Efficient C++ finite element computing with Rheolef

Pierre Saramito

▶ **To cite this version:**

Pierre Saramito. Efficient C++ finite element computing with Rheolef. DEA. Grenoble, France, 2012, pp.161. cel-00573970v9

HAL Id: cel-00573970

https://cel.hal.science/cel-00573970v9

Submitted on 17 Sep 2013 (v9), last revised 2 Jun 2022 (v16)

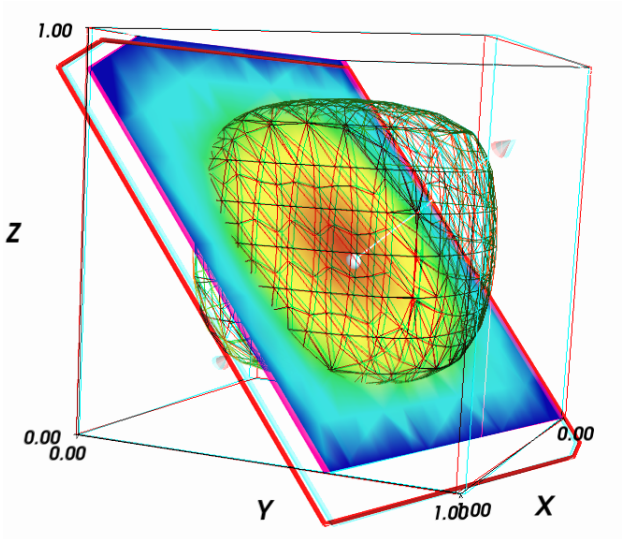
HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

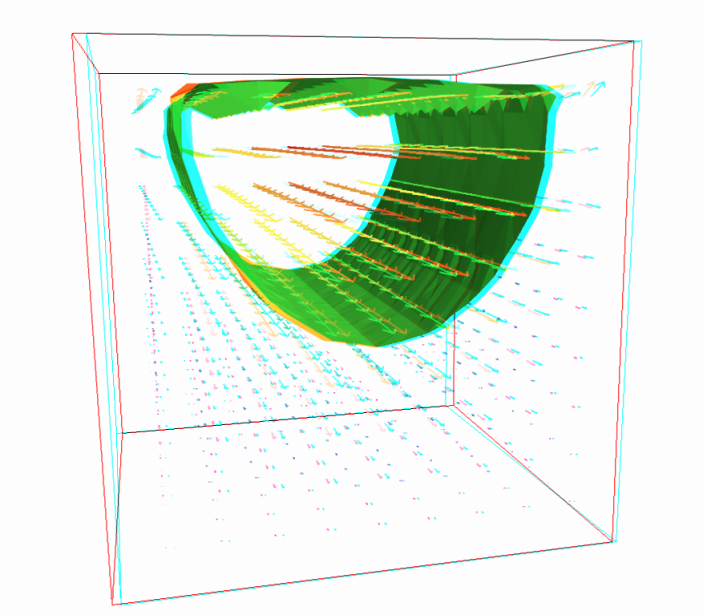
L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Efficient C++ finite element computing with Rheolef

Pierre Saramito

version 6.5 update 17 September 2013





Copyright (c) 2003-2013 Pierre Saramito

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

Introduction

Rheolef is a programming environment for finite element method computing.

This book presents in details how some simple and more complex problems from solid and fluid mechanics can be solved, most of them in less than 20 lines of code. The concision and readability of codes written with **Rheolef** is certainly a major keypoint of this environment (see Fig. 1).

Example: find u such that $-\Delta u = 1$ in Ω and $u = 0$ on $\partial\Omega$

```
int main (int argc, char** argv) {
  environment rheolef (argc, argv);
  geo omega (argv[1]);
  space Xh (omega, argv[2]);
  Xh.block ("boundary");
  trial u (Xh); test v (Xh);
  form a = integrate (dot(grad(u),grad(v)));
  field lh = integrate (v);
  field uh (Xh);
  uh ["boundary"] = 0;
  solver sa (a.uu());
  uh.u = sa.solve (lh.u());
  dout << uh;
}
```

Let $\Omega \subset \mathbb{R}^N, N = 1, 2, 3$

$X_h = \{v \in H^1(\Omega);\ v_{|K} \in P_k,\ \forall K \in \mathcal{T}_h\}$

$V_h = X_h \cap H^1_0(\Omega)$

$a(u,v) = \int_\Omega \nabla u.\nabla v\, dx$

$l(v) = \int_\Omega v\, dx$

(P) : find $u_h \in V_h$ such that

$a(u_h, v_h) = l(v_h), \quad \forall v_h \in V_h$

Figure 1: Example of a **Rheolef** code for solving the Poisson problem with homogeneous boundary conditions. The right column shows the one-to-one line correspondence between the code and the variational formulation of the problem.

Let us quote B. Stroustrup [49], the conceptor of the c++ language:

> *"The time taken to write a program is at best roughly proportional to the* **number of lines** *written, and so is the number of errors in that code. If follows that a good way of writing correct programs is to write* **short programs**. *In other words, we need good libraries to allow us to write correct code that performs well. This in turn means that we need libraries to get our programs finished in a reasonable time. In many fields, such* c++ *libraries exist."*

Rheolef is an attempt to provide such a library in the field of finite element methods for partial differential equations. As a Lego game, the **Rheolef** bricks allow the user to solve most complex nonlinear problems. **Rheolef** provides both a c++ library and a set of unix commands for **shell** programming, providing data structures and algorithms [52].

- Data structures fit the **variational formulation** concept: fields, bilinear forms and functional spaces, are c++ types for variables. They can be combined in expressions, as you write it on the paper.
- Algorithms refer to the most up-to-date ones: preconditioned sparse matrix solvers for linear systems, distributed memory and parallel computations, high order polynomial approximations, incompressible elasticity, Stokes and Navier-Stokes flows, characteristic method for convection dominated heat problems, etc. Also linear and nonlinear generic algorithms such as fixed point and damped Newton methods.

An efficient usage of **Rheolef** supposes a raisonable knowledge of the `c++` programming language (see e.g. [43, 48]) and also of the classical finite element method and its variational principles.

Contacts

email Pierre.Saramito@imag.fr

home page http://www-ljk.imag.fr/membres/Pierre.Saramito/rheolef

Please send all comments and bug reports by electronic mail to

rheolef@grenet.fr

The Rheolef present contributors

from 2008 **Ibrahim Cheddadi**: discontinuous Galerkin method for transport problems.
from 2010 **Mahamar Dicko**: finite element methods for equations on surfaces.
from 2002 **Jocelyn Étienne**: characteristic method for time-dependent problems.
from 2000 **Pierre Saramito**: project leader: main developments and code maintainer.

Past contributors

2010 **Lara Abouorm**: banded level set method for equations on surfaces.
2000 **Nicolas Roquet**: initial versions of Stokes and Bingham flow solvers.

Contents

Notations 8

I Getting started with simple problems 11

1 Getting started with Rheolef 15
1.1 The model problem . . . 15
1.2 Approximation . . . 16
1.3 Comments . . . 16
1.4 How to compile the code . . . 18
1.5 How to run the program . . . 19
1.6 Distributed and parallel runs . . . 19
1.7 Stereo visualization . . . 20
1.8 High-order finite element methods . . . 21
1.9 Tridimensional computations . . . 21
1.10 Quadrangles, prisms and hexahedra . . . 22

2 Standard boundary conditions 23
2.1 Non-homogeneous Dirichlet conditions . . . 23
2.2 Non-homogeneous Neumann boundary conditions for the Helmholtz operator . . . 31
2.3 The Robin boundary conditions . . . 33
2.4 Neumann boundary conditions for the Laplace operator . . . 35

3 Non-constant coefficients and multi-regions 39

II Fluids and solids computations 45

4 The linear elasticity and the Stokes problems 47
4.1 The linear elasticity problem . . . 47
4.2 Computing the stress tensor . . . 51
4.3 Mesh adaptation . . . 54
4.4 The Stokes problem . . . 57
4.5 Computing the vorticity . . . 60
4.6 Computing the stream function . . . 62

5 Nearly incompressible elasticity and the stabilized Stokes problems 65

5.1 The incompressible elasticity problem . . . 65
5.2 The $P_1b - P_1$ element for the Stokes problem . . . 67
5.3 Axisymmetric geometries . . . 73
5.4 The axisymmetric stream function and stress tensor . . . 73

6 Time-dependent problems **77**
6.1 The heat equation . . . 77
6.2 The convection-diffusion problem . . . 80
6.3 The Navier-Stokes problem . . . 86

III Advanced and highly nonlinear problems **95**

7 Equation defined on a surface **97**
7.1 Approximation on an explicit surface mesh . . . 97
7.2 Building a surface mesh from a level set function . . . 106
7.3 The banded level set method . . . 109
7.4 A direct solver for the banded level set method . . . 111

8 The highly nonlinear p-laplacian problem **117**
8.1 Problem statement . . . 117
8.2 The fixed-point algorithm . . . 118
8.3 The Newton algorithm . . . 126
8.4 The damped Newton algorithm . . . 132
8.5 Error analysis . . . 135

IV Technical appendices **139**

A How to write a variational formulation ? **141**
A.1 The Green formula . . . 141
A.2 The vectorial Green formula . . . 141
A.3 The Green formula on a surface . . . 142

B How to prepare a mesh ? **143**
B.1 Bidimensional mesh with `bamg` . . . 143
B.2 Unidimensional mesh with `gmsh` . . . 144
B.3 Bidimensional mesh with `gmsh` . . . 145
B.4 Tridimensional mesh with `gmsh` . . . 146

C Migrating to Rheolef version 6.0 **149**
C.1 What is new in **Rheolef** 6.0 ? . . . 149
C.2 What should I have to change in my 5.x code ? . . . 149
C.3 New features in **Rheolef** 6.4 . . . 151

D GNU Free Documentation License **153**

List of example files **161**

List of commands **163**

Index **164**

Notations

Rheolef	mathematics	description
`d`	$d \in \{1,2,3\}$	dimension of the physical space
`dot(u,v)`	$\mathbf{u}.\mathbf{v} = \sum_{i=0}^{d-1} u_i v_i$	vector scalar product
`ddot(sigma,tau)`	$\sigma : \tau = \sum_{i,j=0}^{d-1} \sigma_{i,j}\tau_{i,j}$	tensor scalar product
`tr(sigma)`	$\mathrm{tr}(\sigma) = \sum_{i=0}^{d-1} \sigma_{i,i}$	trace of a tensor
`trans(sigma)`	σ^T	tensor transposition
`sqr(phi)` `norm2(phi)`	ϕ^2	square of a scalar
`norm2(u)`	$\lvert\mathbf{u}\rvert^2 = \sum_{i=0}^{d-1} u_i^2$	square of the vector norm
`norm2(sigma)`	$\lvert\sigma\rvert^2 = \sum_{i,j=0}^{d-1} \sigma_{i,j}^2$	square of the tensor norm
`abs(phi)` `norm(phi)`	$\lvert\phi\rvert$	absolute value of a scalar
`norm(u)`	$\lvert\mathbf{u}\rvert = \left(\sum_{i=0}^{d-1} u_i^2\right)^{1/2}$	vector norm
`norm(sigma)`	$\lvert\sigma\rvert = \left(\sum_{i,j=0}^{d-1} \sigma_{i,j}^2\right)^{1/2}$	tensor norm
`grad(phi)`	$\nabla\phi = \left(\frac{\partial\phi}{\partial x_i}\right)_{0\leqslant i<d}$	gradient of a scalar field in $\Omega \subset \mathbb{R}^d$
`grad(u)`	$\nabla\mathbf{u} = \left(\frac{\partial u_i}{\partial x_j}\right)_{0\leqslant i,j<d}$	gradient of a vector field
`div(u)`	$\mathrm{div}(\mathbf{u}) = \mathrm{tr}(\nabla\mathbf{u}) = \sum_{i=0}^{d-1} \frac{\partial u_i}{\partial x_i}$	divergence of a vector field
`D(u)`	$D(\mathbf{u}) = \left(\nabla\mathbf{u} + \nabla\mathbf{u}^T\right)/2$	symmetric part of the gradient of a vector field
`curl(u)`	$\mathbf{curl}(\mathbf{u}) = \nabla \wedge \mathbf{u}$	curl of a vector field, when $d = 3$
`curl(phi)`	$\mathbf{curl}(\phi) = \left(\frac{\partial\phi}{\partial x_1}, -\frac{\partial\phi}{\partial x_0}\right)$	curl of a scalar field, when $d = 2$
`curl(u)`	$\mathrm{curl}(\mathbf{u}) = \frac{\partial u_1}{\partial x_0} - \frac{\partial u_0}{\partial x_1}$	curl of a vector field, when $d = 2$
`grad_s(phi)`	$\nabla_s\phi = P\nabla\phi$ where $P = I - \mathbf{n}\otimes\mathbf{n}$	tangential gradient of a scalar

Rheolef	mathematics	description
`grad_s(u)`	$\nabla_s \mathbf{u} = P \nabla \mathbf{u} P$ TODO: $\nabla_s \mathbf{u} = \nabla \mathbf{u} P$	tangential gradient of a vector
`Ds(u)`	$D_s(\mathbf{u}) = P D(\mathbf{u}) P$	symmetrized tangential gradient
`div_s(u)`	$\mathrm{div}_s(\mathbf{u}) = \mathrm{tr}(D_s(\mathbf{u}))$	tangential divergence
`normal()`	$\mathbf{n}$	unit outward normal on $\Gamma = \partial\Omega$ or on an oriented surface Ω or on an internal oriented side S
`jump(phi)`	$[\![\phi]\!] = \phi_{\|K_0} - \phi_{\|K_1}$	jump accros inter-element side $S = \partial K_0 \cap K_1$
`average(phi)`	$\{\!\{\phi\}\!\} = (\phi_{\|K_0} + \phi_{\|K_1})/2$	average accross S
`inner(phi)`	$\phi_{\|K_0}$	inner trace on S
`outer(phi)`	$\phi_{\|K_1}$	outer trace on S
`h_local()`	$h_K = \mathrm{meas}(K)^{1/d}$	length scale on an element K
`penalty()`	$\varpi_s = \max\left(\frac{\mathrm{meas}(\partial K_0)}{\mathrm{meas}(K_0)}, \frac{\mathrm{meas}(\partial K_1)}{\mathrm{meas}(K_1)}\right)$	penalty coefficient on S
`grad_h(phi)`	$(\nabla_h \phi)_{\|K} = \nabla(\phi_{\|K}), \forall K \in \mathcal{T}_h$	broken gradient
`div_h(u)`	$(\mathrm{div}_h \mathbf{u})_{\|K} = \mathrm{div}(\mathbf{u}_{\|K}), \forall K \in \mathcal{T}_h$	broken divergence of a vector field
`Dh(u)`	$(D_h(\mathbf{u}))_{\|K} = D(\mathbf{u}_{\|K}), \forall K \in \mathcal{T}_h$	broken symmetric part of the gradient of a vector field
`sin(phi)`	$\sin(\phi)$	standard mathematical functions extended to scalar fields
`cos(phi)`	$\cos(\phi)$	
`tan(phi)`	$\tan(\phi)$	
`acos(phi)`	$\cos^{-1}(\phi)$	
`asin(phi)`	$\sin^{-1}(\phi)$	
`atan(phi)`	$\tan^{-1}(\phi)$	
`cosh(phi)`	$\cosh(\phi)$	
`sinh(phi)`	$\sinh(\phi)$	
`tanh(phi)`	$\tanh(\phi)$	
`exp(phi)`	$\exp(\phi)$	
`log(phi)`	$\log(\phi)$	
`log10(phi)`	$\log 10(\phi)$	
`floor(phi)`	$\lfloor \phi \rfloor$	largest integral not greater than ϕ
`ceil(phi)`	$\lceil \phi \rceil$	smallest integral not less than ϕ

Rheolef	mathematics	description
`min(phi,psi)`	$\min(\phi,\psi)$	
`max(phi,psi)`	$\max(\phi,\psi)$	
`pow(phi,psi)`	ϕ^ψ	
`atan2(phi,psi)`	$\tan^{-1}(\psi/\phi)$	
`fmod(phi,psi)`	$\phi - \lfloor \phi/\psi + 1/2 \rfloor \, \psi$	floating point remainder
`compose(f,phi)`	$f(\phi)$	applies an unary function f
`compose(f,phi,psi)`	$f(\phi,\psi)$	applies a binary function f
`compose(phi,X)`	$\phi \circ X$, $X(x) = x + \mathbf{d}(x)$	composition with a characteristic

Part I

Getting started with simple problems

The first part of this book starts with the Dirichlet problem with homogeneous boundary condition: this example is declined with details in dimension 1, 2 and 3, as a starting point to **Rheolef**.

Next chapters present various boundary conditions: for completeness, we treat non-homogeneous Dirichlet, Neumann, and Robin boundary conditions for model problems. The last two examples presents some special difficulties that appears in most problems: the first one introduce to problems with non-constant coefficients and the second one, a ill-posed problem where the solution is defined up to a constant.

This first part can be viewed as a pedagogic preparation for more advanced applications, such as Stokes and elasticity, that are treated in the second part of this book. Problem with non-constant coefficients are common as suproblems generated by various algorithms for non-linear problem.

Chapter 1

Getting started with Rheolef

For obtaining and installing **Rheolef**, see the installation instructions on the **Rheolef** home page:

http://www-ljk.imag.fr/membres/Pierre.Saramito/rheolef/

Before to run examples, please check your **Rheolef** installation with:

```
rheolef-config --check
```

The present book is available in the documentation directory of the **Rheolef** distribution. This documentation directory is given by the following unix command:

```
rheolef-config --docdir
```

All examples presented along the present book are also available in the `example/` directory of the **Rheolef** distribution. This directory is given by the following unix command:

```
rheolef-config --exampledir
```

This command returns you a path, something like `/usr/share/doc/rheolef-doc/examples/` and you should make a copy of these files:

```
cp -a /usr/share/doc/rheolef-doc/examples/ .
cd examples
```

1.1 The model problem

Let us consider the classical Poisson problem with homogeneous Dirichlet boundary conditions in a domain bounded $\Omega \subset \mathbb{R}^d$, $d = 1, 2, 3$:
(P): find u, defined in Ω, such that:

$$\begin{aligned} -\Delta u &= 1 \text{ in } \Omega & (1.1)\\ u &= 0 \text{ on } \partial\Omega & (1.2) \end{aligned}$$

where Δ denotes the Laplace operator. The variational formulation of this problem expresses (see appendix A.1 for details):
(VF): find $u \in H_0^1(\Omega)$ such that:

$$a(u,v) = l(v),\ \forall v \in H_0^1(\Omega) \quad (1.3)$$

where the bilinear form $a(.,.)$ and the linear form $l(.)$ are defined by

$$\begin{aligned} a(u,v) &= \int_\Omega \nabla u.\nabla v \, \mathrm{d}x, \quad \forall u,v \in H_0^1(\Omega) \\ l(v) &= \int_\Omega v \, \mathrm{d}x, \quad \forall v \in L^2(\Omega) \end{aligned}$$

The bilinear form $a(.,.)$ defines a scalar product in $H_0^1(\Omega)$ and is related to the *energy* form. This form is associated to the $-\Delta$ operator.

