << endl;

ur = real(u);

// Plot
plot(w, wait=1, value=1, cmm="w");
plot(ur, wait=1, value=1, cmm="u");
v = w - u;
cout << "diff = " << v[].max << " " << v[].min << endl;
assert( norm(v[].max) < 1e-10 && norm(v[].min) < 1e-10);

// Other example
//FFT Lapacian
//-\Delta u = f with biperiodic condition
func f = cos(3*2*pi*x)*cos(2*2*pi*y);
func ue = (1./(square(2*pi)*13.))*cos(3*2*pi*x)*cos(2*2*pi*y); //the exact solution
Vh<complex> ff = f;
Vh<complex> fhat;
Vh<complex> wij;

// FFT
fhat[] = dfft(ff[],ny,-1);

//warning in fact we take mode between -nx/2, nx/2 and -ny/2, ny/2
//thanks to the operator ?:
wij = square(2.*pi)*(square(( x<0.5?x*nx:(x-1)*nx)) + square((y<0.5?y*ny:(y-1)*ny)));
wij[][0] = 1e-5; //to remove div / 0
fhat[] = fhat[] ./ wij[];
u[] = dfft(fhat[], ny, 1);
u[] /= complex(N);
ur = real(u); //the solution
w = real(ue); //the exact solution

// Plot
plot(w, ur, value=1, cmm="ue", wait=1);

// Error
w[] -= ur[];
real err = abs(w[].max) + abs(w[].min);
cout << "err = " << err << endl;
assert(err < 1e-6);

fftwplan p1 = plandfft(u[], v[], ny, -1);
fftwplan p2 = plandfft(u[], v[], ny, 1);
real ccc = square(2.*pi);
cout << "ny = " << ny << endl;
map(wij[], ny, ccc*(x*x+y*y));
wij[][0] = 1e-5;
plot(wij, cmm="wij");
```

6.7.2 Complex

```
real a = 2.45, b = 5.33;
complex z1 = a + b*1i, z2 = a + sqrt(2.)*1i;
```

(continues on next page)

(continued from previous page)

```

func string pc(complex z){
    string r = "(" + real(z);
    if (imag(z) >= 0) r = r + "+";
    return r + imag(z) + "i)";
}

func string toPolar(complex z){
    return "";//abs(z) + "*(cos(" + arg(z) + ")+i*sin(" + arg(z) + "))";
}

cout << "Standard output of the complex " << pc(z1) << " is the pair: " << z1 << endl;
cout << pc(z1) << " + " << pc(z2) << " = " << pc(z1+z2) << endl;
cout << pc(z1) << " - " << pc(z2) << " = " << pc(z1-z2) << endl;
cout << pc(z1) << " * " << pc(z2) << " = " << pc(z1*z2) << endl;
cout << pc(z1) << " / " << pc(z2) << " = " << pc(z1/z2) << endl;
cout << "Real part of " << pc(z1) << " = " << real(z1) << endl;
cout << "Imaginary part of " << pc(z1) << " = " << imag(z1) << endl;
cout << "abs(" << pc(z1) << ") = " << abs(z1) << endl;
cout << "Polar coordinates of " << pc(z2) << " = " << toPolar(z2) << endl;
cout << "de Moivre formula: " << pc(z2) << "^3 = " << toPolar(z2^3) << endl;
cout << " and polar(" << abs(z2) << ", " << arg(z2) << ") = " << pc(polar(abs(z2),
↪arg(z2))) << endl;
cout << "Conjugate of " <<pc(z2) << " = " << pc(conj(z2)) <<endl;
cout << pc(z1) << " ^ " << pc(z2) << " = " << pc(z1^z2) << endl;
```

Output of this script is:

```
Standard output of the complex (2.45+5.33i) is the pair: (2.45,5.33)
(2.45+5.33i) + (2.45+1.41421i) = (4.9+6.74421i)
(2.45+5.33i) - (2.45+1.41421i) = (0+3.91579i)
(2.45+5.33i) * (2.45+1.41421i) = (-1.53526+16.5233i)
(2.45+5.33i) / (2.45+1.41421i) = (1.692+1.19883i)
Real part of (2.45+5.33i) = 2.45
Imaginary part of (2.45+5.33i) = 5.33
abs((2.45+5.33i)) = 5.86612
Polar coordinates of (2.45+1.41421i) =
de Moivre formula: (2.45+1.41421i)^3 =
 and polar(2.82887, 0.523509) = (2.45+1.41421i)
Conjugate of (2.45+1.41421i) = (2.45-1.41421i)
(2.45+5.33i) ^ (2.45+1.41421i) = (8.37072-12.7078i)
```

6.7.3 String

```
// Concatenation
string tt = "toto1" + 1 + " -- 77";

// Append
string t1 = "0123456789";
t1(4:3) = "abcdefghijk-";

// Sub string
string t55 = t1(4:14);

cout << "tt = " << tt << endl;
```

(continues on next page)

(continued from previous page)

```

cout << "t1 = " << t1 << endl;
cout << "t1.find(abc) = " << t1.find("abc") << endl;
cout << "t1.rfind(abc) = " << t1.rfind("abc") << endl;
cout << "t1.find(abc, 10) = " << t1.find("abc",10) << endl;
cout << "t1.ffind(abc, 10) = " << t1.rfind("abc",10) << endl;
cout << "t1.length = " << t1.length << endl;

cout << "t55 = " << t55 << endl;
```

The output of this script is:

```
tt = toto11 -- 77
t1 = 0123abcdefghijk-456789
t1.find(abc) = 4
t1.rfind(abc) = 4
t1.find(abc, 10) = -1
t1.ffind(abc, 10) = 4
t1.length = 22
t55 = abcdefghijk
```

6.7.4 Elementary function

```
real b = 1.;
real a = b;
func real phix(real t){
    return (a+b)*cos(t) - b*cos(t*(a+b)/b);
}
func real phiy(real t){
    return (a+b)*sin(t) - b*sin(t*(a+b)/b);
}

border C(t=0, 2*pi){x=phix(t); y=phiy(t);}
mesh Th = buildmesh(C(50));
plot(Th);
```

Fig. 6.26: Mesh

6.7.5 Array