1.2 Approximation

Let us introduce a mesh $\mathcal{T}_h$ of Ω and the finite dimensional space X_h of continuous piecewise polynomial functions.

$$X_h = \{v \in H^1(\Omega);\ v_{/K} \in P_k,\ \forall K \in \mathcal{T}_h\}$$

where $k = 1$ or 2. Let $V_h = X_h \cap H_0^1(\Omega)$ be the functions of X_h that vanishes on the boundary of Ω. The approximate problem expresses:
$(VF)_h$*: find* $u_h \in V_h$ *such that:*

$$a(u_h, v_h) = l(v_h),\ \forall v_h \in V_h$$

By developing u_h on a basis of V_h, this problem reduces to a linear system. The following C++ code implement this problem in the **Rheolef** environment.

Example file 1.1: `dirichlet.cc`

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
int main(int argc, char**argv) {
  environment rheolef (argc, argv);
  geo omega (argv[1]);
  space Xh (omega, argv[2]);
  Xh.block ("boundary");
  trial u (Xh); test v (Xh);
  form  a  = integrate (dot(grad(u),grad(v)));
  field lh = integrate (v);
  field uh (Xh);
  uh ["boundary"] = 0;
  solver sa (a.uu());
  uh.set_u() = sa.solve (lh.u() - a.ub()*uh.b());
  dout << uh;
}
```

1.3 Comments

This code applies for both one, two or three dimensional meshes and for both piecewise linear or quadratic finite element approximations. Four major classes are involved, namely: `geo`, `space`, `form` and `field`.

Let us now comment the code, line by line.

```
#include "rheolef.h"
```

The first line includes the **Rheolef** header file '`rheolef.h`'.

```
using namespace rheolef;
using namespace std;
```

By default, in order to avoid possible name conflicts when using another library, all class and function names are prefixed by `rheolef::`, as in `rheolef::space`. This feature is called the name space. Here, since there is no possible conflict, and in order to simplify the syntax, we drop all the `rheolef::` prefixes, and do the same with the standard `c++` library classes and variables, that are also prefixed by `std::`.

```
int main(int argc, char**argv) {
```

The entry function of the program is always called `main` and accepts arguments from the unix command line: `argc` is the counter of command line arguments and `argv` is the table of values. The character string `argv[0]` is the program name and `argv[i]`, for $i = 1$ to `argc-1`, are the additional command line arguments.

```
environment rheolef (argc, argv);
```

These two command line parameters are immediately furnished to the distributed environment initializer of the `boost::mpi` library, that is a `c++` library based on the usual message passing interface (MPI) library. Notice that this initialization is required, even when you run with only one processor.

```
geo omega (argv[1]);
```

This command get the first unix command-line argument `argv[1]` as a mesh file name and store the corresponding mesh in the variable `omega`.

```
space Xh (omega, argv[2]);
```

Build the finite element space `Xh` contains all the piecewise polynomial continuous functions. The polynomial type is the second command-line arguments `argv[2]`, and could be either `P1`, `P2` or any `P`k, where $k \geqslant 1$.

```
Xh.block ("boundary");
```

The homogeneous Dirichlet conditions are declared on the boundary.

```
trial u (Xh); test v (Xh);
form a = integrate (dot(grad(u),grad(v)));
```

The bilinear form $a(.,.)$ is the energy form: it is defined for all functions u and v in X_h.

```
field lh = integrate (v);
```

The linear form $lh(.)$ is associated to the constant right-hand side $f = 1$ of the problem. It is defined for all v in X_h.

```
field uh (Xh);
```

The field `uh` contains the the degrees of freedom.

```
uh ["boundary"] = 0;
```

Some degrees of freedom are prescribed as zero on the boundary.

Let $(\varphi_i)_{0 \leqslant i < \dim(X_h)}$ be the basis of X_h associated to the Lagrange nodes, e.g. the vertices of the mesh for the P_1 approximation and the vertices and the middle of the edges for the P_2 approximation. The approximate solution u_h expresses as a linear combination of the continuous piecewise polynomial functions (φ_i):

$$u_h = \sum_i u_i \varphi_i$$

Thus, the field u_h is completely represented by its coefficients (u_i). The coefficients (u_i) of this combination are grouped into to sets: some have zero values, from the boundary condition and are related to *blocked* coefficients, and some others are *unknown*. Blocked coefficients are stored

into the `uh.b` array while unknown one are stored into `uh.u`. Thus, the restriction of the bilinear form $a(.,.)$ to $X_h \times X_h$ can be conveniently represented by a block-matrix structure:

$$a(u_h, v_h) = \begin{pmatrix} \texttt{vh.u} & \texttt{vh.b} \end{pmatrix} \begin{pmatrix} \texttt{a.uu} & \texttt{a.ub} \\ \texttt{a.bu} & \texttt{a.bb} \end{pmatrix} \begin{pmatrix} \texttt{uh.u} \\ \texttt{uh.b} \end{pmatrix}$$

This representation also applies for the linear form $l(.)$:

$$l(v_h) = \begin{pmatrix} \texttt{vh.u} & \texttt{vh.b} \end{pmatrix} \begin{pmatrix} \texttt{lh.u} \\ \texttt{lh.b} \end{pmatrix}$$

Thus, the problem $(VF)_h$ writes now:

$$\begin{pmatrix} \texttt{vh.u} & \texttt{vh.b} \end{pmatrix} \begin{pmatrix} \texttt{a.uu} & \texttt{a.ub} \\ \texttt{a.bu} & \texttt{a.bb} \end{pmatrix} \begin{pmatrix} \texttt{uh.u} \\ \texttt{uh.b} \end{pmatrix} = \begin{pmatrix} \texttt{vh.u} & \texttt{vh.b} \end{pmatrix} \begin{pmatrix} \texttt{lh.u} \\ \texttt{lh.b} \end{pmatrix}$$

for any `vh.u` and where `vh.b` $= 0$. After expansion, the problem reduces to *find* `uh.u` *such that*:

$$\texttt{a.uu} * \texttt{uh.u} = \texttt{l.u} - \texttt{a.ub} * \texttt{uh.b}$$

The resolution of this linear system for the `a.uu` matrix is then performed. A preliminary step build the LDL^T factorization:

```
solver sa (a.uu());
```

Then, the second step solves the *unknown part*:

```
uh.set_u() = sa.solve (lh.u() - a.ub()*uh.b());
```

When $d > 3$, a faster iterative strategy is automatically preferred by the `solver` class for solving the linear system: in that case, the preliminary step build an incomplete Choleski factorization preconditioner, while the second step runs an iterative method: the preconditioned conjugate gradient algorithm. Finally, the field is printed to standard output:

```
dout << uh;
```

The `dout` stream is a specific variable defined in the **Rheolef** library: it is a distributed and parallel extension of the usual `cout` stream in `C++`

1.4 How to compile the code

First, create a file '`Makefile`' as follow:

```
include $(shell rheolef-config --libdir)/rheolef/rheolef.mk
CXXFLAGS  = $(INCLUDES_RHEOLEF)
LDLIBS    = $(LIBS_RHEOLEF)
default: dirichlet
```

Then, enter:

```
make dirichlet
```

Now, your program, linked with **Rheolef**, is ready to run on a mesh.

1.5 How to run the program

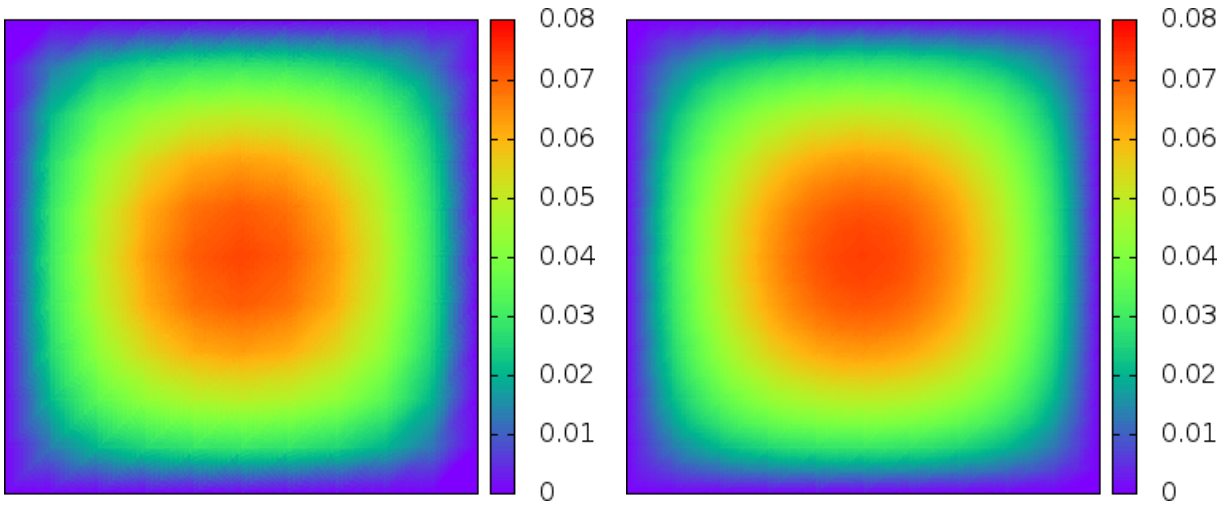


Figure 1.1: Solution of the model problem for $d = 2$: (left) P_1 element; (right) P_2 element.

Enter the commands:

```
mkgeo_grid -t 10 > square.geo
geo square.geo
```

The first command generates a simple 10x10 bidimensional mesh of $\Omega =]0,1[^2$ and stores it in the file `square.geo`. The second command shows the mesh. It uses `gnuplot` visualization program by default.

The next command performs the computation:

```
./dirichlet square.geo P1 > square.field
field square.field
```

1.6 Distributed and parallel runs

Alternatively, a computation in a distributed and parallel environment writes:

```
mpirun -np 4 ./dirichlet square.geo P1 > square.field
```

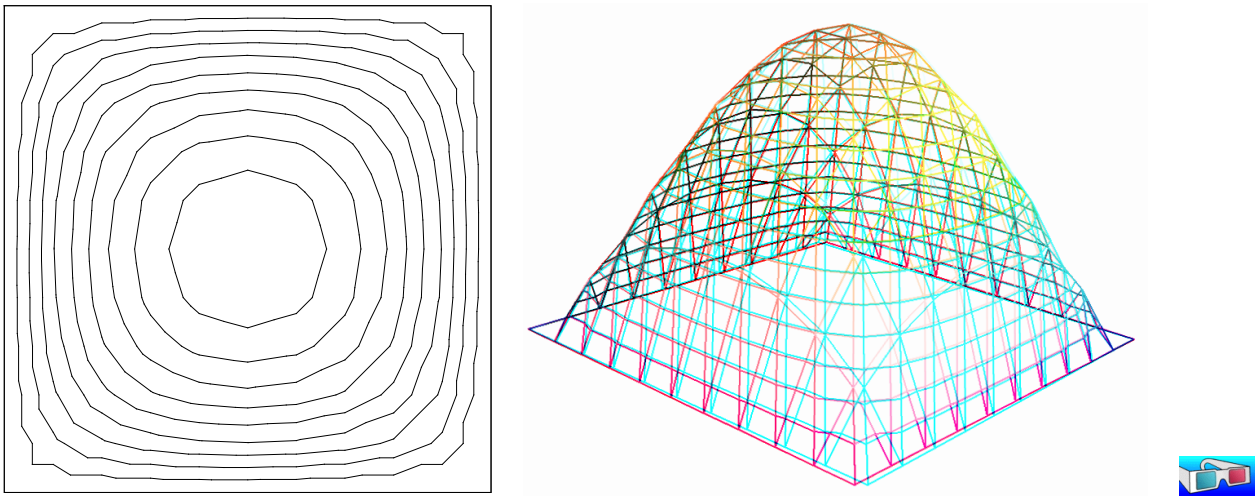


Figure 1.2: Alternative representations of the solution of the model problem ($d = 2$ and the P_1 element): (left) in black-and-white; (right) in elevation and stereoscopic anaglyph mode.

1.7 Stereo visualization

Also explore some graphic rendering modes (see Fig. 1.2):

```
field square.field -bw
field square.field -gray
field square.field -mayavi
field square.field -elevation -nofill -stereo
```

The last command shows the solution in elevation and in stereoscopic anaglyph mode (see Fig. 1.4, left). The anaglyph mode requires red-cyan glasses: red for the left eye and cyan for the right one, as shown on Fig. 1.3.

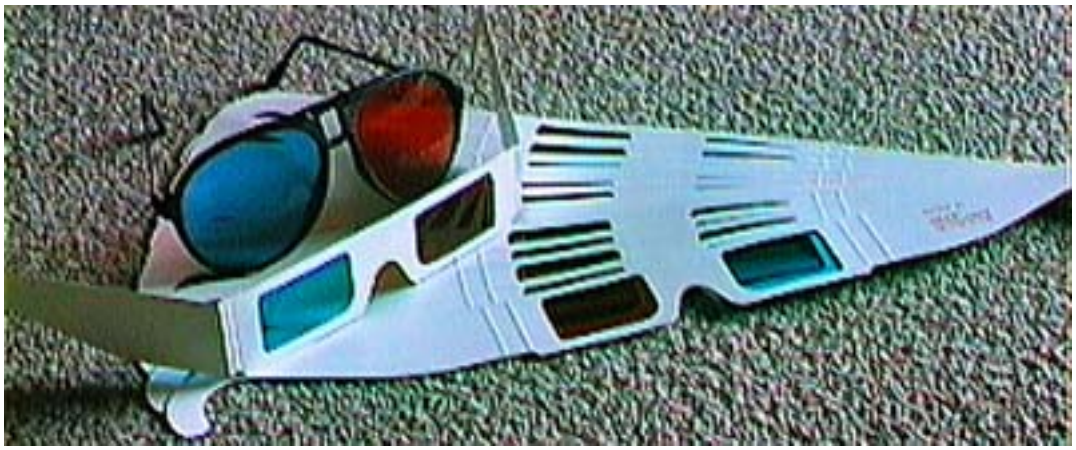


Figure 1.3: Red-cyan anaglyph glasses for the stereoscopic visualization.

In the book, stereo figures are indicated by the logo in the right margin. See http://en.wikipedia.org/wiki/Anaglyph_image for more and http://www.alpes-stereo.com/

lunettes.html for how to find anaglyph red-cyan glasses. Please, consults the corresponding unix manual page for more on `field`, `geo` and `mkgeo_grid`:

```
man mkgeo_grid
man geo
man field
```

1.8 High-order finite element methods

Turning to the `P2` or `P3` approximations simply writes:

```
./dirichlet square.geo P2 > square-P2.field
field square-P2.field
```

Fig. 1.1.right shows the result. You can replace the `P2` command-line argument by any `P`k, where $k \geqslant 1$. Now, let us consider a mono-dimensional problem $\Omega =]0,1[$:

```
mkgeo_grid -e 10 > line.geo
geo line.geo
./dirichlet line.geo P1 | field -
```

The first command generates a subdivision containing ten edge elements. The last two lines show the mesh and the solution via `gnuplot` visualization, respectively.
Conversely, the `P2` case writes:

```
./dirichlet line.geo P2 | field -
```

1.9 Tridimensional computations

Let us consider a three-dimensional problem $\Omega =]0,1[^3$. First, let us generate a mesh:

```
mkgeo_grid -T 10 > cube.geo
geo cube.geo
geo cube.geo -stereo -full
geo cube.geo -stereo -cut
```

The previous commands draw the mesh with all internal edges (`-full`), stereoscopic anaglyph (`-stereo`)and then with a cut (`-cut`) inside the internal structure: a simple click on the central arrow draws the cut plane normal vector or its origin, while the red square allows a translation.

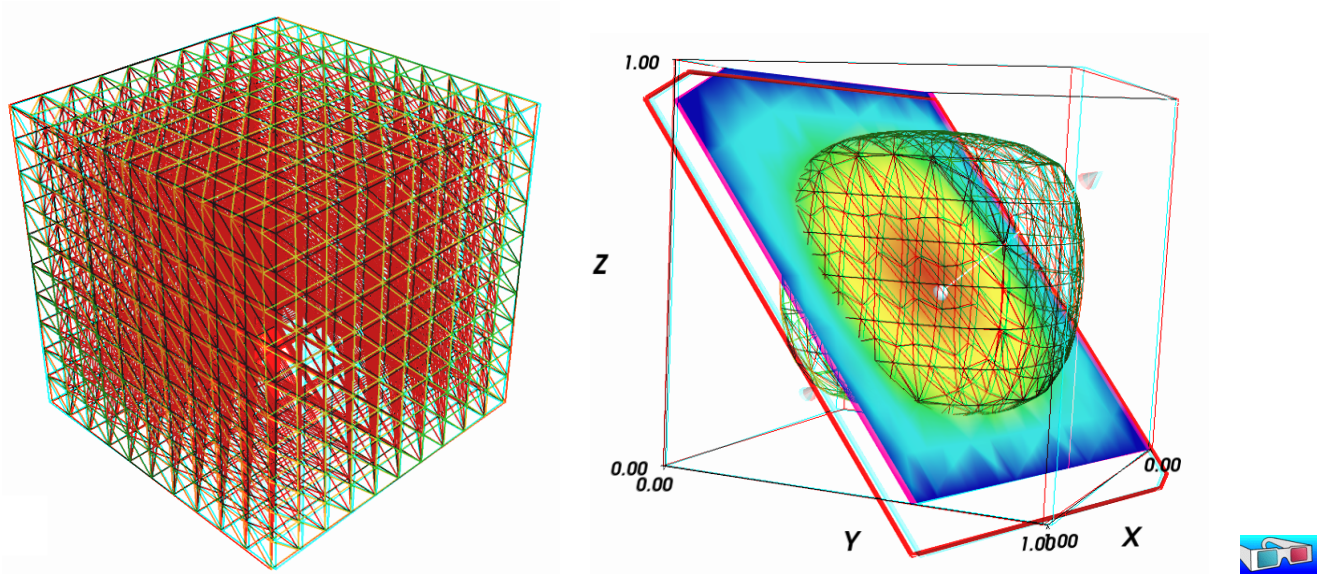


Figure 1.4: Solution of the model problem for $d = 3$ and the P_1 element : (left) mesh; (right) isovalue, cut planes and stereo anaglyph renderings.

Then, we perform the computation and the visualization:

```
./dirichlet cube.geo P1 > cube.field
field cube.field
```

The visualization presents an isosurface. Also here, you can interact with the cutting plane. Click on `IsoSurface` in the left menu and change the value of the isosurface. Finally exit from the visualization and explore the stereoscopic anaglyph mode (see Fig. 1.4, right):

```
field cube.field -stereo
```

It is also possible to add a second `IsoSurface` or `ScalarCutPlane` module to this scene by using the `Visualize` menu. After this exploration of the 3D visualization capacities of our environment, let us go back to the Dirichlet problem and perform the `P2` approximation:

```
./dirichlet cube.geo P2 | field -
```

1.10 Quadrangles, prisms and hexahedra

Quadrangles and hexahedra are also supported in meshes:

```
mkgeo_grid -q 10 > square.geo
geo square.geo
mkgeo_grid -H 10 > cube.geo
geo cube.geo
```

Notices also that the one-dimensional exact solution writes:

$$u(x) = \frac{x(1-x)}{2}$$

while the two-and three dimensional ones support a Fourier expansion (see e.g. [45], annex).

Chapter 2

Standard boundary conditions

We show how to deal with various non-homogeneous boundary conditions of Dirichlet, Neuman and Robin type.

2.1 Non-homogeneous Dirichlet conditions

Formulation

We turn now to the case of a non-homogeneous Dirichlet boundary conditions. Let $f \in H^{-1}(\Omega)$ and $g \in H^{\frac{1}{2}}(\partial\Omega)$. The problem writes:
(P_2) *find u, defined in Ω such that:*

$$\begin{aligned} -\Delta u &= f \text{ in } \Omega \\ u &= g \text{ on } \partial\Omega \end{aligned}$$

The variational formulation of this problem expresses:
(VF_2) *find $u \in V$ such that:*

$$a(u,v) = l(v),\ \forall v \in V_0$$

where

$$\begin{aligned} a(u,v) &= \int_\Omega \nabla u.\nabla v \,\mathrm{d}x \\ l(v) &= \int_\Omega f\, v \,\mathrm{d}x \\ V &= \{v \in H^1(\Omega);\ v_{|\partial\Omega} = g\} \\ V_0 &= H_0^1(\Omega) \end{aligned}$$

Approximation

As usual, we introduce a mesh $\mathcal{T}_h$ of Ω and the finite dimensional space X_h:

$$X_h = \{v \in H^1(\Omega);\ v_{/K} \in P_k,\ \forall K \in \mathcal{T}_h\}$$

Then, we introduce:

$$\begin{aligned} V_h &= \{v \in X_h;\ v_{|\partial\Omega} = \pi_h(g)\} \\ V_{0,h} &= \{v \in X_h;\ v_{|\partial\Omega} = 0\} \end{aligned}$$

where π_h denotes the Lagrange interpolation operator. The approximate problem writes: $(VF_2)_h$: *find* $u_h \in V_h$ *such that:*

$$a(u_h, v_h) = l(v_h), \ \forall v_h \in V_{0,h}$$

The following C++ code implement this problem in the **Rheolef** environment.

Example file 2.1: `dirichlet-nh.cc`

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
#include "cosinusprod_laplace.icc"
int main(int argc, char**argv) {
  environment rheolef(argc, argv);
  geo omega (argv[1]);
  size_t d = omega.dimension();
  space Xh (omega, argv[2]);
  Xh.block ("boundary");
  trial u (Xh); test v (Xh);
  form  a  = integrate (dot(grad(u),grad(v)));
  field lh = integrate (f(d)*v);
  field uh (Xh);
  space Wh (omega["boundary"], argv[2]);
  uh ["boundary"] = interpolate(Wh, g(d));
  solver sa (a.uu());
  uh.set_u() = sa.solve (lh.u() - a.ub()*uh.b());
  dout << uh;
}
```

Let us choose $\Omega \subset \mathbb{R}^d$, $d = 1, 2, 3$ with

$$f(x) = d\,\pi^2 \prod_{i=0}^{d-1} \cos(\pi x_i) \quad \text{and} \quad g(x) = \prod_{i=0}^{d-1} \cos(\pi x_i) \tag{2.1}$$

Remarks the notation $x = (x_0, \ldots, x_{d-1})$ for the Cartesian coordinates in $\mathbb{R}^d$: since all arrays start at index zero in `C++` codes, and in order to avoid any mistakes between the code and the mathematical formulation, we also adopt this convention here. This choice of f and g is convenient, since the exact solution is known:

$$u(x) = \prod_{i=0}^{d-1} \cos(\pi x_i)$$

The following C++ code implement this problem by using the concept of *function object*, also called *class-function* (see e.g. [29]). A convenient feature is the ability for function objects to store auxiliary parameters, such as the space dimension d for f here, or some constants, as π for f and g.

Example file 2.2: `cosinusprod_laplace.icc`

```
struct f : field_functor<f,Float> {
  Float operator() (const point& x) const {
    return d*pi*pi*cos(pi*x[0])*cos(pi*x[1])*cos(pi*x[2]); }
  f(size_t d1) : d(d1), pi(acos(Float(-1))) {}
  size_t d; const Float pi;
};
struct g : field_functor<g,Float> {
  Float operator() (const point& x) const {
    return cos(pi*x[0])*cos(pi*x[1])*cos(pi*x[2]); }
  g(size_t d1) : pi(acos(Float(-1))) {}
  const Float pi;
};
```

Comments

The class `point` describes the coordinates of a point $(x_0, \ldots, x_{d-1}) \in \mathbb{R}^d$ as a d-uplet of `Float`. The `Float` type is usually a `double`. This type depends upon the **Rheolef** configuration (see [42], installation instructions), and could also represent some high precision floating point class. The `dirichlet-nh.cc` code looks like the previous one `dirichlet.cc` related to homogeneous boundary conditions. Let us comments the changes. The dimension d comes from the geometry Ω:

```
size_t d = omega.dimension();
```

The linear form $l(.)$ is associated to the right-hand side f and writes:

```
field lh = integrate (f(d)*v);
```

Notice that the function object f is build with the dimension d as parameter. Notice also the use of `field_functor`[1] in the definition of the class `f`: this trick allows us to mixt functions, fields and test-functions in the same expression, as $f(d) * v$.

The space W_h of piecewise P_k functions defined on the boundary $\partial\Omega$ is defined by:

```
space Wh (omega["boundary"], argv[2]);
```

where P_k is defined via the second command line argument `argv[2]`. This space is suitable for the Lagrange interpolation of g on the boundary:

```
uh ["boundary"] = interpolate(Wh, g(d));
```

The values of the degrees of freedom related to the boundary are stored into the field `uh.b`, where non-homogeneous Dirichlet conditions applies. The rest of the code is similar to the homogeneous Dirichlet case.

2.1.1 How to run the program

First, compile the program:

```
make dirichlet-nh
```

Running the program is obtained from the homogeneous Dirichlet case, by replacing `dirichlet` by `dirichlet-nh`:

```
mkgeo_grid -e 10 > line.geo
./dirichlet-nh line.geo P1 > line.field
field line.field
```

for the bidimensional case:

```
mkgeo_grid -t 10 > square.geo
./dirichlet-nh square.geo P1 > square.field
field square.field
```

and for the tridimensional case:

```
mkgeo_grid -T 10 > box.geo
./dirichlet-nh box.geo P1 > box.field
field -mayavi box.field
```

Here, the `P1` approximation can be replaced by `P2`, `P3`, etc, by modifying the command-line argument.

[1] The actual implementation of a `field_functor` class bases on the *curiously recurring template pattern* (CRTP) C++ idiom: the definition of the class `f` derives from `field_functor`<f,Float> that depend itself upon `f`. So, be carrefull when using copy-paste, as there a no checks if you write e.g. `field_functor`<g,Float> with another function `g` instead of `f`.

2.1.2 Error analysis

Principle

Since the solution u is regular, the following error estimates holds:

$$\begin{aligned} \|u-u_h\|_{0,2,\Omega} &\approx \mathcal{O}(h^{k+1}) \\ \|u-u_h\|_{0,\infty,\Omega} &\approx \mathcal{O}(h^{k+1}) \\ \|u-u_h\|_{1,2,\Omega} &\approx \mathcal{O}(h^{k}) \end{aligned}$$

providing the approximate solution u_h uses P_k continuous finite element method, $k \geqslant 1$. Here, $\|.\|_{0,2,\Omega}$, $\|.\|_{0,\infty,\Omega}$ and $\|.\|_{1,2,\Omega}$ denotes as usual the $L^2(\Omega)$, $L^\infty(\Omega)$ and $H^1(\Omega)$ norms.
By denoting π_h the Lagrange interpolation operator, the triangular inequality leads to:

$$\|u-u_h\|_{0,2,\Omega} \leqslant \|(I-\pi_h)(u)\|_{0,2,\Omega} + \|u_h-\pi_h u\|_{0,2,\Omega}$$

From the fundamental properties of the Laplace interpolation π_h, and since u is smooth enough, we have $\|(I-\pi_h)(u)\|_{0,2,\Omega} \approx O(h^{k+1})$. Thus, we have just to check the $\|u_h-\pi_h u\|_{0,2,\Omega}$ term. The following code implement the computation of the error.

Example file 2.3: `cosinusprod_error.cc`

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
#include "cosinusprod.icc"
int main(int argc, char**argv) {
  environment rheolef(argc, argv);
  Float error_linf_expected = (argc > 1) ? atof(argv[1]) : 1e+38;
  field uh; din >> uh;
  space Xh = uh.get_space();
  size_t d = Xh.get_geo().dimension();
  field pi_h_u = interpolate(Xh, u_exact(d));
  field eh = uh - pi_h_u;
  trial u (Xh); test v (Xh);
  form m = integrate (u*v);
  form a = integrate (dot(grad(u),grad(v)));
  dout << "error_l2   " << sqrt(m(eh,eh)) << endl
       << "error_linf " << eh.max_abs()   << endl
       << "error_h1   " << sqrt(a(eh,eh)) << endl;
  return (eh.max_abs() <= error_linf_expected) ? 0 : 1;
}
```

Example file 2.4: `cosinusprod.icc`

```
struct u_exact : field_functor<u_exact,Float> {
  Float operator() (const point& x) const {
    return cos(pi*x[0])*cos(pi*x[1])*cos(pi*x[2]); }
  u_exact(size_t d1) : d(d1), pi(acos(Float(-1.0))) {}
  size_t d; Float pi;
};
```

The $m(.,.)$ is here the classical scalar product on $L^2(\Omega)$, and is related to the *mass* form.

Running the program

```
make dirichlet-nh cosinusprod_error
```

After compilation, run the code by using the command:

```
mkgeo_grid -t 10 > square.geo
./dirichlet-nh square.geo P1 | ./cosinusprod_error
```

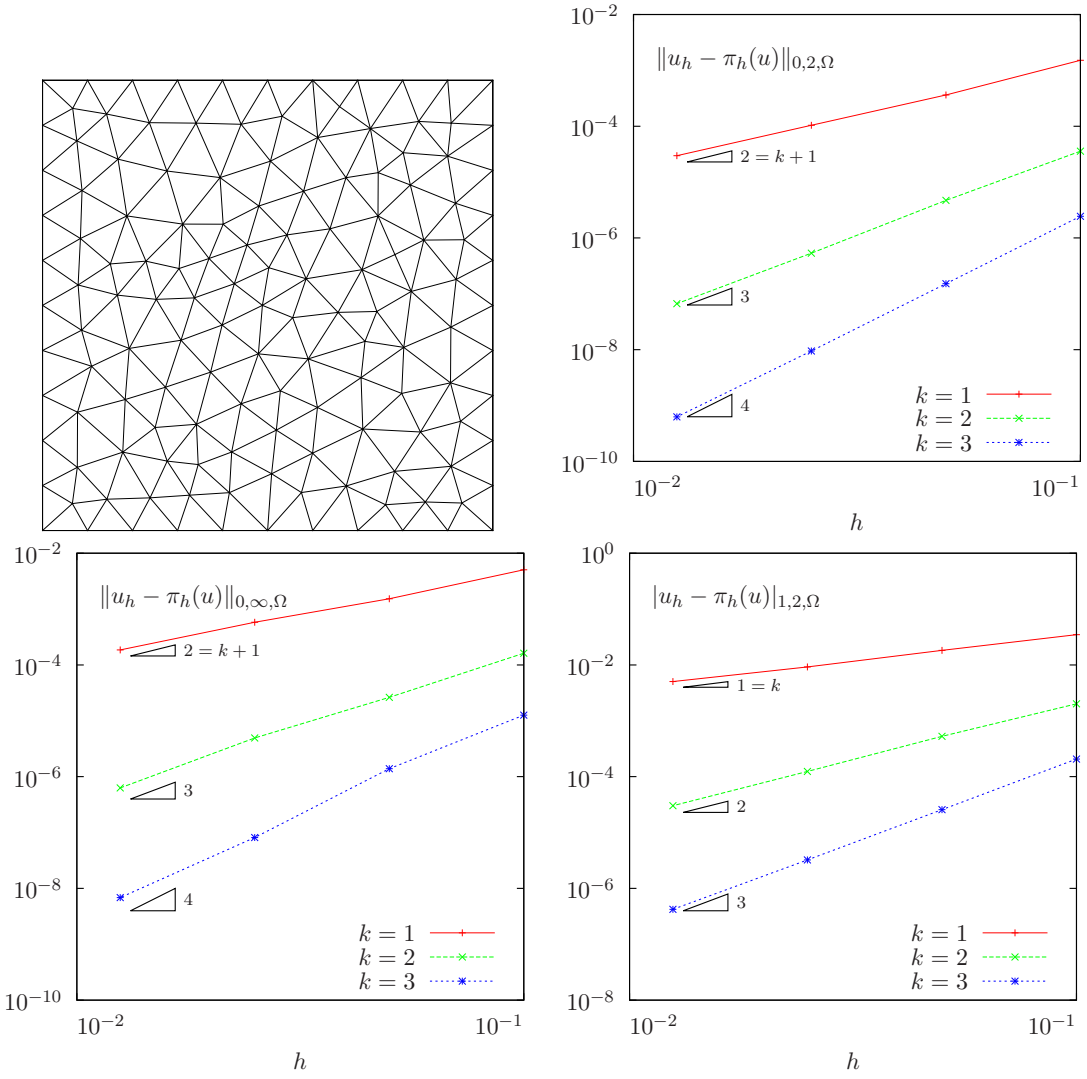


Figure 2.1: Strait geometry: error analysis in L^2, L^∞ and H^1 norms.

The three L^2, L^∞ and H^1 errors are printed for a $h = 1/10$ uniform mesh. Note that an unstructured quasi-uniform mesh can be simply generated by using the `mkgeo_ugrid` command:

```
mkgeo_ugrid -t 10 > square.geo
geo square.geo
```

Let n_{el} denotes the number of elements in the mesh. Since the mesh is quasi-uniform, we have $h \approx n_{\text{el}}^{\frac{1}{d}}$ where d is the physical space dimension. Here $d = 2$ for our bidimensional mesh. Figure 2.1 plots in logarithmic scale the error versus $n_{\text{el}}^{\frac{1}{2}}$ for both P_k approximations, $k = 1, 2, 3$ and the various norms. Observe that the error behaves as predicted by the theory.

Curved domains

The error analysis applies also for curved boundaries and high order approximations.

Example file 2.5: `cosinusrad_laplace.icc`

```
struct f : field_functor<f,Float> {
  Float operator() (const point& x) const {
    Float r = sqrt(sqr(x[0])+sqr(x[1])+sqr(x[2]));
    Float sin_over_ar = (r == 0) ? 1 : sin(a*r)/(a*r);
    return sqr(a)*((d-1)*sin_over_ar + cos(a*r)); }
  f(size_t d1) : d(d1), a(acos(Float(-1.0))) {}
  size_t d; Float a;
};
struct g : field_functor<g,Float> {
  Float operator() (const point& x) const {
    return cos(a*sqrt(sqr(x[0])+sqr(x[1])+sqr(x[2]))); }
  g(size_t=0) : a(acos(Float(-1.0))) {}
  Float a;
};
```

Example file 2.6: `cosinusrad.icc`

```
struct u_exact : field_functor<u_exact,Float> {
  Float operator() (const point& x) const {
    Float r = sqrt(sqr(x[0])+sqr(x[1])+sqr(x[2]));
    return cos(a*r); }
  u_exact(size_t=0) : a(acos(Float(-1.0))) {}
  Float a;
};
```

First, generate the test source file and compile it:

```
sed -e 's/sinusprod/sinusrad/' < dirichlet-nh.cc      > dirichlet_nh_ball.cc
sed -e 's/sinusprod/sinusrad/' < cosinusprod_error.cc > cosinusrad_error.cc
make dirichlet_nh_ball cosinusrad_error
```

Then, generates the mesh of a circle and run the test case:

```
mkgeo_ball -order 1 -t 10 > circle-P1.geo
geo circle-P1
./dirichlet_nh_ball circle-P1.geo P1 | ./cosinusrad_error
```

For a high order $k = 3$ isoparametric approximation:

```
mkgeo_ball -order 3 -t 10 > circle-P3.geo
geo circle-P3
./dirichlet_nh_ball circle-P3.geo P3 | ./cosinusrad_error
```

Observe Fig. 2.2: for meshes based on triangles: the error behave as expected when $k = 1, 2, 3, 4$. A similar result occurs for quadrangles, as shown on Fig. 2.3.

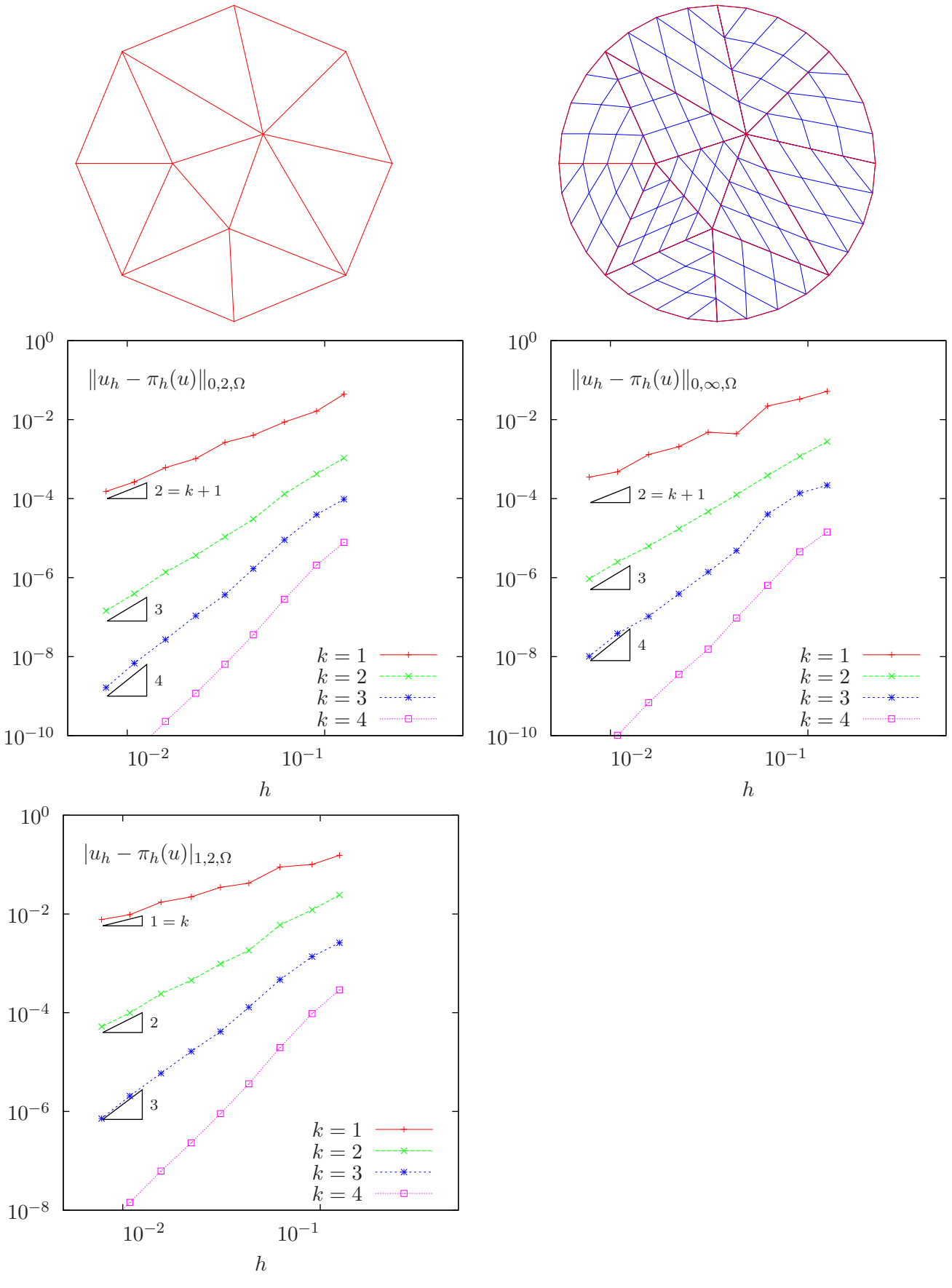


Figure 2.2: Curved domains (triangles): error analysis in L^2, L^∞ and H^1 norms.

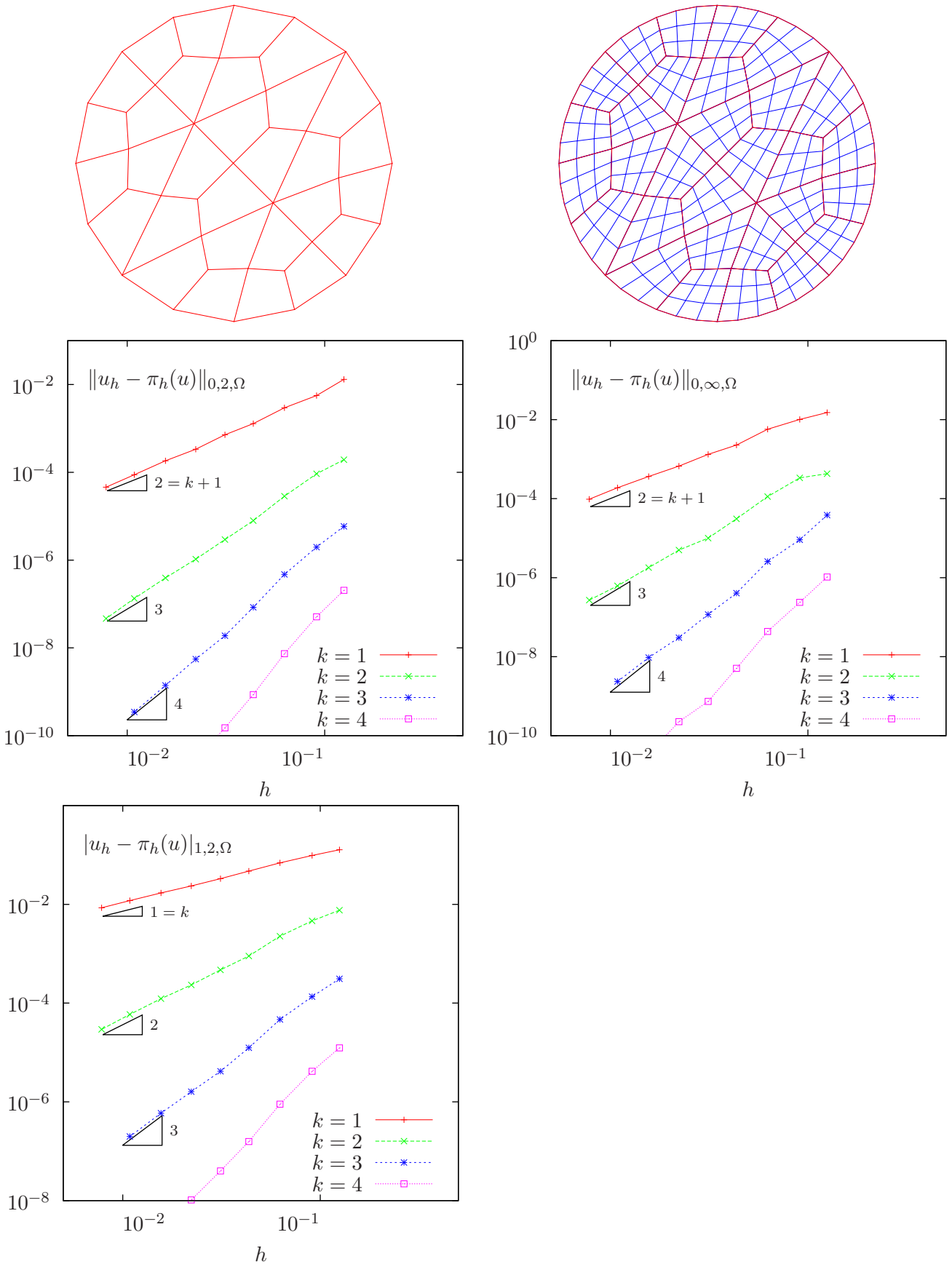


Figure 2.3: Curved domains (quadrangles): error analysis in L^2, L^∞ and H^1 norms.

```
mkgeo_ball -order 3 -q 10 > circle-q-P3.geo
geo circle-q-P3
./dirichlet_nh_ball circle-q-P3.geo P3 | ./cosinusrad_error
```

These features are currently in development for arbitrarily P_k high order approximations and three-dimensional geometries.

2.2 Non-homogeneous Neumann boundary conditions for the Helmholtz operator

Formulation

Let us show how to insert Neumann boundary conditions. Let $f \in H^{-1}(\Omega)$ and $g \in H^{-\frac{1}{2}}(\partial\Omega)$. The problem writes:
(P_3): *find u, defined in Ω such that:*

$$\begin{aligned} u - \Delta u &= f \text{ in } \Omega \\ \frac{\partial u}{\partial n} &= g \text{ on } \partial\Omega \end{aligned}$$

The variational formulation of this problem expresses:
(VF_3): *find $u \in H^1(\Omega)$ such that:*

$$a(u,v) = l(v),\ \forall v \in H^1(\Omega)$$

where

$$\begin{aligned} a(u,v) &= \int_\Omega (u\,v + \nabla u.\nabla v)\ \mathrm{d}x \\ l(v) &= \int_\Omega f\,v\,\mathrm{d}x + \int_{\partial\Omega} g\,v\,\mathrm{d}s \end{aligned}$$

Approximation

As usual, we introduce a mesh $\mathcal{T}_h$ of Ω and the finite dimensional space X_h:

$$X_h = \{v \in H^1(\Omega);\ v_{/K} \in P_k,\ \forall K \in \mathcal{T}_h\}$$

The approximate problem writes:
$(VF_3)_h$: *find* $u_h \in X_h$ *such that:*

$$a(u_h, v_h) = l(v_h),\ \forall v_h \in X_h$$

Example file 2.7: `neumann-nh.cc`

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
#include "sinusprod_helmholtz.icc"
int main(int argc, char**argv) {
  environment rheolef(argc, argv);
  geo omega (argv[1]);
  size_t d = omega.dimension();
  space Xh (omega, argv[2]);
  trial u (Xh); test v (Xh);
  form  a  = integrate (u*v + dot(grad(u),grad(v)));
  field lh = integrate (f(d)*v) + integrate ("boundary", g(d)*v);
  field uh (Xh);
  solver sa (a.uu());
  uh.set_u() = sa.solve (lh.u() - a.ub()*uh.b());
  dout << uh;
}
```

Let us choose $\Omega \subset \mathbb{R}^d$, $d = 1, 2, 3$ and

$$\begin{aligned} f(x) &= (1 + d\pi^2) \prod_{i=0}^{d-1} \sin(\pi x_i) \\ g(x) &= \begin{cases} -\pi & \text{when } d = 1 \\ -\pi \left(\sum_{i=0}^{d-1} \sin(\pi x_i) \right) & \text{when } d = 2 \\ -\pi \left(\sum_{i=0}^{d-1} \sin(\pi x_i) \sin(x_{(i+1) \bmod d} \right) & \text{when } d = 3 \end{cases} \end{aligned}$$

This example is convenient, since the exact solution is known:

$$u(x) = \prod_{i=0}^{d-1} \sin(\pi x_i) \tag{2.2}$$

Example file 2.8: `sinusprod_helmholtz.icc`

```
struct f : field_functor<f,Float> {
 Float operator() (const point& x) const {
  switch (d) {
  case 1:  return (1+d*pi*pi)*sin(pi*x[0]);
  case 2:  return (1+d*pi*pi)*sin(pi*x[0])*sin(pi*x[1]);
  default: return (1+d*pi*pi)*sin(pi*x[0])*sin(pi*x[1])*sin(pi*x[2]);
 }}
 f(size_t d1) : d(d1), pi(acos(Float(-1.0))) {}
 size_t d; const Float pi;
};
struct g : field_functor<g,Float> {
 Float operator() (const point& x) const {
  switch (d) {
  case 1:  return -pi;
  case 2:  return -pi*(sin(pi*x[0]) + sin(pi*x[1]));
  default: return -pi*( sin(pi*x[0])*sin(pi*x[1])
                      + sin(pi*x[1])*sin(pi*x[2])
                      + sin(pi*x[2])*sin(pi*x[0]));
 }}
 g(size_t d1) : d(d1), pi(acos(Float(-1.0))) {}
 size_t d; const Float pi;
};
```

Comments

The `neumann-nh.cc` code looks like the previous one `dirichlet-nh.cc`. Let us comments only the changes.

```
  form  a  = integrate (u*v + dot(grad(u),grad(v)));
```

The bilinear form $a(.,.)$ is defined. Notes the flexibility of the `integrate` function that takes as argument an expression involving the trial and test functions. The right-hand side is computed as:

```
  field lh = integrate (f(d)*v) + integrate ("boundary", g(d)*v);
```

The second integration is perfomed on $\partial\Omega$. The additional first argument of the `integrate` function is here the name of the integration domain.

2.2.1 How to run the program

First, compile the program:

```
  make neumann-nh
```

Running the program is obtained from the homogeneous Dirichlet case, by replacing `dirichlet` by `neumann-nh`.

2.3 The Robin boundary conditions

Formulation

Let $f \in H^{-1}(\Omega)$ and Let $g \in H^{\frac{1}{2}}(\partial\Omega)$. The problem writes:
(P_4) *find u, defined in Ω such that:*

$$\begin{aligned} -\Delta u &= f \text{ in } \Omega \\ \frac{\partial u}{\partial n} + u &= g \text{ on } \partial\Omega \end{aligned}$$

The variational formulation of this problem expresses:
(VF_4): *find* $u \in H^1(\Omega)$ *such that:*

$$a(u,v) = l(v),\ \forall v \in H^1(\Omega)$$

where

$$\begin{aligned} a(u,v) &= \int_\Omega \nabla u.\nabla v \,\mathrm{d}x + \int_{\partial\Omega} uv \,\mathrm{d}s \\ l(v) &= \int_\Omega uv \,\mathrm{d}x + \int_{\partial\Omega} gv \,\mathrm{d}s \end{aligned}$$

Approximation

As usual, let

$$X_h = \{v \in H^1(\Omega);\ v_{/K} \in P_k,\ \forall K \in \mathcal{T}_h\}$$

The approximate problem writes:
$(VF_4)_h$: *find* $u_h \in X_h$ *such that:*

$$a(u_h, v_h) = l(v_h),\ \forall v_h \in X_h$$

Example file 2.9: `robin.cc`

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
#include "cosinusprod_laplace.icc"
int main(int argc, char**argv) {
  environment rheolef(argc, argv);
  geo omega (argv[1]);
  size_t d = omega.dimension();
  space Xh (omega, argv[2]);
  trial u (Xh); test v (Xh);
  form  a  = integrate (dot(grad(u),grad(v))) + integrate ("boundary", u*v);
  field lh = integrate (f(d)*v) + integrate ("boundary", g(d)*v);
  field uh (Xh);
  solver sa (a.uu());
  uh.set_u() = sa.solve (lh.u() - a.ub()*uh.b());
  dout << uh;
}
```

Comments

The code `robin.cc` looks like the previous one `neumann-nh.cc`. Let us comments the changes.

```
  form  a  = integrate (dot(grad(u),grad(v))) + integrate ("boundary", u*v);
```

This statement reflects directly the definition of the bilinear form $a(.,.)$, as the sum of two integrals, the first one over Ω and the second one over its boundary.

2.3.1 How to run the program

First, compile the program:

```
make robin
```

Running the program is obtained from the homogeneous Dirichlet case, by replacing `dirichlet` by `robin`.

2.4 Neumann boundary conditions for the Laplace operator

In this chapter we study how to solve a ill-posed problem with a solution defined up to a constant.

Formulation

Let Ω be a bounded open and simply connected subset of $\mathbb{R}^d$, $d = 1, 2$ or 3. Let $f \in L^2(\Omega)$ and $g \in H^{\frac{1}{2}}(\partial\Omega)$ satisfying the following compatibility condition:

$$\int_\Omega f \,\mathrm{d}x + \int_{\partial\Omega} g \,\mathrm{d}s = 0$$

The problem writes:
$(P_5)_h$: *find u, defined in Ω such that:*

$$\begin{aligned} -\Delta u &= f \text{ in } \Omega \\ \frac{\partial u}{\partial n} &= g \text{ on } \partial\Omega \end{aligned}$$

Since this problem only involves the derivatives of u, it is clear that its solution is never unique [19, p. 11]. A discrete version of this problem could be solved iteratively by the conjugate gradient or the MINRES algorithm [34]. In order to solve it by a direct method, we turn the difficulty by seeking u in the following space

$$V = \{v \in H^1(\Omega); \quad b(v,1) = 0\}$$

where

$$b(v,\mu) = \int_\Omega v \,\mathrm{d}x, \quad \forall v \in L^2(\Omega), \forall \mu \in \mathbb{R}$$

The variational formulation of this problem writes:
(VF_5): *find $u \in V$ such that:*

$$a(u,v) = l(v), \ \forall v \in V$$

where

$$\begin{aligned} a(u,v) &= \int_\Omega \nabla u.\nabla v \, dx \\ l(v) &= m(f,v) + m_b(g,v) \\ m(f,v) &= \int_\Omega f v \, dx \\ m_b(g,v) &= \int_{\partial\Omega} g v \, ds \end{aligned}$$

Since the direct discretization of the space V is not an obvious task, the constraint $b(u,1) = 0$ is enforced by a Lagrange multiplier $\lambda \in \mathbb{R}$. Let us introduce the Lagrangian, defined for all $v \in H^1(\Omega)$ and $\mu \in \mathbb{R}$ by:

$$L(v,\mu) = \frac{1}{2}a(v,v) + b(v,\mu) - l(v)$$

The saddle point $(u,\lambda) \in H^1(\Omega) \times \mathbb{R}$ of this Lagrangian is characterized as the unique solution of:

$$\begin{aligned} a(u,v) + b(v,\lambda) &= l(v), \quad \forall v \in H^1(\Omega) \\ b(u,\mu) &= 0, \qquad \forall \mu \in \mathbb{R} \end{aligned}$$

It is clear that if (u,λ) is solution of this problem, then $u \in V$ and u is a solution of (VF_5). Conversely, let $u \in V$ the solution of (VF_5). Choosing $v = v_0$ where $v_0(x) = 1$, $\forall x \in \Omega$ leads to $\lambda \,\mathrm{meas}(\Omega) = l(v_0)$. From the definition of $l(.)$ and the compatibility condition between the data f and g, we get $\lambda = 0$. Notice that the saddle point problem extends to the case when f and g does not satisfies the compatibility condition, and in that case $\lambda = l(v_0)/\mathrm{meas}(\Omega)$.

Approximation

As usual, we introduce a mesh $\mathcal{T}_h$ of Ω and the finite dimensional space X_h:

$$X_h = \{v \in H^1(\Omega);\ v_{/K} \in P_k,\ \forall K \in \mathcal{T}_h\}$$

The approximate problem writes:
$(VF_5)_h$: *find* $(u_h, \lambda_h) \in X_h \times \mathbb{R}$ *such that:*

$$\begin{array}{rcll} a(u_h, v) + b(v, \lambda_h) & = & l_h(v), & \forall v \in X_h \\ b(u_h, \mu) & = & 0, & \forall \mu \in \mathbb{R} \end{array}$$

where

$$l_h(v) = m(\Pi_h f, v_h) + m_b(\pi_h g, v_h)$$

Example file 2.10: `neumann-laplace.cc`

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
size_t d;
Float f (const point& x) { return 1; }
Float g (const point& x) { return -0.5/d; }
int main(int argc, char**argv) {
  environment rheolef (argc, argv);
  geo omega (argv[1]);
  d = omega.dimension();
  space Xh (omega, argv[2]);
  trial u (Xh); test v (Xh);
  form  m  = integrate (u*v);
  form  a  = integrate (dot(grad(u),grad(v)));
  field b  = m*field(Xh,1);
  field lh = integrate (f*v) + integrate ("boundary", g*v);
  csr<Float> A = {{  a.uu(),     b.u()},
                  {trans(b.u()),  0   }};
  vec<Float> B =  {  lh.u(),     0   };
  A.set_symmetry(true);
  solver sa = ldlt(A);
  vec<Float> U = sa.solve (B);
  field uh(Xh);
  uh.set_u() = U [range(0,uh.u().size())];
  Float lambda = (U.size() == uh.u().size()+1) ? U [uh.u().size()] : 0;
#ifdef _RHEOLEF_HAVE_MPI
  mpi::broadcast (U.comm(), lambda, U.comm().size() - 1);
#endif // _RHEOLEF_HAVE_MPI
  dout << uh
       << "lambda" << lambda << endl;
}
```