```
real[int] tab(10), tab1(10); //2 array of 10 real
//real[int] tab2; //bug: array with no size

tab = 1.03; //set all the array to 1.03
tab[1] = 2.15;

cout << "tab: " << tab << endl;
cout << "min: " << tab.min << endl;
cout << "max: " << tab.max << endl;
cout << "sum: " << tab.sum << endl;

tab.resize(12); //change the size of array tab to 12 with preserving first value
tab(10:11) = 3.14; //set values 10 & 11
cout << "resized tab: " << tab << endl;

tab.sort ; //sort the array tab
cout << "sorted tab:" << tab << endl;

real[string] tt; //array with string index
tt["+"] = 1.5;
cout << "tt[\"a\"] = " << tt["a"] << endl;
cout << "tt[\"+\"] = " << tt["+"] << endl;

real[int] a(5), b(5), c(5), d(5);
a = 1;
b = 2;
c = 3;
a[2] = 0;
d = ( a ? b : c ); //for i = 0, n-1 : d[i] = a[i] ? b[i] : c[i]
cout << " d = ( a ? b : c ) is " << d << endl;
d = ( a ? 1 : c ); //for i = 0, n-1: d[i] = a[i] ? 1 : c[i]
d = ( a ? b : 0 ); //for i = 0, n-1: d[i] = a[i] ? b[i] : 0
d = ( a ? 1 : 0 ); //for i = 0, n-1: d[i] = a[i] ? 0 : 1

int[int] ii(0:d.n-1); //set array ii to 0, 1, ..., d.n-1
d = -1:-5; //set d to -1, -2, ..., -5

sort(d, ii); //sort array d and ii in parallel
cout << "d: " << d << endl;
cout << "ii: " << ii << endl;


{
    int[int] A1(2:10); //2, 3, 4, 5, 6, 7, 8, 9, 10
    int[int] A2(2:3:10); //2, 5, 8
    cout << "A1(2:10): " << A1 << endl;
    cout << "A2(2:3:10): " << A1 << endl;
    A1 = 1:2:5;
    cout << "1:2:5 => " << A1 << endl;
}
{
    real[int] A1(2:10); //2, 3, 4, 5, 6, 7, 8, 9, 10
    real[int] A2(2:3:10); //2, 5, 8
    cout << "A1(2:10): " << A1 << endl;
    cout << "A2(2:3:10): " << A1 << endl;
```

(continues on next page)

(continued from previous page)

```
    A1 = 1.:0.5:3.999;
    cout << "1.:0.5:3.999 => " << A1 << endl;
}
{
    complex[int] A1(2.+0i:10.+0i); //2, 3, 4, 5, 6, 7, 8, 9, 10
    complex[int] A2(2.:3.:10.); //2, 5, 8
    cout << " A1(2.+0i:10.+0i): " << A1 << endl;
    cout << " A2(2.:3.:10.)= " << A2 << endl;
    cout << " A1.re real part array: " << A1.re << endl ;
    // he real part array of the complex array
    cout << " A1.im imag part array: " << A1.im << endl ;
    //the imaginary part array of the complex array
}

// Integer array operators
{
    int N = 5;
    real[int] a(N), b(N), c(N);
    a = 1;
    a(0:4:2) = 2;
    a(3:4) = 4;
    cout << "a: " << a << endl;
    b = a + a;
    cout <<"b = a + a: " << b << endl;
    b += a;
    cout <<"b += a: " << b << endl;
    b += 2*a;
    cout <<"b += 2*a: " << b << endl;
    b /= 2;
    cout <<" b /= 2: " << b << endl;
    b .*= a; // same as b = b .* a
    cout << "b .*= a: " << b << endl;
    b ./= a; //same as b = b ./ a
    cout << "b ./= a: " << b << endl;
    c = a + b;
    cout << "c = a + b: " << c << endl;
    c = 2*a + 4*b;
    cout << "c = 2*a + 4b: " << c << endl;
    c = a + 4*b;
    cout << "c = a + 4b: " << c << endl;
    c = -a + 4*b;
    cout << "c = -a + 4b: " << c << endl;
    c = -a - 4*b;
    cout << "c = -a - 4b: " << c << endl;
    c = -a - b;
    cout << "c = -a -b: " << c << endl;

    c = a .* b;
    cout << "c = a .* b: " << c << endl;
    c = a ./ b;
    cout << "c = a ./ b: " << c << endl;
    c = 2 * b;
    cout << "c = 2 * b: " << c << endl;
    c = b * 2;
    cout << "c = b * 2: " << c << endl;

    //this operator do not exist
```

(continues on next page)

(continued from previous page)

```
    //c = b/2;
    //cout << "c = b / 2: " << c << endl;

    //Array methods
    cout << "||a||_1 = " << a.l1 << endl;
    cout << "||a||_2 = " << a.l2 << endl;
    cout << "||a||_infty = " << a.linfty << endl;
    cout << "sum a_i = " << a.sum << endl;
    cout << "max a_i = " << a.max << " a[ " << a.imax << " ] = " << a[a.imax] << endl;
    cout << "min a_i = " << a.min << " a[ " << a.imin << " ] = " << a[a.imin] << endl;

    cout << "a' * a = " << (a'*a) << endl;
    cout << "a quantile 0.2 = " << a.quantile(0.2) << endl;

    //Array mapping
    int[int] I = [2, 3, 4, -1, 3];
    b = c = -3;
    b = a(I); //for (i = 0; i < b.n; i++) if (I[i] >= 0) b[i] = a[I[i]];
    c(I) = a; //for (i = 0; i < I.n; i++) if (I[i] >= 0) C(I[i]) = a[i];
    cout << "b = a(I) : " << b << endl;
    cout << "c(I) = a " << c << endl;
    c(I) += a; //for (i = 0; i < I.n; i++) if (I[i] >= 0) C(I[i]) += a[i];
    cout << "b = a(I) : " << b << endl;
    cout << "c(I) = a " << c << endl;

}

{
    // Array versus matrix
    int N = 3, M = 4;

    real[int, int] A(N, M);
    real[int] b(N), c(M);
    b = [1, 2, 3];
    c = [4, 5, 6, 7];

    complex[int, int] C(N, M);
    complex[int] cb = [1, 2, 3], cc = [10i, 20i, 30i, 40i];

    b = [1, 2, 3];

    int [int] I = [2, 0, 1];
    int [int] J = [2, 0, 1, 3];