Comments

Let $\Omega \subset \mathbb{R}^d$, $d = 1, 2, 3$. We choose $f(x) = 1$ and $g(x) = -1/(2d)$. This example is convenient, since the exact solution is known:

$$u(x) = -\frac{1}{12} + \frac{1}{2d}\sum_{i=1}^{d} x_i(1 - x_i)$$

The code looks like the previous ones. Let us comment the changes. The discrete bilinear form b is computed as $b_h \in X_h$ that interprets as a linear application from X_h to $\mathbb{R}$: $b_h(v_h) = m(v_h, 1)$. Thus b_h is computed as

```
  field b = m*field(Xh,1.0);
```

where the discrete bilinear form m is identified to its matrix and `field(Xh,1.0)` is the constant vector equal to 1. Let

$$\mathcal{A} = \begin{pmatrix} \texttt{a.uu} & \texttt{trans(b.u)} \\ \texttt{b.u} & 0 \end{pmatrix}, \quad \mathcal{U} = \begin{pmatrix} \texttt{uh.u} \\ \texttt{lambda} \end{pmatrix}, \quad \mathcal{B} = \begin{pmatrix} \texttt{lh.u} \\ 0 \end{pmatrix}$$

The problem admits the following matrix form:

$$\mathcal{A}\,\mathcal{U} = \mathcal{B}$$

The matrix and right-hand side of the system are assembled by concatenation:

```
csr<Float> A = {{  a.uu,     b.u},
                {trans(b.u), 0 }};
vec<Float> B =  {  lh.u,      0 };
```

where `csr` and `vec` are respectively the matrix and vector classes. The `csr` is the abbreviation of *compressed sparse row*, a sparse matrix compression standard format. Notice that the matrix $\mathcal{A}$ is symmetric and non-singular, but indefinite : it admits eigenvalues that are either strictly positive or strictly negative. While the Choleski factorization is not possible, its variant the LDL^T one is performed, thanks to the `ldlt` function:

```
solver sa = ldlt(A);
```

Then, the `uh.u` vector is extracted from the `U` one:

```
uh.u = U [range(0,uh.u.size())];
```

The extraction of `lambda` from `U` is more technical in a distributed environment. In a sequential one, since it is stored after the `uh.u` values, it could be simply written as:

```
Float lambda = U [uh.u.size()];
```

In a distributed environment, `lambda` is stored in `U` on the last processor, identified by `U.comm().size()-1`. Here `U.comm()` denotes the `communicator`, from the `boost::mpi` library and `U.comm().size()` is the number of processors in use, e.g. as specified by the `mpirun` command. On this last processor, the array `U` has size equal to `uh.u.size()+1` and `lambda` is stored in `U[uh.u.size()]`. On the others processors, the array `U` has size equal to `uh.u.size()` and lambda is not available. The following statement extract `lambda` on the last processor and set it to zero on the others:

```
Float lambda = (U.size() == uh.u.size()+1) ? U [uh.u.size()] : 0;
```

Then, the value of `lambda` is broadcasted on the others processors:

```
mpi::broadcast (U.comm(), lambda, U.comm().size() - 1);
```

The preprocessing guards `#idef...#endif` assure that this example compile also when the library is not installed with the MPI support. Finally, the statement

```
dout << catchmark("u") << uh
     << catchmark("lambda") << lambda << endl;
```

writes the solution (u_h, λ). The `catchmark` function writes marks together with the solution in the output stream. These marks are suitable for a future reading with the same format, as:

```
din  >> catchmark("u") >> uh
     >> catchmark("lambda") >> lambda;
```

This is useful for post-treatment, visualization and error analysis.

2.4.1 How to run the program

As usual, enter:

```
make neumann-laplace
mkgeo_grid -t 10 > square.geo
./neumann-laplace square P1 | field -
```

Chapter 3

Non-constant coefficients and multi-regions

This chapter is related to the so-called transmission problem. We introduce some new concepts: problems with non-constant coefficients, regions in the mesh, weighted forms and discontinuous approximations.

Formulation

Let us consider a diffusion problem with a non-constant diffusion coefficient η in a domain bounded $\Omega \subset \mathbb{R}^d$, $d = 1, 2, 3$:
(P)*: find* u *defined in* Ω *such that:*

$$
\begin{aligned}
-\mathrm{div}(\eta \boldsymbol{\nabla} u) &= f \text{ in } \Omega && (3.1)\\
u &= 0 \text{ on } \Gamma_{\text{left}} \cup \Gamma_{\text{right}} && (3.2)\\
\frac{\partial u}{\partial n} &= 0 \text{ on } \Gamma_{\text{top}} \cup \Gamma_{\text{bottom}} \text{ when } d \geqslant 2 && (3.3)\\
\frac{\partial u}{\partial n} &= 0 \text{ on } \Gamma_{\text{front}} \cup \Gamma_{\text{back}} \text{ when } d = 3 && (3.4)
\end{aligned}
$$

where f is a given source term.

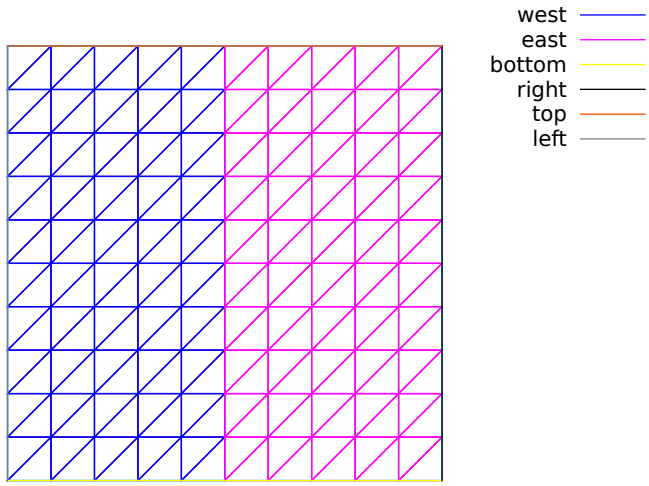


Figure 3.1: Transmission problem: the domain Ω partition: ($\Omega_{\rm west}$ and $\Omega_{\rm east}$).

We consider here the important special case when η is piecewise constant:

$$\eta(x) = \begin{cases} \varepsilon & \text{when } x \in \Omega_{\rm west} \\ 1 & \text{when } x \in \Omega_{\rm east} \end{cases}$$

where $(\Omega_{\rm west}, \Omega_{\rm east})$ is a partition of Ω in two parts (see Fig. 3.1). This is the so-called **transmission** problem: the solution and the flux are continuous on the interface $\Gamma_0 = \partial\Omega_{\rm east} \cap \partial\Omega_{\rm west}$ between the two domains where the problem reduce to a constant diffusion one:

$$\begin{aligned} u_{\Omega_{\rm west}} &= u_{\Omega_{\rm east}} \text{ on } \Gamma_0 \\ \varepsilon \frac{\partial u_{/\Omega_{\rm west}}}{\partial n} &= \frac{\partial u_{\Omega_{\rm east}}}{\partial n} \text{ on } \Gamma_0 \end{aligned}$$

It expresses the transmission of the quantity u and its flux across the interface Γ_0 between two regions that have different diffusion properties: Notice that the more classical problem, with constant diffusion η on Ω is obtained by simply choosing when $\varepsilon = 1$.

The variational formulation of this problem expresses:

(VF): find $u \in V$ such that:

$$a(u,v) = l(v), \ \forall v \in V$$

where the bilinear forms $a(.,.)$ and the linear one $l(.)$ are defined by

$$\begin{aligned} a(u,v) &= \int_\Omega \eta \, \nabla u . \nabla v \, dx, \quad \forall u, v \in H^1(\Omega) \\ l(v) &= \int_\Omega f \, v \, dx, \quad \forall v \in L^2(\Omega) \\ V &= \{ v \in H^1(\Omega); \ v = 0 \text{ on } \Gamma_{\rm left} \cup \Gamma_{\rm right} \} \end{aligned}$$

The bilinear form $a(.,.)$ defines a scalar product in V and is related to the *energy* form. This form is associated to the $-\mathrm{div}\, \eta \boldsymbol{\nabla}$ operator.

The approximation of this problem could performed by a standard Lagrange P_k continuous approximation.

Example file 3.1: `transmission.cc`

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
int main(int argc, char**argv) {
  environment rheolef (argc, argv);
  const Float epsilon = 0.01;
  geo omega (argv[1]);
  space Xh (omega, argv[2]);
  Xh.block ("left");
  Xh.block ("right");
  string eta_approx = "P" + itos(Xh.degree()-1) + "d";
  space Qh (omega, eta_approx);
  field eta_h (Qh);
  eta_h ["east"] = 1;
  eta_h ["west"] = epsilon;
  trial u (Xh); test v (Xh);
  form  a  = integrate (eta_h*dot(grad(u),grad(v)));
  field lh = integrate (v);
  field uh (Xh);
  uh["left"] = uh["right"] = 0;
  solver sa (a.uu());
  uh.set_u() = sa.solve (lh.u() - a.ub()*uh.b());
  dout << catchmark("epsilon") << epsilon << endl
       << catchmark("u")       << uh;
}
```

Comments

This file is quite similar to those studied in the first chapters of this book. Let us comment only the changes. The Dirichlet boundary condition applies no more on the whole boundary $\partial\Omega$ but on two parts Γ_{left} and Γ_{right}. On the other boundary parts, an homogeneous Neumann boundary condition is used: since these conditions does not produce any additional terms in the variational formulation, there are also nothing to write in the `C++` code for these boundaries. We choose $f = 1$: this leads to a convenient test-problem, since the exact solution is known when $\Omega =]0,1[^d$:

$$u(x) = \begin{cases} \dfrac{x_0}{2\varepsilon}\left(\dfrac{1+3\varepsilon}{2(1+\varepsilon)} - x_0\right) & \text{when } x_0 < 1/2 \\ \dfrac{1-x_0}{2}\left(x_0 + \dfrac{1-\varepsilon}{2(1+\varepsilon)}\right) & \text{otherwise} \end{cases}$$

The field η belongs to a discontinuous finite element P_{k-1} space denoted by Q_h:

```
  string eta_approx = "P" + itos(Xh.degree()-1) + "d";
  space Qh (omega, eta_approx);
  field eta (Qh);
```

For instance, when `argv[2]` contains `"P2"`, i.e. $k = 2$, then the string `eta_approx` takes value `"P1d"`. Then η is initialized by:

```
  eta["east"]  = 1;
  eta["weast"] = epsilon;
```

The energy form a is then constructed with η as additional parameter that acts as a integration *weight*:

```
  form a = integrate (eta_h*dot(grad(u),grad(v)));
```

Such forms with a additional *weight* function are called *weighted forms* in **Rheolef**.

How to run the program ?

Build the program as usual: `make transmission`. Then, creates a one-dimensional geometry with two regions:

```
mkgeo_grid -e 100 -region > line.geo
geo line.geo
```

The trivial mesh generator `mkgeo_grid`, defines two regions `east` and `west` when used with the `-region` option. This correspond to the situation:

$$\Omega = [0,1]^d, \quad \Omega_{\text{west}} = \Omega \cap \{x_0 < 1/2\} \text{ and } \Omega_{\text{east}} = \Omega \cap \{x_0 > 1/2\}.$$

In order to avoid mistakes with the `C++` style indexes, we denote by $(x_0, \ldots, x_{d-1})$ the Cartesian coordinate system in $\mathbb{R}^d$.

Finally, run the program and look at the solution:

```
make transmission
./transmission line.geo P1 > line.field
field line.field
```

Since the exact solution is a piecewise second order polynomial and the change in the diffusion coefficient value fits the element boundaries, we obtain the exact solution for all the degrees of freedom of any P_k approximation, $k \geqslant 1$, as shown on Fig. 3.2 when $k = 1$. Moreover, when $k \geqslant 2$ then $u_h = u$ since X_h contains the exact solution u.

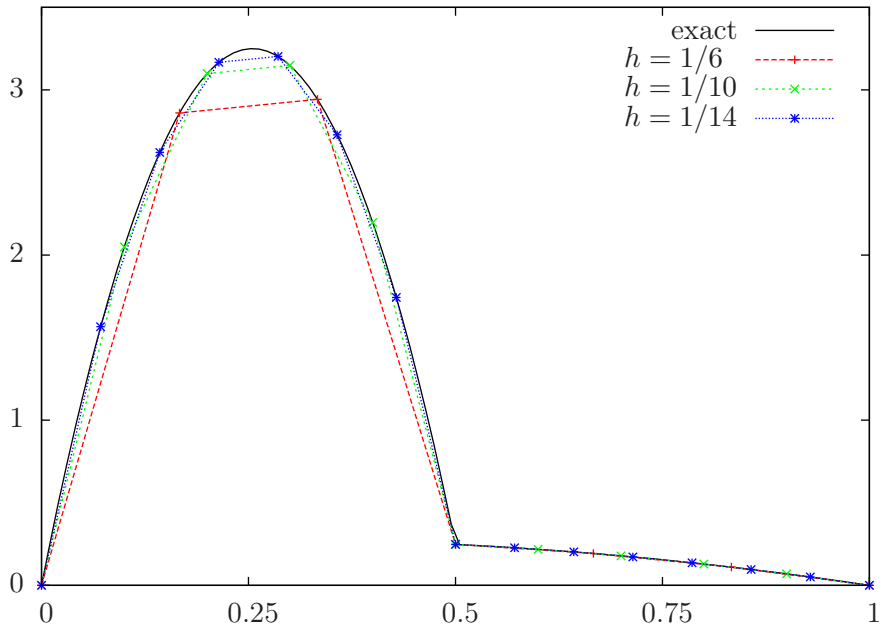


Figure 3.2: Transmission problem: $u_h = \pi_h(u)$ ($\varepsilon = 10^{-2}$, $d = 1$, P_1 approximation).

The two dimensional case corresponds to the commands:

```
mkgeo_grid -t 10 -region > square.geo
geo square.geo
./transmission square.geo P1 > square.field
field square.field -elevation
```

while the tridimensional to