    A = 1; //set all the matrix
    A(2, :) = 4; //the full line 2
    A(:, 1) = 5; //the full column 1
    A(0:N-1, 2) = 2; //set the column 2
    A(1, 0:2) = 3; //set the line 1 from 0 to 2

    cout << "A = " << A << endl;

    //outer product
    C = cb * cc';
    C += 3 * cb * cc';
    C -= 5i * cb * cc';
    cout << "C = " << C << endl;
```

(continues on next page)

(continued from previous page)

```

    //this transforms an array into a sparse matrix
    matrix B;
    B = A;
    B = A(I, J); //B(i, j) = A(I(i), J(j))
    B = A(I^-1, J^-1); //B(I(i), J(j)) = A(i,j)

    //outer product
    A = 2. * b * c';
    cout << "A = " << A << endl;
    B = b*c'; //outer product B(i, j) = b(i)*c(j)
    B = b*c'; //outer product B(i, j) = b(i)*c(j)
    B = (2*b*c')(I, J); //outer product B(i, j) = b(I(i))*c(J(j))
    B = (3.*b*c')(I^-1,J^-1); //outer product B(I(i), J(j)) = b(i)*c(j)
    cout << "B = (3.*b*c')(I^-1,J^-1) = " << B << endl;

    //row and column of the maximal coefficient of A
    int i, j, ii, jj;
    ijmax(A, ii, jj);

    i = A.imax;
    j = A.jmax;

    cout << "Max " << i << " " << j << ", = " << A.max << endl;

    //row and column of the minimal coefficient of A
    ijmin(A, i, j);

    ii = A.imin;
    jj = A.jmin;

    cout << "Min " << ii << " " << jj << ", = " << A.min << endl;
}
```

The output os this script is:

```
tab: 10
    1.03    2.15    1.03    1.03    1.03
    1.03    1.03    1.03    1.03    1.03

min: 1.03
max: 2.15
sum: 11.42
resized tab: 12
    1.03    2.15    1.03    1.03    1.03
    1.03    1.03    1.03    1.03    1.03
    3.14    3.14
sorted tab:12
    1.03    1.03    1.03    1.03    1.03
    1.03    1.03    1.03    1.03    2.15
    3.14    3.14
tt["a"] = 0
tt["+"] = 1.5
 d = ( a ? b : c ) is 5
      2   2   3   2   2

d: 5
```

(continues on next page)

(continued from previous page)

```
     -5  -4  -3  -2  -1

ii: 5
      4   3   2   1   0

A1(2:10): 9
      2   3   4   5   6
      7   8   9  10
A2(2:3:10): 9
      2   3   4   5   6
      7   8   9  10
1:2:5 => 3
      1   3   5
A1(2:10): 9
      2   3   4   5   6
      7   8   9  10
A2(2:3:10): 9
      2   3   4   5   6
      7   8   9  10
1.:0.5:3.999 => 6
      1 1.5   2 2.5   3
    3.5
 A1(2.+0i:10.+0i): 9
    (2,0)   (3,0)   (4,0)   (5,0)   (6,0)
    (7,0)   (8,0)   (9,0)   (10,0)
 A2(2.:3.:10.)= 3
    (2,0)   (5,0)   (8,0)
 A1.re real part array: 9
      2   3   4   5   6
      7   8   9  10
 A1.im imag part array: 9
      0   0   0   0   0
      0   0   0   0
a: 5
      2   1   2   4   4

b = a + a: 5
      4   2   4   8   8

b += a: 5
      6   3   6  12  12

b += 2*a: 5
     10   5  10  20  20

 b /= 2: 5
      5 2.5   5  10  10

b .*= a: 5
     10 2.5  10  40  40

b ./= a: 5
      5 2.5   5  10  10

c = a + b: 5
      7 3.5   7  14  14

```

(continues on next page)

(continued from previous page)

```
c = 2*a + 4b: 5
     24  12  24  48  48

c = a + 4b: 5
     22  11  22  44  44

c = -a + 4b: 5
     18   9  18  36  36

c = -a - 4b: 5
    -22 -11 -22 -44 -44

c = -a -b: 5
     -7 -3.5     -7 -14 -14

c = a .* b: 5
     10 2.5  10  40  40

c = a ./ b: 5
    0.4 0.4 0.4 0.4 0.4

c = 2 * b: 5
     10   5  10  20  20

c = b * 2: 5
     10   5  10  20  20

||a||_1 = 13
||a||_2 = 6.40312
||a||_infty = 4
sum a_i = 13
max a_i = 4 a[ 3 ] = 4
min a_i = 1 a[ 1 ] = 1
a' * a = 41
a quantile 0.2 = 2
b = a(I) : 5
      2   4   4  -3   4

c(I) = a 5
     -3  -3   2   4   2

b = a(I) : 5
      2   4   4  -3   4

c(I) = a 5
     -3  -3   4   9   4

A = 3 4
       1   5   2   1
       3   3   3   1
       4   5   2   4

C = 3 4
     (-50,-40) (-100,-80) (-150,-120) (-200,-160)
     (-100,-80) (-200,-160) (-300,-240) (-400,-320)
     (-150,-120) (-300,-240) (-450,-360) (-600,-480)

```

(continues on next page)

(continued from previous page)

```
A = 3 4
       8  10  12  14
      16  20  24  28
      24  30  36  42

B = (3.*b*c')(I^-1,J^-1) = # Sparse Matrix (Morse)
# first line: n m (is symmetic) nbcoef
# after for each nonzero coefficient:   i j a_ij where (i,j) \in  {1,...,n}x{1,...,m}
3 4 0  12
        1         1 10
        1         2 12
        1         3 8
        1         4 14
        2         1 15
        2         2 18
        2         3 12
        2         4 21
        3         1 5
        3         2 6
        3         3 4
        3         4 7
```

6.7.6 Block matrix

```
// Parameters
real f1 = 1.;
real f2 = 1.5;

// Mesh
mesh Th1 = square(10, 10);
mesh Th2 = square(10, 10, [1+x, -1+y]);
plot(Th1, Th2);

// Fespace
fespace Uh1(Th1, P1);
Uh1 u1;

fespace Uh2(Th2, P2);
Uh2 u2;

// Macro
macro grad(u) [dx(u), dy(u)] //

// Problem
varf vPoisson1 (u, v)
    = int2d(Th1)(
          grad(u)' * grad(v)
    )
    - int2d(Th1)(
          f1 * v
    )
    + on(1, 2, 3, 4, u=0)
    ;

varf vPoisson2 (u, v)
```

(continues on next page)

(continued from previous page)

```
    = int2d(Th2)(
          grad(u)' * grad(v)
    )
    - int2d(Th2)(
          f1 * v
    )
    + on(1, 2, 3, 4, u=0)
    ;
matrix<real> Poisson1 = vPoisson1(Uh1, Uh1);
real[int] Poisson1b = vPoisson1(0, Uh1);

matrix<real> Poisson2 = vPoisson2(Uh2, Uh2);
real[int] Poisson2b = vPoisson2(0, Uh2);

//block matrix
matrix<real> G = [[Poisson1, 0], [0, Poisson2]];
set(G, solver=sparsesolver);

//block right hand side
real[int] Gb = [Poisson1b, Poisson2b];

// Solve
real[int] sol = G^-1 * Gb;

// Dispatch
[u1[], u2[]] = sol;

// Plot
plot(u1, u2);
```

Fig. 6.27: Result

6.7.7 Matrix operations

```
// Mesh
mesh Th = square(2, 1);

// Fespace
fespace Vh(Th, P1);
Vh f, g;
f = x*y;
g = sin(pi*x);

Vh<complex> ff, gg; //a complex valued finite element function
ff= x*(y+1i);
gg = exp(pi*x*1i);

// Problem
varf mat (u, v)
    = int2d(Th)(
          1*dx(u)*dx(v)
        + 2*dx(u)*dy(v)
        + 3*dy(u)*dx(v)
        + 4*dy(u)*dy(v)
    )
    + on(1, 2, 3, 4, u=1)
    ;

varf mati (u, v)
    = int2d(Th)(
         1*dx(u)*dx(v)
        + 2i*dx(u)*dy(v)
        + 3*dy(u)*dx(v)
        + 4*dy(u)*dy(v)
    )
    + on(1, 2, 3, 4, u=1)
    ;

matrix A = mat(Vh, Vh);
matrix<complex> AA = mati(Vh, Vh); //a complex sparse matrix

// Operations
Vh m0; m0[] = A*f[];
Vh m01; m01[] = A'*f[];
Vh m1; m1[] = f[].*g[];
Vh m2; m2[] = f[]./g[];

// Display
cout << "f = " << f[] << endl;
cout << "g = " << g[] << endl;
cout << "A = " << A << endl;
cout << "m0 = " << m0[] << endl;
cout << "m01 = " << m01[] << endl;
cout << "m1 = "<< m1[] << endl;
cout << "m2 = "<< m2[] << endl;
cout << "dot Product = "<< f[]'*g[] << endl;
cout << "hermitien Product = "<< ff[]'*gg[] << endl;
cout << "outer Product = "<< (A=f[]*g[]') << endl;
cout << "hermitien outer Product = "<< (AA=ff[]*gg[]') << endl;
```

(continues on next page)

(continued from previous page)

```

// Diagonal
real[int] diagofA(A.n);
diagofA = A.diag; //get the diagonal of the matrix
A.diag = diagofA ; //set the diagonal of the matrix

// Sparse matrix set
int[int] I(1), J(1);
real[int] C(1);

[I, J, C] = A; //get the sparse term of the matrix A (the array are resized)
cout << "I = " << I << endl;
cout << "J = " << J << endl;
cout << "C = " << C << endl;

A = [I, J, C]; //set a new matrix
matrix D = [diagofA]; //set a diagonal matrix D from the array diagofA
cout << "D = " << D << endl;
```

The output of this script is:

```
f = 6
      0   0   0   0 0.5
      1
g = 6
      0   1 1.224646799e-16   0   1
    1.224646799e-16
A = # Sparse Matrix (Morse)
# first line: n m (is symmetic) nbcoef
# after for each nonzero coefficient:   i j a_ij where (i,j) \in  {1,...,n}x{1,...,m}
6 6 0  24
        1         1 1.0000000000000000199e+30
        1         2 0.49999999999999994449
        1         4 0
        1         5 -2.5
        2         1 0
        2         2 1.0000000000000000199e+30
        2         3 0.49999999999999994449
        2         5 0.49999999999999977796
        2         6 -2.5
        3         2 0
        3         3 1.0000000000000000199e+30
        3         6 0.49999999999999977796
        4         1 0.49999999999999977796
        4         4 1.0000000000000000199e+30
        4         5 0
        5         1 -2.5
        5         2 0.49999999999999977796
        5         4 0.49999999999999994449
        5         5 1.0000000000000000199e+30
        5         6 0
        6         2 -2.5
        6         3 0
        6         5 0.49999999999999994449
        6         6 1.0000000000000000199e+30

m0 = 6
```

(continues on next page)

(continued from previous page)

```
    -1.25   -2.25   0.5   0 5e+29
    1e+30
m01 = 6
    -1.25   -2.25     0 0.25    5e+29
    1e+30
m1 = 6
      0   0   0   0 0.5
    1.224646799e-16
m2 = 6
    -nan      0   0 -nan    0.5
    8.165619677e+15
dot Product = 0.5
hermitien Product = (1.11022e-16,2.5)
outer Product = # Sparse Matrix (Morse)
# first line: n m (is symmetic) nbcoef
# after for each nonzero coefficient:   i j a_ij where (i,j) \in  {1,...,n}x{1,...,m}
6 6 0  8
        5         2 0.5
        5         3 6.1232339957367660359e-17
        5         5 0.5
        5         6 6.1232339957367660359e-17
        6         2 1
        6         3 1.2246467991473532072e-16
        6         5 1
        6         6 1.2246467991473532072e-16

hermitien outer Product = # Sparse Matrix (Morse)
# first line: n m (is symmetic) nbcoef
# after for each nonzero coefficient:   i j a_ij where (i,j) \in  {1,...,n}x{1,...,m}
6 6 0  24
        2         1 (0,0.5)
        2         2 (0.5,3.0616169978683830179e-17)
        2         3 (6.1232339957367660359e-17,-0.5)
        2         4 (0,0.5)
        2         5 (0.5,3.0616169978683830179e-17)
        2         6 (6.1232339957367660359e-17,-0.5)
        3         1 (0,1)
        3         2 (1,6.1232339957367660359e-17)
        3         3 (1.2246467991473532072e-16,-1)
        3         4 (0,1)
        3         5 (1,6.1232339957367660359e-17)
        3         6 (1.2246467991473532072e-16,-1)
        5         1 (0.5,0.5)
        5         2 (0.5,-0.49999999999999994449)
        5         3 (-0.49999999999999994449,-0.50000000000000011102)
        5         4 (0.5,0.5)
        5         5 (0.5,-0.49999999999999994449)
        5         6 (-0.49999999999999994449,-0.50000000000000011102)
        6         1 (1,1)
        6         2 (1,-0.99999999999999988898)
        6         3 (-0.99999999999999988898,-1.000000000000000222)
        6         4 (1,1)
        6         5 (1,-0.99999999999999988898)
        6         6 (-0.99999999999999988898,-1.000000000000000222)

I = 8
      4   4   4   4   5
```

(continues on next page)

(continued from previous page)

```
      5   5   5
J = 8
      1   2   4   5   1
      2   4   5
C = 8
    0.5 6.123233996e-17 0.5 6.123233996e-17   1
    1.224646799e-16   1 1.224646799e-16
  -- Raw Matrix    nxm  =6x6 nb  none zero coef. 8
  -- Raw Matrix    nxm  =6x6 nb  none zero coef. 6
D = # Sparse Matrix (Morse)
# first line: n m (is symmetic) nbcoef
# after for each nonzero coefficient:   i j a_ij where (i,j) \in  {1,...,n}x{1,...,m}
6 6 1  6
        1         1 0
        2         2 0
        3         3 0
        4         4 0
        5         5 0.5
        6         6 1.2246467991473532072e-16
```

Warning: Due to `Fortran` indices starting at one, the output of a diagonal matrix `D` is indexed from 1. but in **FreeFEM**, the indices start from 0.

6.7.8 Matrix inversion

```
load "lapack"
load "fflapack"

// Matrix
int n = 5;
real[int, int] A(n, n), A1(n, n), B(n,n);
for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j)
        A(i, j) = (i == j) ? n+1 : 1;
cout << A << endl;

// Inversion (lapack)
A1 = A^-1; //def in "lapack"
cout << A1 << endl;

B = 0;
for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j)
        for (int k = 0; k < n; ++k)
            B(i, j) += A(i,k)*A1(k,j);
cout << B << endl;

// Inversion (fflapack)
inv(A1); //def in "fflapack"
cout << A1 << endl;
```

The output of this script is:

```
5 5
       6   1   1   1   1
       1   6   1   1   1
       1   1   6   1   1
       1   1   1   6   1
       1   1   1   1   6

5 5
     0.18 -0.02 -0.02 -0.02 -0.02
     -0.02 0.18 -0.02 -0.02 -0.02
     -0.02 -0.02 0.18 -0.02 -0.02
     -0.02 -0.02 -0.02 0.18 -0.02
     -0.02 -0.02 -0.02 -0.02 0.18

5 5
       1 1.040834086e-17 1.040834086e-17 1.734723476e-17 2.775557562e-17
     3.469446952e-18   1 -1.734723476e-17 1.734723476e-17 2.775557562e-17
     2.428612866e-17 -3.122502257e-17   1 1.734723476e-17 2.775557562e-17
     2.081668171e-17 -6.938893904e-17 -3.469446952e-17   1   0
     2.775557562e-17 -4.163336342e-17 -2.775557562e-17   0   1

5 5
       6   1   1   1   1
       1   6   1   1   1
       1   1   6   1   1
       1   1   1   6   1
       1   1   1   1   6
```

Tip: To compile `lapack.cpp` and `fflapack.cpp`, you must have the `lapack` library on your system and compile the plugin with the command:

```
ff-c++ lapack.cpp -llapack     ff-c++ fflapack.cpp -llapack
```

6.7.9 FE array

```
// Mesh
mesh Th = square(20, 20, [2*x, 2*y]);

// Fespace
fespace Vh(Th, P1);
Vh u, v, f;

// Problem
problem Poisson (u, v)
    = int2d(Th)(
          dx(u)*dx(v)
        + dy(u)*dy(v)
    )
    + int2d(Th)(
        - f*v
    )
    + on(1, 2, 3, 4, u=0)
    ;
```

(continues on next page)

(a) First result

(b) Second result

(c) Third result

Fig. 6.28: Finite element array

(continued from previous page)

```
Vh[int] uu(3); //an array of FE function
// Solve problem 1
f = 1;
Poisson;
uu[0] = u;
// Solve problem 2
f = sin(pi*x)*cos(pi*y);
Poisson;
uu[1] = u;
// Solve problem 3
f = abs(x-1)*abs(y-1);
Poisson;
uu[2] = u;

// Plot
for (int i = 0; i < 3; i++)
    plot(uu[i], wait=true);
```

6.7.10 Loop

```
for (int i = 0; i < 10; i=i+1)
    cout << i << endl;

real eps = 1.;
while (eps > 1e-5){
    eps = eps/2;
    if (i++ < 100)
        break;
    cout << eps << endl;
}

for (int j = 0; j < 20; j++){
    if (j < 10) continue;
    cout << "j = " << j << endl;
}
```

6.7.11 Implicit loop

```
real [int, int] a(10, 10);
real [int] b(10);

for [i, bi : b]{
    bi = i+1;
    cout << i << " " << bi << endl;
}
cout << "b = " << b << endl;

for [i, j, aij : a]{
    aij = 1./(2+i+j);
    if (abs(aij) < 0.2) aij = 0;
}
cout << "a = " << a << endl;

matrix A = a;
string[string] ss; //a map
ss["1"] = 1;
ss["2"] = 2;
ss["3"] = 5;
for [i, bi : ss]
    bi = i + 6 + "-dddd";
cout << "ss = " << ss << endl;

int[string] si;
si[1] = 2;
si[50] = 1;
for [i, vi : si]{
    cout << " i " << setw(3) << i << " " << setw(10) << vi << endl;
    vi = atoi(i)*2;
}
cout << "si = " << si << endl;

for [i, j, aij : A]{
    cout << i << " " << j << " " << aij << endl;
```

(continues on next page)

(continued from previous page)

```
    aij = -aij;
}
cout << A << endl;
```

The output of this script is:

```
0 1
1 2
2 3
3 4
4 5
5 6
6 7
7 8
8 9
9 10
b = 10
      1   2   3   4   5
      6   7   8   9  10

a = 10 10
     0.5 0.3333333333 0.25 0.2   0   0   0   0   0   0
     0.3333333333 0.25 0.2   0   0   0   0   0   0   0
     0.25 0.2   0   0   0   0   0   0   0   0
     0.2   0   0   0   0   0   0   0   0   0
       0   0   0   0   0   0   0   0   0   0
       0   0   0   0   0   0   0   0   0   0
       0   0   0   0   0   0   0   0   0   0
       0   0   0   0   0   0   0   0   0   0
       0   0   0   0   0   0   0   0   0   0
       0   0   0   0   0   0   0   0   0   0

ss = 1 1
2 2
3 5

 i   1          2
 i  50          1
si = 1 2
50 100

0 0 0.5
0 1 0.333333
0 2 0.25
0 3 0.2
1 0 0.333333
1 1 0.25
1 2 0.2
2 0 0.25
2 1 0.2
3 0 0.2
# Sparse Matrix (Morse)
# first line: n m (is symmetic) nbcoef
# after for each nonzero coefficient:   i j a_ij where (i,j) \in  {1,...,n}x{1,...,m}
10 10 0  10
        1         1 -0.5
        1         2 -0.33333333333333331483
```

(continues on next page)

(continued from previous page)

```
        1          3 -0.25
        1          4 -0.2000000000000000111
        2          1 -0.33333333333333331483
        2          2 -0.25
        2          3 -0.2000000000000000111
        3          1 -0.25
        3          2 -0.2000000000000000111
        4          1 -0.2000000000000000111
```

6.7.12 I/O

```
int i;
cout << "std-out" << endl;
cout << " enter i = ?";
cin >> i;

{
    ofstream f("toto.txt");
    f << i << "hello world'\n";
} //close the file f because the variable f is delete

{
    ifstream f("toto.txt");
    f >> i;
}

{
    ofstream f("toto.txt", append);
    //to append to the existing file "toto.txt"
    f << i << "hello world'\n";
} //close the file f because the variable f is delete

cout << i << endl;
```

6.7.13 File stream

```
int where;
real[int] f = [0, 1, 2, 3, 4, 5];
real[int] g(6);

{
    ofstream file("f.txt", binary);
    file.precision(16);
    file << f << endl;
    where = file.tellp();
    file << 0.1 ;

    cout << "Where in file " << where << endl;
    file << " # comment bla bla ... 0.3 \n";
    file << 0.2 << endl;
    file.flush; //to flush the buffer of file
}
```

(continues on next page)

(continued from previous page)

```

//Function to skip comment starting with # in a file
func ifstream skipcomment(ifstream &ff){
    while(1){
        int where = ff.tellg(); //store file position
        string comment;
        ff >> comment;
        if (!ff.good()) break;
        if (comment(0:0)=="#"){
            getline(ff, comment);
            cout << " -- #" << comment << endl;
        }
        else{
            ff.seekg(where); //restore file position
            break;
        }
    }
    return ff;
}

{
    real xx;
    ifstream file("f.txt", binary);
    cout << "Where " << file.seekg << endl;
    file.seekg(where);
    file >> xx;
    cout << " xx = " << xx << " good ? " << file.good() << endl;
    assert(xx == 0.1);
    skipcomment(file) >> xx;
    assert(xx == 0.2);
    file.seekg(0); //rewind
    cout << "Where " << file.tellg() << " " << file.good() << endl;
    file >> g;
}
```

6.7.14 Command line arguments

When using the command:

```
FreeFem++ script.edp arg1 arg2
```

The arguments can be used in the script with:

```
for (int i = 0; i < ARGV.n; i++)
    cout << ARGV[i] << endl;
```

When using the command:

```
FreeFem++ script.edp -n 10 -a 1. -d 42.
```

The arguments can be used in the script with:

```
include "getARGV.idp"

int n = getARGV("-n", 1);
```

(continues on next page)

(continued from previous page)

```
real a = getARGV("-a", 1.);
real d = getARGV("-d", 1.);
```

6.7.15 Macro

```
// Macro without parameters
macro xxx() {
    real i = 0;
    int j = 0;
    cout << i << " " << j << endl;
}//

xxx

// Macro with parameters
macro toto(i) i //

toto({real i = 0; int j = 0; cout << i << " " << j << endl;})

// Macro as parameter of a macro
real[int,int] CC(7, 7), EE(6, 3), EEps(4, 4);

macro VIL6(v, i) [v(1,i), v(2,i), v(4,i), v(5,i), v(6,i)] //
macro VIL3(v, i) [v(1,i), v(2,i)] //
macro VV6(v, vv) [
    v(vv,1), v(vv,2),
    v(vv,4), v(vv,5),
    v(vv,6)] //
macro VV3(v, vv) [v(vv,1), v(vv,2)] //

func C5x5 = VV6(VIL6, CC);
func E5x2 = VV6(VIL3, EE);
func Eps = VV3(VIL3, EEps);

// Macro concatenation
mesh Th = square(2, 2);
fespace Vh(Th, P1);
Vh Ux=x, Uy=y;

macro div(V) (dx(V#x) + dy(V#y)) //

cout << int2d(Th)(div(U)) << endl;

// Verify the quoting
macro foo(i, j, k) i j k //
foo(, , )
foo({int[}, {int] a(10}, {);})

//NewMacro - EndMacro
NewMacro grad(u) [dx(u), dy(u)] EndMacro
cout << int2d(Th)(grad(Ux)' * grad(Uy)) << endl;

// IFMACRO - ENDIFMACRO
macro AA CAS1 //

```

(continues on next page)

(continued from previous page)

```
IFMACRO(AA,CAS1 )
cout << "AA = " << Stringification(AA) << endl;
macro CASE file1.edp//
ENDIFMACRO
IFMACRO(AA, CAS2)
macro CASE file2.edp//
ENDIFMACRO

cout << "CASE = " << Stringification(CASE) << endl;

IFMACRO(CASE)
include Stringification(CASE)
ENDIFMACRO

// FILE - LINE
cout << "In " << FILE << ", line " << LINE << endl;
```

The output script generated with macros is:

```
1 : // Macro without parameters
2 :  macro xxx {
3 :     real i = 0;
4 :     int j = 0;
5 :     cout << i << " " << j << endl;
6 : }//
7 :
8 :
1 :
2 :
3 :
4 :  {
1 :     real i = 0;
2 :     int j = 0;
3 :     cout << i << " " << j << endl;
4 : }
9 :
10 : // Macro with parameters
11 :  macro toto(i )   i //
12 :
13 :                    real i = 0; int j = 0; cout << i << " " << j << endl;
14 :
15 : // Macro as parameter of a macro
16 : real[int,int] CC(7, 7), EE(6, 3), EEps(4, 4);
17 :
18 :   macro VIL6(v,i )   [v(1,i), v(2,i), v(4,i), v(5,i), v(6,i)] //
19 :   macro VIL3(v,i )   [v(1,i), v(2,i)] //
20 :   macro VV6(v,vv )   [
21 :    v(vv,1), v(vv,2),
22 :    v(vv,4), v(vv,5),
23 :    v(vv,6)] //
24 :   macro VV3(v,vv )   [v(vv,1), v(vv,2)] //
25 :
26 : func C5x5 =
1 :
2 :
3 :       [
1 :              [ CC(1,1),  CC(2,1),  CC(4,1),  CC(5,1),  CC(6,1)] ,          [ CC(1,
↪2),  CC(2,2),  CC(4,2),  CC(5,2),  CC(6,2)] ,
```

(continues on next page)

(continued from previous page)

```
2 :             [ CC(1,4),  CC(2,4),  CC(4,4),  CC(5,4),  CC(6,4)] ,          [ CC(1,
↪5),  CC(2,5),  CC(4,5),  CC(5,5),  CC(6,5)] ,
3 :             [ CC(1,6),  CC(2,6),  CC(4,6),  CC(5,6),  CC(6,6)] ] ;
27 : func E5x2 =
1 :
2 :
3 :       [
1 :          [ EE(1,1),  EE(2,1)] ,      [ EE(1,2),  EE(2,2)] ,
2 :          [ EE(1,4),  EE(2,4)] ,      [ EE(1,5),  EE(2,5)] ,
3 :          [ EE(1,6),  EE(2,6)] ] ;
28 : func Eps =      [     [ EEps(1,1),  EEps(2,1)] ,      [ EEps(1,2),  EEps(2,2)] ]␣
↪;
29 :
30 : // Macro concatenation
31 : mesh Th = square(2, 2);
32 : fespace Vh(Th, P1);
33 : Vh Ux=x, Uy=y;
34 :
35 :  macro div(V )    (dx(V#x) + dy(V#y)) //
36 :
37 : cout << int2d(Th)(     (dx(Ux) + dy(Uy)) ) << endl;
38 :
39 : // Verify the quoting
40 :    macro foo(i,j,k )   i j k //
41 :
42 :         int[ int] a(10 );
43 :
44 : //NewMacro - EndMacro
45 :  macro grad(u )   [dx(u), dy(u)]
46 : cout << int2d(Th)(    [dx(Ux), dy(Ux)] ' *     [dx(Uy), dy(Uy)] ) << endl;
47 :
48 : // IFMACRO - ENDIFMACRO
49 :   macro AACAS1 //
50 :
51 :
1 : cout << "AA = " << Stringification( CAS1 ) << endl;
2 :   macro CASEfile1.edp//
3 :
52 :
53 :
54 : cout << "CASE = " << Stringification(file1.edp) << endl;
55 :
56 :
1 : include Stringification(file1.edp)cout << "This is the file 1" << endl;
2 :
2 :
57 :
58 : // FILE - LINE
59 : cout << "In " << FILE << ", line " << LINE << endl;
```

The output os this script is:

```
AA = CAS1
CASE = file1.edp
This is the file 1
In Macro.edp, line 59
```

6.7.16 Basic error handling

```
real a;
try{
    a = 1./0.;
}
catch (...) //all exceptions can be caught
{
    cout << "Catch an ExecError" << endl;
    a = 0.;
}
```

The output of this script is:

```
1/0 : d d d
  current line = 3
Exec error :  Div by 0
   -- number :1
Catch an ExecError
```

6.7.17 Error handling

```
// Parameters
int nn = 5;
func f = 1; //right hand side function
func g = 0; //boundary condition function

// Mesh
mesh Th = square(nn, nn);

// Fespace
fespace Vh(Th, P1);
Vh uh, vh;

// Problem
real cpu = clock();
problem laplace (uh, vh, solver=Cholesky, tolpivot=1e-6)
    = int2d(Th)(
          dx(uh)*dx(vh)
        + dy(uh)*dy(vh)
    )
    + int2d(Th)(
        - f*vh
    )
    ;

try{
    cout << "Try Cholesky" << endl;

    // Solve
    laplace;

    // Plot
    plot(uh);

```

(continues on next page)

(continued from previous page)

```
    // Display
    cout << "laplacian Cholesky " << nn << ", x_" << nn << " : " << -cpu+clock() << "␣
↪s, max = " << uh[].max << endl;
}
catch(...) { //catch all error
    cout << " Catch cholesky PB " << endl;
}
```

The output of this script is:

```
Try Cholesky
ERREUR choleskypivot (35)= -6.43929e-15 < 1e-06
  current line = 29
Exec error : FATAL ERREUR dans ./../femlib/MatriceCreuse_tpl.hpp
cholesky line:
   -- number :688
 catch an erreur in  solve  =>  set  sol = 0 !!!!!!!
 Catch cholesky PB
```

BIBLIOGRAPHY

[PIRONNEAU1998] PIRONNEAU, Olivier and LUCQUIN-DESREUX, Brigitte. Introduction to scientific computing. Wiley, 1998.

[WÄCHTER2006] WÄCHTER, Andreas and BIEGLER, Lorenz T. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. Mathematical programming, 2006, vol. 106, no 1, p. 25-57.

[FORSGREN2002] FORSGREN, Anders, GILL, Philip E., and WRIGHT, Margaret H. Interior methods for nonlinear optimization. SIAM review, 2002, vol. 44, no 4, p. 525-597.

[GEORGE1996] GEORGE, P. L. and BOROUCHAKI, H. Automatic triangulation. 1996.

[HECHT1998] HECHT, F. The mesh adapting software: bamg. INRIA report, 1998, vol. 250, p. 252.

[PREPARATA1985] PREPARATA, F. P. and SHAMOS, M. I. Computational Geometry Springer-Verlag. New York, 1985.

[STROUSTRUP2000] STROUSTRUP, Bjarne. The C++ programming language. Pearson Education India, 2000.

[HECHT2002] HECHT, Frédéric. C++ Tools to construct our user-level language. ESAIM: Mathematical Modelling and Numerical Analysis, 2002, vol. 36, no 5, p. 809-836.

[HANG2006] SI, Hang. TetGen Users' guide: A quality tetrahedral mesh generator and three-dimensional delaunay triangulator. 2006

[SHEWCHUK1998] SHEWCHUK, Jonathan Richard. Tetrahedral mesh generation by Delaunay refinement. In : Proceedings of the fourteenth annual symposium on Computational geometry. ACM, 1998. p. 86-95.

[HECHT1992] HECHT, F. Outils et algorithmes pour la méthode des éléments finis. HdR, Université Pierre et Marie Curie, France, 1992.

[HECHT1998_2] HECHT, Frédéric. BAMG: bidimensional anisotropic mesh generator. User Guide. INRIA, Rocquencourt, 1998.

[KARYPIS1995] KARYPIS, George and KUMAR, Vipin. METIS–unstructured graph partitioning and sparse matrix ordering system, version 2.0. 1995.

[CAI1989] CAI, Xiao-Chuan. Some domain decomposition algorithms for nonselfadjoint elliptic and parabolic partial differential equations. 1989.

[SAAD2003] SAAD, Yousef. Iterative methods for sparse linear systems. siam, 2003.

[SMITH1996] SMITH, B. P. Bj rstad and W. Gropp, Domain Decomposition. 1996.

[OGDEN1984] OGDEN, Ray W. Non-linear elastic deformations. 1984.

[RAVIART1998] RAVIART, Pierre-Arnaud, THOMAS, Jean-Marie, CIARLET, Philippe G., et al. Introduction à l'analyse numérique des équations aux dérivées partielles. Paris : Dunod, 1998.

[HORGAN2004] HORGAN, Cornelius O. and SACCOMANDI, Giuseppe. Constitutive models for compressible nonlinearly elastic materials with limiting chain extensibility. Journal of Elasticity, 2004, vol. 77, no 2, p. 123-138.

[LEHOUCQ1998] LEHOUCQ, Richard B., SORENSEN, Danny C., and YANG, Chao. ARPACK users' guide: solution of large-scale eigenvalue problems with implicitly restarted Arnoldi methods. Siam, 1998.

[NECAS2017] NECAS, Jindrich and HLAVÁCEK, Ivan. Mathematical theory of elastic and elasto-plastic bodies: an introduction. Elsevier, 2017.

[OHTSUKA2000] OHTSUKA, K. Theoretical and Numerical analysis of energy release rate in 2D fracture. INFORMATION, 2000, vol. 3, p. 303-315.

[TABATA1994] TABATA, M. Numerical solutions of partial differential equations II. Iwanami Applied Math, 1994.

[LUCQUIN1998] PIRONNEAU, O. and LUCQUIN-DESREUX, B. Introduction to scientific computing. Wiley, 1998.

[WILMOTT1995] WILMOTT, Paul, HOWISON, Sam and DEWYNNE, Jeff. A student introduction to mathematical finance. 1995.

[ACHDOU2005] ACHDOU, Yves and PIRONNEAU, Olivier. Computational methods for option pricing. Siam, 2005.

[TEMAM1977] TEMAM, Roger. Navier-Stokes equations: theory and numerical analysis. 1977.

[ROBERTS1993] ROBERTS, J. E. and THOMAS, J. M. Mixed and Hybrid Methods, Handbook of Numerical Anaysis, Vol. II. North-Holland, 1993, vol. 183, p. 184.

[GLOWINSKI1979] GLOWINSKI, R. and PIRONNEAU, O. On numerical methods for the Stokes problem. In: Energy methods in finite element analysis.(A79-53076 24-39) Chichester, Sussex, England, Wiley-Interscience, 1979, p. 243-264., 1979, p. 243-264.

[GLOWINSKI1985] GLOWINSKI, Roland and ODEN, J. Tinsley. Numerical methods for nonlinear variational problems. Journal of Applied Mechanics, 1985, vol. 52, p. 739.

[GLOWINSKI2003] GLOWINSKI, Roland. Finite element methods for incompressible viscous flow. Handbook of numerical analysis, 2003, vol. 9, p. 3-1176.

[ITO2003] ITO, Kazufumi and KUNISCH, Karl. Semi–smooth Newton methods for variational inequalities of the first kind. ESAIM: Mathematical Modelling and Numerical Analysis, 2003, vol. 37, no 1, p. 41-62.

[OXBORROW2007] OXBORROW, Mark. Traceable 2-D finite-element simulation of the whispering-gallery modes of axisymmetric electromagnetic resonators. IEEE Transactions on Microwave Theory and Techniques, 2007, vol. 55, no 6, p. 1209-1218.

[GRUDININ2012] GRUDININ, Ivan S. and YU, Nan. Finite-element modeling of coupled optical microdisk resonators for displacement sensing. JOSA B, 2012, vol. 29, no 11, p. 3010-3014.

[ERN2006] ERN, A. and GUERMOND, J. L. Discontinuous Galerkin methods for Friedrichs' symmetric systems. I. General theory. SIAM J. Numer. Anal.

[BERNADOU1980] BERNADOU, Michel, BOISSERIE, Jean-Marie and HASSAN, Kamal. Sur l'implémentation des éléments finis de Hsieh-Clough-Tocher complet et réduit. 1980. Thèse de doctorat. INRIA.

[BERNARDI1985] BERNARDI, Christine and RAUGEL, Genevieve. Analysis of some finite elements for the Stokes problem. Mathematics of Computation, 1985, p. 71-79.

[THOMASSET2012] THOMASSET, François. Implementation of finite element methods for Navier-Stokes equations. Springer Science & Business Media, 2012.

[CROUZEIX1984] CROUZEIX, Michel and MIGNOT, Alain L. Analyse numérique des équations différentielles. Masson, 1984.

[TAYLOR2005] TAYLOR, Mark A., WINGATE, Beth A. and BOS, Len P. Several new quadrature formulas for polynomial integration in the triangle. arXiv preprint math/0501496, 2005.

[CHOW1997] CHOW, Edmond and SAAD, Yousef. Parallel Approximate Inverse Preconditioners. In : PPSC. 1997.