```
mkgeo_grid -T 10 -region > cube.geo
./transmission cube.geo P1 > cube.mfield
field cube.field
```

As for all the others examples, you can replace `P1` by higher-order approximations, change elements shapes, such as `q`, `H` or `P`, and run distributed computations computations with `mpirun`.

Part II

Fluids and solids computations

Chapter 4

The linear elasticity and the Stokes problems

4.1 The linear elasticity problem

Formulation

The total Cauchy stress tensor expresses:

$$\sigma(\mathbf{u}) = \lambda \operatorname{div}(\mathbf{u}).I + 2\mu D(\mathbf{u}) \tag{4.1}$$

where λ and μ are the Lamé coefficients. Here, $D(\mathbf{u})$ denotes the symmetric part of the gradient operator and div is the divergence operator. Let us consider the elasticity problem for the *embankment*, in $\Omega =]0,1[^d$, $d = 2, 3$. The problem writes:

(P): *find* $\mathbf{u} = (u_0, \ldots, u_{d-1})$, *defined in* Ω, *such that:*

$$\begin{array}{rcl} -\operatorname{\mathbf{div}} \sigma(\mathbf{u}) & = & \mathbf{f} \text{ in } \Omega, \\ \dfrac{\partial \mathbf{u}}{\partial \mathbf{n}} & = & 0 \text{ on } \Gamma_{\text{top}} \cup \Gamma_{\text{right}} \\ \mathbf{u} & = & 0 \text{ on } \Gamma_{\text{left}} \cup \Gamma_{\text{bottom}}, \\ \mathbf{u} & = & 0 \text{ on } \Gamma_{\text{front}} \cup \Gamma_{\text{back}}, \text{ when } d = 3 \end{array} \tag{4.2}$$

where $\mathbf{f} = (0, -1)$ when $d = 2$ and $\mathbf{f} = (0, 0, -1)$ when $d = 3$. The Lamé coefficients are assumed to satisfy $\mu > 0$ and $\lambda + \mu > 0$. Since the problem is linear, we can suppose that $\mu = 1$ without any loss of generality.

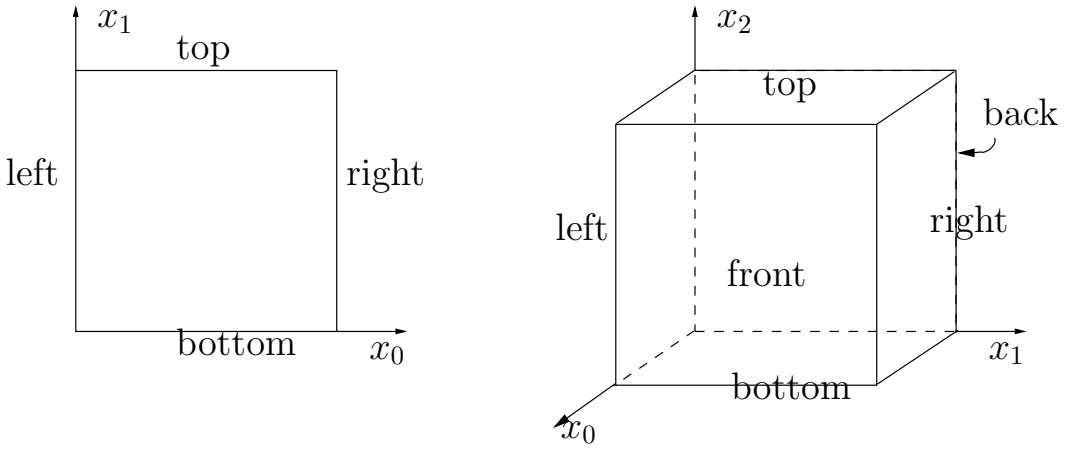


Figure 4.1: The boundary domains for the square and the cube.

recall that, in order to avoid mistakes with the `C++` style indexes, we denote by $(x_0, \ldots, x_{d-1})$ the Cartesian coordinate system in
mathbbR^d.

For $d = 2$ we define the boundaries:

$$\begin{array}{rclrcl} \Gamma_{\text{left}} & = & \{0\}\times]0,1[, & \Gamma_{\text{right}} & = & \{1\}\times]0,1[\\ \Gamma_{\text{bottom}} & = &]0,1[\times\{0\}, & \Gamma_{\text{top}} & = &]0,1[\times\{1\} \end{array}$$

and for $d = 3$:

$$\begin{array}{rclrcl} \Gamma_{\text{back}} & = & \{0\}\times]0,1[^2, & \Gamma_{\text{front}} & = & \{1\}\times]0,1[^2 \\ \Gamma_{\text{left}} & = &]0,1[\times\{0\}\times]0,1[, & \Gamma_{\text{right}} & = &]0,1[\times\{1\}\times]0,1[\\ \Gamma_{\text{bottom}} & = &]0,1[^2\times\{0\}, & \Gamma_{\text{top}} & = &]0,1[^2\times\{1\} \end{array}$$

These boundaries are represented on Fig. 4.1.

The variational formulation of this problem expresses:

(VF): *find* $\mathbf{u} \in \mathbf{V}$ *such that:*

$$a(\mathbf{u}, \mathbf{v}) = l(\mathbf{v}),\ \forall \mathbf{v} \in \mathbf{V}, \tag{4.3}$$

where

$$\begin{array}{rcl} a(\mathbf{u}, \mathbf{v}) & = & \displaystyle\int_\Omega (\lambda \operatorname{div}\mathbf{u} \operatorname{div}\mathbf{v} + 2D(\mathbf{u}) : D(\mathbf{v}))\ dx, \\ l(\mathbf{v}) & = & \displaystyle\int_\Omega \mathbf{f}.\mathbf{v}\ dx, \\ \mathbf{V} & = & \{\mathbf{v} \in (H^1(\Omega))^2;\ \mathbf{v} = 0 \text{ on } \Gamma_{\text{left}} \cup \Gamma_{\text{bottom}}\},\ \text{when } d = 2 \\ \mathbf{V} & = & \{\mathbf{v} \in (H^1(\Omega))^3;\ \mathbf{v} = 0 \text{ on } \Gamma_{\text{left}} \cup \Gamma_{\text{bottom}} \cup \Gamma_{\text{right}} \cup \Gamma_{\text{back}}\},\ \text{when } d = 3 \end{array}$$

Approximation

We introduce a mesh $\mathcal{T}_h$ of Ω and for $k \geqslant 1$, the following finite dimensional spaces:

$$\begin{array}{rcl} \mathbf{X}_h & = & \{\mathbf{v}_h \in (H^1(\Omega))^d;\ \mathbf{v}_{h/K} \in (P_k)^d,\ \forall K \in \mathcal{T}_h\}, \\ \mathbf{V}_h & = & \mathbf{X}_h \cap \mathbf{V} \end{array}$$

The approximate problem writes:

$(VF)_h$: *find* $\mathbf{u}_h \in \mathbf{V}_h$ *such that:*

$$a(\mathbf{u}_h, \mathbf{v}_h) = l(\mathbf{v}_h),\ \forall \mathbf{v}_h \in \mathbf{V}_h$$

Example file 4.1: `embankment.cc`

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
#include "embankment.icc"
int main(int argc, char**argv) {
  environment rheolef(argc, argv);
  geo  omega (argv[1]);
  space Xh = embankment_space (omega, argv[2]);
  Float lambda = (argc > 3) ? atof(argv[3]) : 1;
  size_t d = omega.dimension();
  point f (0,0,0);
  f[d-1] = -1;
  trial u (Xh); test v (Xh);
  form  a  = integrate (lambda*div(u)*div(v) + 2*ddot(D(u),D(v)));
  field lh = integrate (dot(f,v));
  solver sa (a.uu());
  field uh (Xh, 0);
  uh.set_u() = sa.solve (lh.u() - a.ub()*uh.b());
  dout << catchmark("inv_lambda") << 1/lambda << endl
       << catchmark("u")          << uh;
}
```

Example file 4.2: `embankment.icc`

```
space embankment_space (const geo& omega, string approx) {
  space Xh (omega, approx, "vector");
  Xh.block("left");
  if (omega.dimension() >= 2)
    Xh.block("bottom");
  if (omega.dimension() == 3) {
    Xh.block("right");
    Xh.block("back");
  }
  return Xh;
}
```

Comments

The space is defined in a separate file '`embankment.icc`', since it will be reused in others examples along this chapter:

```
  space Vh (omega, "P2", "vector");
```

Note here the multi-component specification `"vector"` as a supplementary argument to the `space` constructor. The boundary condition contain a special cases for bi- and tridimensional cases. The right-hand-side $\mathbf{f}_h$ represents the dimensionless gravity forces, oriented on the vertical axis: the last component of $\mathbf{f}_h$ is set to -1 as:

```
    fh [d-1] = -1;
```

The code for the bilinear form $a(.,.)$ and the linear one $l(.)$ are closed to their mathematical definitions:

```
  form  a  = integrate (lambda*div(u)*div(v) + 2*ddot(D(u),D(v)));
  field lh = integrate (dot(f,v));
```

Finally, the $1/\lambda$ parameter and the multi-field result are printed, using mark labels, thanks to the `catchmark` stream manipulator. Labels are convenient for post-processing purpose, as we will see in the next paragraph.

How to run the program

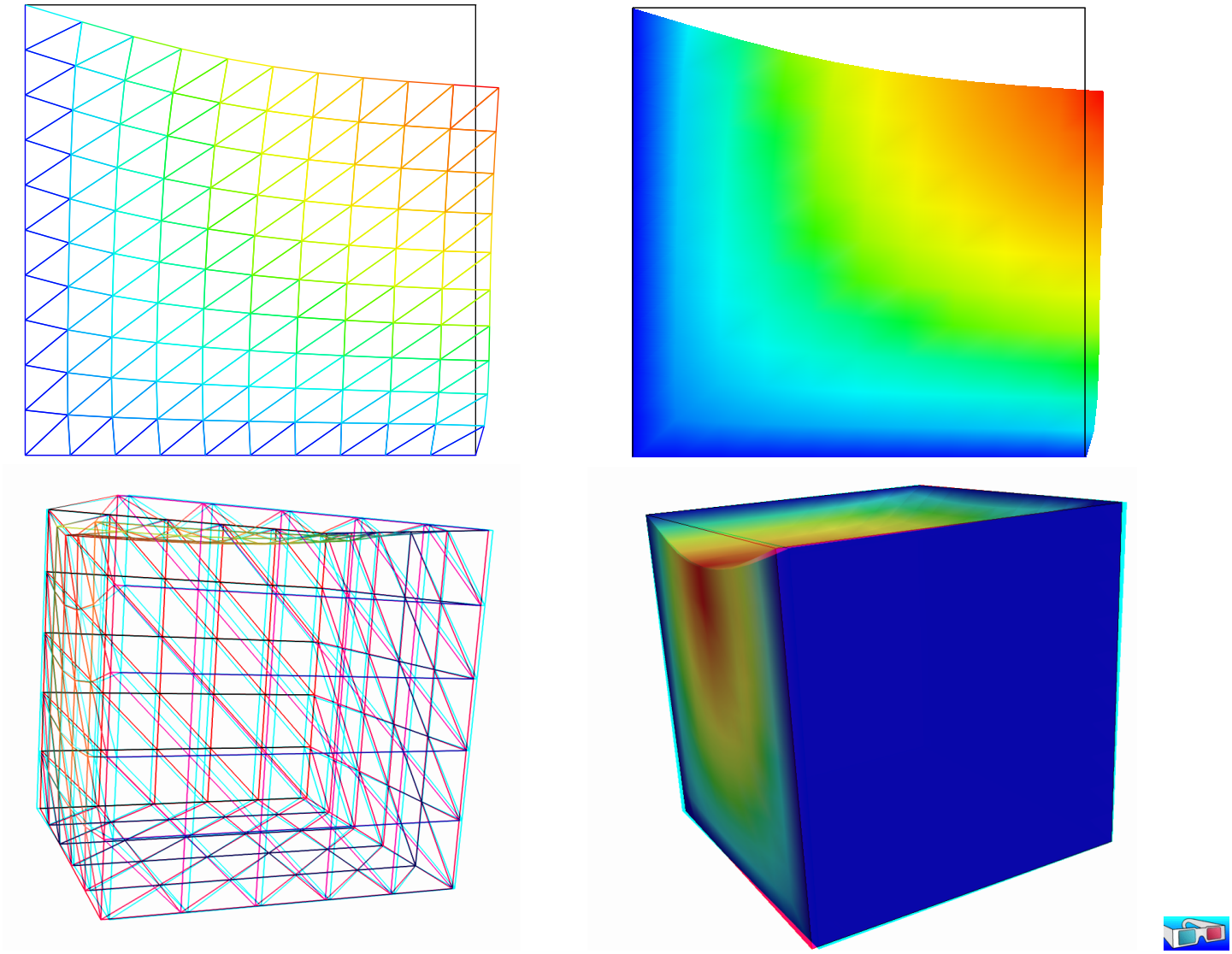


Figure 4.2: The linear elasticity for $\lambda = 1$ and $d = 2$ and $d = 3$: both wireframe and filled surfaces ; stereoscopic anaglyph mode for 3D solutions.

Compile the program as usual (see page 18):

```
make embankment
```

and enter the commands:

```
mkgeo_grid -t 10 > square.geo
geo square.geo
```

The triangular mesh has four boundary domains, named `left`, `right`, `top` and `bottom`. Then, enter:

```
./embankment square.geo P1 > square-P1.field
```

The previous command solves the problem for the corresponding mesh and writes the multi-component solution in the '`.field`' file format. Run the deformation vector field visualization using the default `gnuplot` render:

```
field square-P1.field
field square-P1.field -nofill
```

Note the graphic options usage ; the unix manual for the `field` command is available as:

```
man field
```

The view is shown on Fig. 4.2. A specific field component can be also selected for a scalar visualization:

```
field -comp 0 square-P1.field
field -comp 1 square-P1.field
```

Next, perform a P_2 approximation of the solution:

```
./embankment square.geo P2 > square-P2.field
field square-P2.field -mayavi -nofill
```

Finally, let us consider the three dimensional case

```
mkgeo_grid -T 10 > cube.geo
./embankment cube.geo P1 > cube-P1.field
field cube-P1.field -stereo
field cube-P1.field -stereo -fill
```

The two last commands show the solution in 3D stereoscopic anaglyph mode. The graphic is represented on Fig. 4.2. The P_2 approximation writes:

```
./embankment cube.geo P2 > cube-P2.field
field cube-P2.field
```

4.2 Computing the stress tensor

Formulation and approximation

The following code computes the total Cauchy stress tensor, reading the Lamé coefficient λ and the deformation field $\mathbf{u}_h$ from a '`.field`' file. Let us introduce:

$$T_h = \{\tau_h \in (L^2(\Omega))^{d\times d};\ \tau_h = \tau_h^T \text{ and } \tau_{h;ij/K} \in P_{k-1},\ \forall K \in \mathcal{T}_h,\ 1 \leqslant i,j \leqslant d\}$$

This computation expresses:

find σ_h such that:

$$m(\sigma_h,\tau) = b(\tau,\mathbf{u}_h), \forall \tau \in T_h$$

where

$$\begin{aligned} m(\sigma,\tau) &= \int_\Omega \sigma : \tau\, dx, \\ b(\tau,\mathbf{u}) &= \int_\Omega (2D(\mathbf{u}) : \tau\, dx + \lambda \mathrm{div}(\mathbf{u})\,\mathrm{tr}(\tau))\ dx, \end{aligned}$$

where $\mathrm{tr}(\tau) = \sum_{i=1}^d \tau_{ii}$ is the trace of the tensor τ.

Example file 4.3: `stress.cc`

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
int main(int argc, char** argv) {
  environment rheolef (argc,argv);
  Float inv_lambda;
  field uh;
  din >> catchmark("inv_lambda") >> inv_lambda
      >> catchmark("u")          >> uh;
  const geo& omega = uh.get_geo();
  const space& Xh  = uh.get_space();
  string grad_approx = "P" + itos(Xh.degree()-1) + "d";
  space Th  (omega, grad_approx, "tensor");
  size_t d = omega.dimension();
  tensor I = tensor::eye (d);
  field sigma_h = (inv_lambda == 0) ?
      interpolate (Th, 2*D(uh)) :
      interpolate (Th, 2*D(uh) + (1/inv_lambda)*div(uh)*I);
  dout << catchmark("s") << sigma_h;
}
```

Comments

In order to our code `stress.cc` to apply also for the forthcomming incompressible case $\lambda = +\infty$, the Lamé coefficient is introduced as $1/\lambda$. Its value is zero in the incompressible case. By this way, the previous code applies for any deformation field, and is not restricted to our *embankment* problem. The stress tensor is obtained by a direct interpolation of the u_h first derivatives. As u_h is continuous and piecewise polynomial P_k, its derivatives are also piecewise polynomials with degree $k-1$, but *discontinuous* at inter-element boundaries : this approximation is denoted as $P_{k-1,d}$. Thus, the stress tensor belongs to the space T_h with the $P_{k-1,d}$ element.

How to run the program

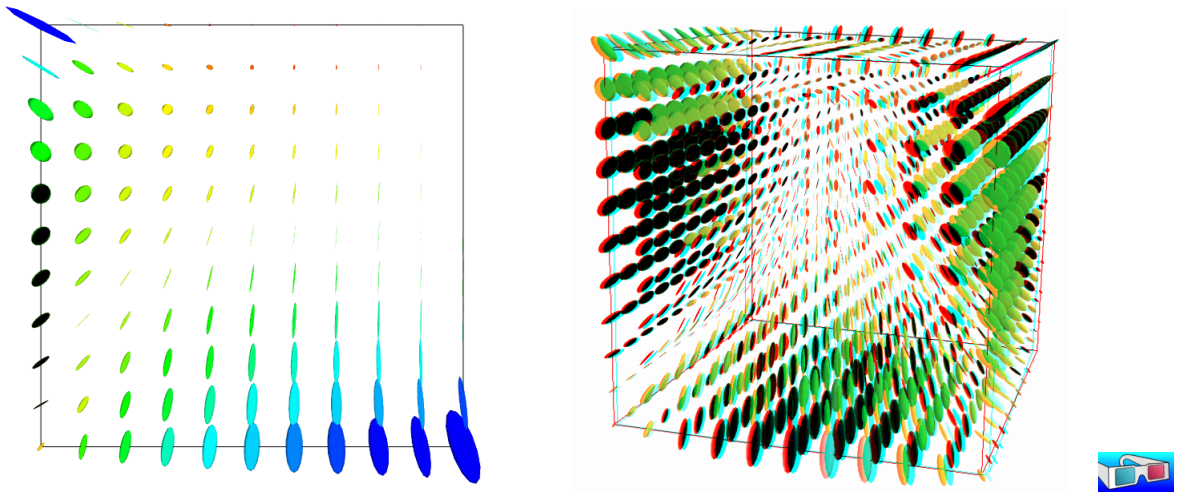


Figure 4.3: The stress tensor visualization (linear elasticity $\lambda = 1$).

First, compile the program:

```
make stress
```

The visualization for the stress tensor as ellipses writes:

```
./stress < square-P1.field > square-stress-P1.field
./stress < square-P2.field > square-stress-P2.field
field square-stress-P1.field -proj -mayavi
```

Recall that the stress, as a derivative of the deformation, is P0 (resp. P1d) and discontinuous when the deformation is P1 (resp. P2) and continuous. The approximate stress tensor field is projected on a continuous piecewise linear space, using the `-proj` option. Conversely, the 3D visualization bases on ellipsoids:

```
./stress < cube-P1.field > cube-stress-P1.field
field cube-stress-P1.field -proj -mayavi -stereo
```

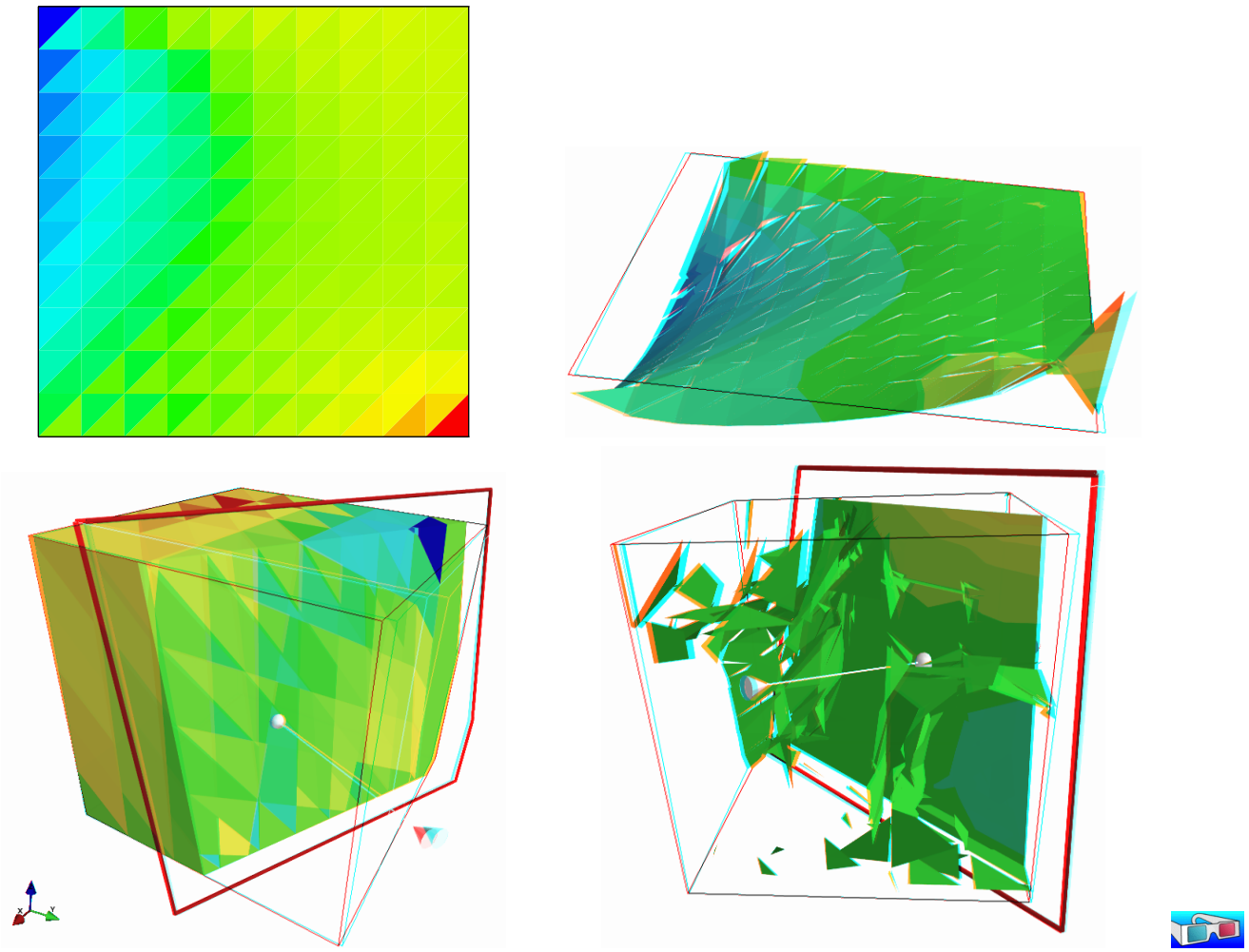


Figure 4.4: The σ_{01} stress component (linear elasticity $\lambda = 1$): $d = 2$ (top) and $d = 3$ (bottom) ; P_0 (left) and P_1 discontinuous approximation (right).

You can observe a discontinuous constant or piecewise linear representation of the approximate stress component σ_{01} (see Fig. 4.4):

```
field square-stress-P1.field -comp 01
```

```
field square-stress-P2.field -comp 01 -elevation
field square-stress-P2.field -comp 01 -elevation -stereo
```

Notice that the `-stereo` implies the `-mayavi` one, as this feature is not available with others visualization systems. The approximate stress field can be also projected on a continuous piecewise space:

```
field square-stress-P2.field -comp 01 -elevation -proj
```

The tridimensional case writes simply (see Fig. 4.4):

```
./stress < cube-P1.field > cube-stress-P1.field
./stress < cube-P2.field > cube-stress-P2.field
field cube-stress-P1.field -comp 01 -stereo
field cube-stress-P2.field -comp 01 -stereo -iso
```

and also the P1-projected versions write:

```
field cube-stress-P1.field -comp 01 -stereo -proj -iso
field cube-stress-P2.field -comp 01 -stereo -proj -iso
```

These operations can be repeated for each σ_{ij} components and for both `P1` and `P2` approximation of the deformation field.

4.3 Mesh adaptation

The main principle of the auto-adaptive mesh writes [6, 10, 21, 39, 50]:

```
cin >> omega;
uh = solve(omega);
for (unsigned int i = 0; i < n; i++) {
    ch = criterion(uh);
    omega = adapt(ch);
    uh = solve(omega);
}
```

The initial mesh is used to compute a first solution. The adaptive loop compute an *adaptive criterion*, denoted by `ch`, that depends upon the problem under consideration and the polynomial approximation used. Then, a new mesh is generated, based on this criterion. A second solution on an adapted mesh can be constructed. The adaptation loop converges generally in roughly 5 to 20 iterations.

Let us apply this principle to the elasticity problem.

Example file 4.4: `embankment_adapt.cc`

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
#include "elasticity_solve.icc"
#include "elasticity_criterion.icc"
#include "embankment.icc"
int main(int argc, char**argv) {
  environment rheolef (argc, argv);
  const Float lambda = 1;
  geo  omega (argv[1]);
  adapt_option_type options;
  string approx  = (argc > 2) ? argv[2] : "P1";
  options.err    = (argc > 3) ? atof(argv[3]) : 5e-3;
  size_t n_adapt = (argc > 4) ? atoi(argv[4]) : 5;
  options.hmin   = 0.004;
  for (size_t i = 0; true; i++) {
    space Xh = embankment_space (omega, approx);
    field uh = elasticity_solve (Xh, lambda);
    odiststream of (omega.name(), "field");
    of << catchmark("lambda")  << lambda << endl
       << catchmark("u")       << uh;
    if (i == n_adapt) break;
    field ch = elasticity_criterion (lambda,uh);
    omega = adapt(ch, options);
    odiststream og (omega.name(), "geo");
    og << omega;
  }
}
```

Example file 4.5: `elasticity_solve.icc`

```
field elasticity_solve (const space& Xh, Float lambda) {
  size_t d = Xh.get_geo().dimension();
  point f (0,0,0);
  f[d-1] = -1;
  trial u (Xh); test v (Xh);
  field lh = integrate (dot(f,v));
  form  a  = integrate (lambda*div(u)*div(v) + 2*ddot(D(u),D(v)));
  solver sa (a.uu());
  field uh (Xh, 0);
  uh.set_u() = sa.solve (lh.u() - a.ub()*uh.b());
  return uh;
}
```

Example file 4.6: `elasticity_criterion.icc`

```
field elasticity_criterion (Float lambda, const field& uh) {
  string grad_approx = (uh.get_approx() == "P2") ? "P1d" : "P0";
  space Xh  (uh.get_geo(), grad_approx);
  if (grad_approx == "P0") return interpolate (Xh, norm(uh));
  space T0h  (uh.get_geo(), grad_approx);
  size_t d = uh.get_geo().dimension();
  tensor I = tensor::eye (d);
  return interpolate (T0h, sqrt(2*norm2(D(uh)) + lambda*sqr(div(uh))));
}
```

Comments

The criterion is here:

$$c_h = \begin{cases} |\mathbf{u}_h| & \text{when using } P_1 \\ (\sigma(\mathbf{u}_h) : D(\mathbf{u}_h))^{1/2} & \text{when using } P_2 \end{cases}$$

The `elasticity_criterion` function compute it as

```
  return interpolate (Xh, norm(uh));
```

when using P_1, and as

```
return interpolate (T0h, sqrt(2*norm2(D(uh)) + lambda*sqr(div(uh))));
```

when using P_2. The `sqr` function returns the square of a scalar. Conversely, the `norm2` function returns the square of the norm. In the min programm, the result of the `elasticity_criterion` function is send to the `adapt` function:

```
field ch = elasticity_criterion (lambda, uh);
omega = adapt (ch, options);
```

The `adapt_option_type` declaration is used by **Rheolef** to send options to the mesh generator. The `err` parameter controls the error via the edge length of the mesh: the smaller it is, the smaller the edges of the mesh are. In our example, is set by default to one. Conversely, the `hmin` parameter controls minimal edge length.

How to run the program

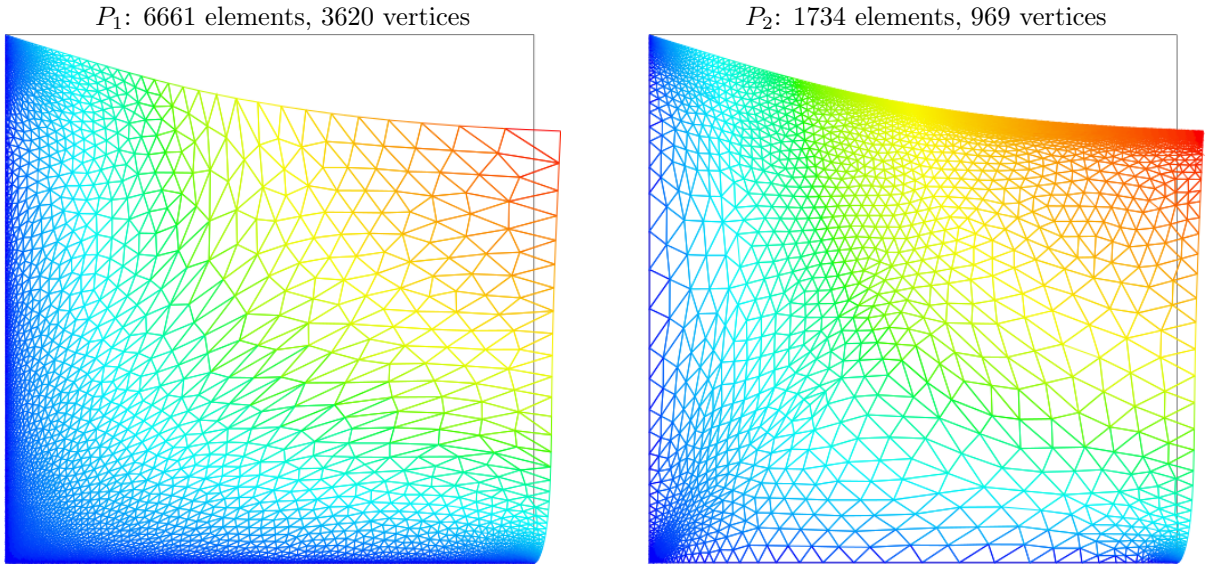


Figure 4.5: Adapted meshes: the deformation visualization for P_1 and P_2 approximations.

The compilation command writes:

```
make embankment_adapt
```

The mesh loop adaptation is initiated from a `bamg` mesh (see also appendix B.1).

```
bamg -g square.bamgcad -o square.bamg
bamg2geo square.bamg square.dmn > square.geo
./embankment_adapt square P1 2e-2
```

The last command line argument is the target error. The code performs a loop of five mesh adaptations: the corresponding meshes are stored in files, from `square-1.geo.gz` to `square-5.geo.gz`, and the associated solutions in files, from `square-1.field.gz` to `square-5.field.gz`. The additional '`.gz`' suffix expresses that the files are compressed using `gzip`.

```
geo square-5.geo
field square-5.field -mayavi -nofill
```

Note that the '`.gz`' suffix is automatically assumed by the `geo` and the `field` commands.
For a piecewise quadratic approximation:

```
./embankment_adapt square P2 5e-3
```

Then, the visualization writes:

```
geo square-5.geo
field square-5.field -mayavi -nofill
```

A one-dimensional mesh adaptive procedure is also possible:

```
gmsh -1 line.mshcad -o line.msh
msh2geo line.msh > line.geo
geo line.geo
./embankment_adapt line P2
geo line-5.geo
field line-5.field -comp 0 -elevation
```

The three-dimensional extension of this mesh adaptive procedure is in development.

4.4 The Stokes problem

Formulation

Let us consider the Stokes problem for the driven cavity in $\Omega =]0,1[^d$, $d = 2, 3$. The problem writes:

(S) *find* $\mathbf{u} = (u_0, \ldots, u_{d-1})$ *and* p *defined in* Ω *such that:*

$$\begin{array}{rcrcl}
-\mathbf{div}(2D(\mathbf{u})) & + & \boldsymbol{\nabla} p & = & 0 \text{ in } \Omega, \\
-\operatorname{div}\mathbf{u} & & & = & 0 \text{ in } \Omega, \\
\mathbf{u} & & & = & (1,0) \text{ on } \Gamma_{\text{top}}, \\
\mathbf{u} & & & = & 0 \text{ on } \Gamma_{\text{left}} \cup \Gamma_{\text{right}} \cup \Gamma_{\text{bottom}}, \\
\dfrac{\partial u_0}{\partial \mathbf{n}} & = & \dfrac{\partial u_1}{\partial \mathbf{n}} & = & u_2 = 0 \text{ on } \Gamma_{\text{back}} \cup \Gamma_{\text{front}} \quad \text{when } d = 3,
\end{array}$$

where $D(\mathbf{u}) = (\boldsymbol{\nabla}\mathbf{u} + \boldsymbol{\nabla}\mathbf{u}^T)/2$. The boundaries are represented on Fig. 4.1, page 48.
The variational formulation of this problem expresses:

(VFS) *find* $\mathbf{u} \in \mathbf{V}(1)$ *and* $p \in L^2_0(\Omega)$ *such that:*

$$\begin{array}{rcrcl}
a(\mathbf{u},\mathbf{v}) & + & b(\mathbf{v},p) & = & 0, \ \forall \mathbf{v} \in \mathbf{V}(0), \\
b(\mathbf{u},q) & & & = & 0, \ \forall q \in L^2_0(\Omega),
\end{array}$$

where

$$\begin{aligned}
a(\mathbf{u},\mathbf{v}) &= \int_\Omega 2D(\mathbf{u}) : D(\mathbf{v})\, dx, \\
b(\mathbf{v},q) &= -\int_\Omega \operatorname{div}(\mathbf{v})\, q\, dx. \\
\mathbf{V}(\alpha) &= \{\mathbf{v} \in (H^1(\Omega))^2;\ \mathbf{v} = 0 \text{ on } \Gamma_{\text{left}} \cup \Gamma_{\text{right}} \cup \Gamma_{\text{bottom}} \text{ and } \mathbf{v} = (\alpha,0) \text{ on } \Gamma_{\text{top}}\}, \quad \text{when } d = 2, \\
\mathbf{V}(\alpha) &= \{\mathbf{v} \in (H^1(\Omega))^3;\ \mathbf{v} = 0 \text{ on } \Gamma_{\text{left}} \cup \Gamma_{\text{right}} \cup \Gamma_{\text{bottom}}, \\
&\qquad \mathbf{v} = (\alpha,0,0) \text{ on } \Gamma_{\text{top}} \text{ and } v_2 = 0 \text{ on } \Gamma_{\text{back}} \cup \Gamma_{\text{front}}\}, \quad \text{when } d = 3, \\
L^2_0(\Omega) &= \{q \in L^2(\Omega);\ \int_\Omega q\, dx = 0\}.
\end{aligned}$$

Approximation

The Taylor-Hood [22] finite element approximation of the Stokes problem is considered. We introduce a mesh $\mathcal{T}_h$ of Ω and the following finite dimensional spaces:

$$\begin{aligned} \mathbf{X}_h &= \{\mathbf{v} \in (H^1(\Omega))^d;\ \mathbf{v}_{/K} \in (P_2)^d,\ \forall K \in \mathcal{T}_h\}, \\ \mathbf{V}_h(\alpha) &= \mathbf{X}_h \cap \mathbf{V}(\alpha), \\ Q_h &= \{q \in L^2(\Omega)) \cap C^0(\bar{\Omega});\ q_{/K} \in P_1,\ \forall K \in \mathcal{T}_h\}, \end{aligned}$$

The approximate problem writes:

$(VFS)_h$ *find* $\mathbf{u}_h \in \mathbf{V}_h(1)$ *and* $p \in Q_h$ *such that:*

$$\begin{array}{lcll} a(\mathbf{u}_h, \mathbf{v}) + b(\mathbf{v}, p_h) &=& 0, & \forall \mathbf{v} \in \mathbf{V}_h(0), \\ b(\mathbf{u}_h, q) &=& 0, & \forall q \in Q_h. \end{array} \tag{4.4}$$

Example file 4.7: `cavity.icc`

```
space cavity_space (const geo& omega_h, std::string approx) {
    space Xh (omega_h, approx, "vector");
    Xh.block("top");  Xh.block("bottom");
    if (omega_h.dimension() == 3) {
      Xh.block("back"); Xh.block("front");
      Xh[1].block("left");  Xh[1].block("right");
    } else {
      Xh.block("left"); Xh.block("right");
    }
    return Xh;
}
field cavity_field (const space& Xh, Float alpha) {
    field uh (Xh, 0.);
    uh[0]["top"] = alpha;
    return uh;
}
```

Example file 4.8: `stokes_cavity.cc`

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
#include "cavity.icc"
int main(int argc, char**argv) {
  environment rheolef (argc, argv);
  geo  omega (argv[1]);
  space Xh = cavity_space (omega, "P2");
  space Qh (omega, "P1");
  trial u (Xh), p (Qh); test v (Xh), q (Qh);
  form a  = integrate (2*ddot(D(u),D(v)));
  form b  = integrate (-div(u)*q);
  form mp = integrate (p*q);
  field uh = cavity_field (Xh, 1);
  field ph (Qh, 0.);
  solver_abtb stokes (a.uu(), b.uu(), mp.uu());
  stokes.solve (-(a.ub()*uh.b()), -(b.ub()*uh.b()),
                uh.set_u(),       ph.set_u());
  dout << catchmark("u")  << uh
       << catchmark("p")  << ph;
}
```

Comments

The spaces and boundary conditions and grouped in specific functions, defined in file '`cavity.icc`'. This file is suitable for a future re-usage. Next, forms are defined as usual, in file '`stokes_cavity.cc`'.

The problem admits the following matrix form:

$$\begin{pmatrix} \texttt{a.uu} & \texttt{trans(b.uu)} \\ \texttt{b.uu} & 0 \end{pmatrix} \begin{pmatrix} \texttt{uh.u} \\ \texttt{ph.u} \end{pmatrix} = \begin{pmatrix} -\texttt{a.ub} * \texttt{uh.b} \\ -\texttt{b.ub} * \texttt{uh.b} \end{pmatrix}$$

An initial value for the pressure field is provided:

```
field ph (Qh, 0);
```

The main Stokes solver call writes:

```
solver_abtb stokes (a.uu(), b.uu(), mp.uu());
stokes.solve (-(a.ub()*uh.b()), -(b.ub()*uh.b()),
              uh.set_u(),        ph.set_u());
```

For tridimensional geometries ($d = 3$), this system is solved by the preconditioned conjugate gradient algorithm. the preconditioner is here the mass matrix `mp.uu` for the pressure: as showed in [24], the number of iterations need by the conjugate gradient algorithm to reach a given precision is then independent of the mesh size. For more details, see the Rheolef reference manual related to mixed solvers, available e.g. via the unix command:

```
man solver_abtb
```

When $d = 2$, it is interesting to turn to direct methods and factorize the whole matrix of the linear system. As the pressure is defined up to a constant, the whole matrix is singular. By adding a Lagrange multiplier that impose a null average pressure value, the system becomes regular and the modified matrix can be inverted. Such a technique has already been presented in section 2.4 for the Neumann-Laplace problem. Finally, he choice between iterative and direct algorithm for the Stokes solver is automatically done, regarding the geometry dimension.

How to run the program

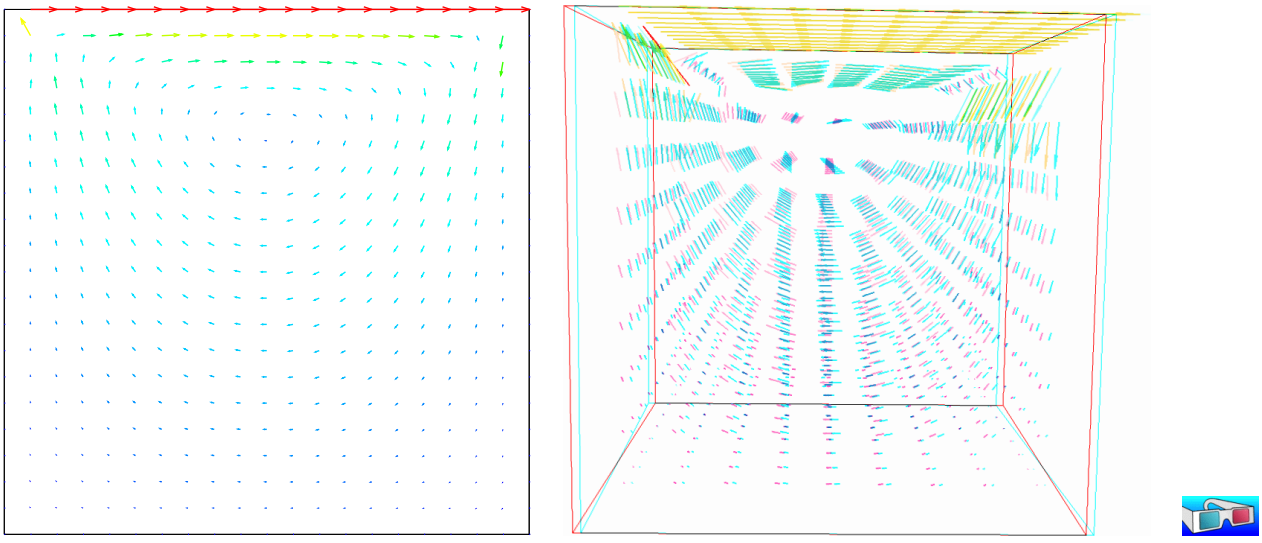


Figure 4.6: The velocity visualization for $d = 2$ and $d = 3$ with stereo anaglyph.

We assume that the previous code is contained in the file '`stokes_cavity.cc`'. Then, compile the program as usual (see page 18):

```
make stokes_cavity
```

and enter the commands:

```
mkgeo_grid -t 10 > square.geo
./stokes_cavity square > square.field
```

The previous command solves the problem for the corresponding mesh and writes the solution in a '`.field`' file. Run the velocity vector visualization :

```
field square.field -velocity
```

Run also some scalar visualizations:

```
field square.field -comp 0
field square.field -comp 1
field square.field -catchmark p
```

Note the `-catchmark` option to the `field` command: the file reader jumps to the label and then starts to read the selected field. Next, perform another computation on a finer mesh:

```
mkgeo_grid -t 20 > square-20.geo
./stokes_cavity square-20.geo > square-20.field
```

and observe the convergence.
Finally, let us consider the three dimensional case:

```
mkgeo_grid -T 5 > cube.geo
./stokes_cavity cube.geo > cube.field
```

and the corresponding visualization:

```
field cube.field -velocity
field cube.field -comp 0
field cube.field -comp 1
field cube.field -comp 2
field cube.field -catchmark p
```

4.5 Computing the vorticity

Formulation and approximation

When $d = 2$, we define [19, page 30] for any distributions ϕ and $\mathbf{v}$:

$$\begin{aligned} \mathbf{curl}\,\phi &= \left(\frac{\partial\phi}{\partial x_1}, -\frac{\partial\phi}{\partial x_0}\right), \\ \operatorname{curl}\mathbf{v} &= \frac{\partial v_1}{\partial x_0} - \frac{\partial v_0}{\partial x_1}, \end{aligned}$$

and when $d = 3$:

$$\mathbf{curl}\,\mathbf{v} = \left(\frac{\partial v_2}{\partial x_1} - \frac{\partial v_1}{\partial x_2}, \frac{\partial v_0}{\partial x_2} - \frac{\partial v_2}{\partial x_0}, \frac{\partial v_1}{\partial x_0} - \frac{\partial v_0}{\partial x_1}\right)$$

Let $\mathbf{u}$ be the solution of the Stokes problem (S). The vorticity is defined by:

$$\begin{aligned} \omega &= \operatorname{curl}\mathbf{u} \quad \text{when } d = 2, \\ \boldsymbol{\omega} &= \mathbf{curl}\,\mathbf{u} \quad \text{when } d = 3. \end{aligned}$$

Since the approximation of the velocity is piecewise quadratic, we are looking for a discontinuous piecewise linear vorticity field that belongs to:

$$\begin{array}{rcll} Y_h & = & \{\xi \in L^2(\Omega);\ \xi_{/K} \in P_1,\ \forall K \in \mathcal{T}_h\}, & \text{when } d=2 \\ \mathbf{Y}_h & = & \{\boldsymbol{\xi} \in (L^2(\Omega))^3;\ \xi_{i/K} \in P_1,\ \forall K \in \mathcal{T}_h\}, & \text{when } d=3 \end{array}$$

The approximate variational formulation writes:

$$\omega_h \in Y_h,\ \int_\Omega \omega_h\,\xi\,dx = \int_\Omega \operatorname{curl}\mathbf{u}_h\,\xi\,dx,\ \forall \xi \in Y_h \quad \text{when } d=2,$$

$$\boldsymbol{\omega} \in \mathbf{Y}_h,\ \int_\Omega \boldsymbol{\omega}_h.\boldsymbol{\xi}\,dx = \int_\Omega \mathbf{curl}\,\mathbf{u}_h.\boldsymbol{\xi}\,dx,\ \forall \boldsymbol{\xi} \in \mathbf{Y}_h \quad \text{when } d=3.$$

Example file 4.9: `vorticity.cc`

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
int main(int argc, char** argv) {
  environment rheolef (argc, argv);
  field uh;
  din >> uh;
  const space& Xh = uh.get_space();
  string grad_approx = "P" + itos(Xh.degree()-1) + "d";
  string valued = (uh.size() == 3) ? "vector" : "scalar";
  space Lh  (uh.get_geo(), grad_approx, valued);
  field curl_uh = interpolate (Lh, curl(uh));
  dout << catchmark("w") << curl_uh;
}
```

Comments

As for the stress tensor (see `stress.cc`, page 52), the vorticity is obtained by a direct interpolation of the u_h first derivatives and its approximation is *discontinuous* at inter-element boundaries.

How to run the program

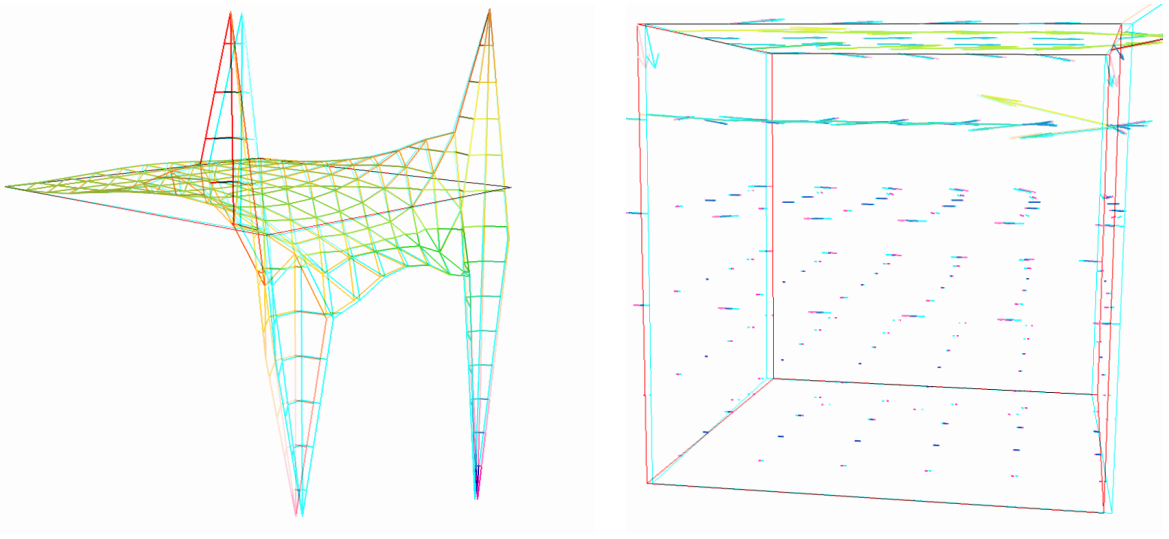


Figure 4.7: The vorticity: elevation view for $d = 2$ and vector representation for $d = 3$ (with anaglyph).

For $d = 2$, just enter:

```
make vorticity
./vorticity < square.field | field -elevation -stereo -
```

and you observe a discontinuous piecewise linear representation of the approximate vorticity. Also, the vorticity presents two sharp peaks at the upper corners of the driven cavity: the vorticity is unbounded and the peaks will increase with mesh refinements. This singularity of the solution is due to the boundary condition for the first component of the velocity u_0 that jumps from zero to one at the corners. The approximate vorticity field can also be projected on a continuous piecewise linear space, using the `-proj` option (See Fig. 4.7 left):

```
./vorticity < square.field | field -elevation -stereo -nofill -
./vorticity < square.field | field -elevation -stereo -proj -
```

For $d = 3$, the whole vorticity vector can also be visualized (See Fig. 4.7 right):

```
./vorticity < cube.field | field -proj -velocity -stereo -
```

In the previous command, the `-proj` option has been used: since the 3D render has no support for discontinuous piecewise linear fields, the P1-discontinuous field is transformed into a P1-continuous one, thanks to a L^2 projection. P1 The following command shows the second component of the vorticity vector, roughly similar to the bidimensional case.

```
./vorticity < cube.field | field -comp 1 -
./vorticity < cube.field | field -comp 1 -proj -
```

4.6 Computing the stream function

Formulation and approximation

When $d = 3$, the stream function is a vector-valued field $\boldsymbol{\psi}$ that satisfies [19, page 90]: $\mathbf{curl}\,\boldsymbol{\psi} = \mathbf{u}$ and $\operatorname{div}\boldsymbol{\psi} = 0$. From the identity:

$$\mathbf{curl}\,\mathbf{curl}\,\boldsymbol{\psi} = -\Delta\boldsymbol{\psi} + \nabla(\operatorname{div}\boldsymbol{\psi})$$

we obtain the following characterization of $\boldsymbol{\psi}$:

$$\begin{array}{rcll} -\Delta\boldsymbol{\psi} & = & \mathbf{curl}\,\mathbf{u} & \text{in } \Omega, \\ \boldsymbol{\psi} & = & 0 & \text{on } \Gamma_{\text{back}} \cup \Gamma_{\text{front}} \cup \Gamma_{\text{top}} \cup \Gamma_{\text{bottom}}, \\ \dfrac{\partial\boldsymbol{\psi}}{\partial n} & = & 0 & \text{on } \Gamma_{\text{left}} \cup \Gamma_{\text{right}}. \end{array}$$

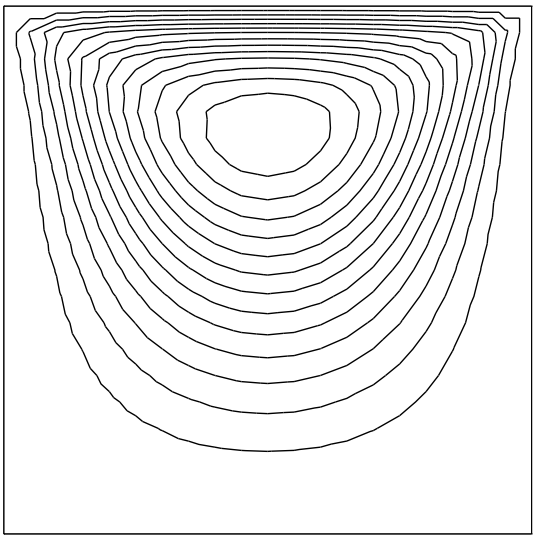
When $d = 2$, the stream function ψ is a scalar-valued field the solution of the following problem [19, page 88]:

$$\begin{array}{rcll} -\Delta\psi & = & \operatorname{curl}\mathbf{u} & \text{in } \Omega, \\ \psi & = & 0 & \text{on } \partial\Omega. \end{array}$$

Example file 4.10: `streamf_cavity.cc`

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
int main (int argc, char** argv) {
  environment rheolef (argc, argv);
  field uh;
  din >> uh;
  const space& Xh = uh.get_space();
  size_t d = uh.get_geo().dimension();
  string valued = (d == 3) ? "vector" : "scalar";
  space Ph (uh.get_geo(), "P2", valued);
  Ph.block("top");  Ph.block("bottom");
  if (d == 3) {
    Ph.block("back"); Ph.block("front");
  } else {
    Ph.block("left"); Ph.block("right");
  }
  trial u (Xh), psi (Ph); test phi (Ph);
  form a = (d == 3) ? integrate (ddot(grad(psi),grad(phi)))
                    : integrate ( dot(grad(psi),grad(phi)));
  form b = (d==3) ? integrate (dot(curl(u),phi))
                  : integrate (curl(u)*phi);
  field psi_h (Ph, 0.);
  field lh = b*uh;
  solver sa (a.uu());
  psi_h.set_u() = sa.solve (lh.u() - a.ub()*psi_h.b());
  dout << catchmark("psi") << psi_h;
}
```

How to run the program



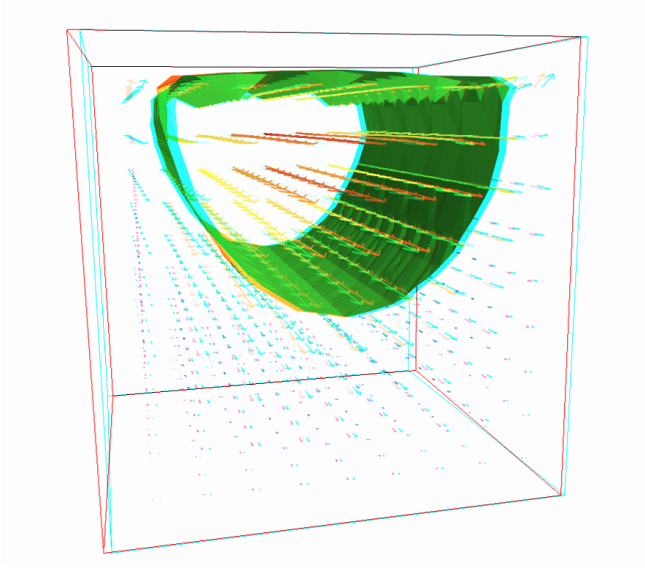


Figure 4.8: The stream function visualization: isolines for $d = 2$, and combined vectors and isonorm surface for $d = 3$.

For $d = 2$, just enter (see Fig. 4.8 left):

```
make streamf_cavity
./streamf_cavity  < square.field | field -bw -
```

For $d = 3$, the whole stream function vector can be visualized:

```
./streamf_cavity < cube.field | field -velocity -
```

The second component of the stream function is showed by:

```
./streamf_cavity < cube.field | field -comp 1 -
```

The combined representation of Fig. 4.8.right has been obtained in two steps. First, enter:

```
./streamf_cavity < cube.field | field -comp 1 -noclean -noexecute -
mv output.vtk psi1.vtk
./streamf_cavity < cube.field | field -velocity -
```

The `-noclean -noexecute` options cause the creation of the '`.vtk`' file for the second component, without running the `mayavi` visualization. Next, in the `mayavi` window associated to the whole stream function, select the `File/Load data/VTK file` menu and load '`psi1.vtk`'. Finally, select the `Vizualize/Module/IsoSurface` menu. Observe that the 3D stream function is mainly represented by its second component.

Chapter 5

Nearly incompressible elasticity and the stabilized Stokes problems

5.1 The incompressible elasticity problem

Formulation

Let us go back to the linear elasticity problem.

When λ becomes large, this problem is related to the *incompressible elasticity* and cannot be solved as it was previously done. To overcome this difficulty, the pressure is introduced :

$$p = -\lambda \mathrm{div}\, \mathbf{u}$$

and the problem becomes:

(E) *find* $\mathbf{u}$ *and* p *defined in* Ω *such that:*

$$\begin{array}{rcrcl} -\,\mathbf{div}(2D(\mathbf{u})) & + & \boldsymbol{\nabla} p & = & \mathbf{f} \text{ in } \Omega, \\ -\,\mathrm{div}\,\mathbf{u} & - & \dfrac{1}{\lambda} p & = & 0 \text{ in } \Omega, \\ +B.C. & & & & \end{array}$$

The variational formulation of this problem expresses:

(VFE) *find* $\mathbf{u} \in V(1)$ *and* $p \in L^2(\Omega)$ *such that:*

$$\begin{array}{rcrcl} a(\mathbf{u},\mathbf{v}) & + & b(\mathbf{v},p) & = & m(\mathbf{f},\mathbf{v}),\ \forall \mathbf{v} \in V(0), \\ b(\mathbf{u},q) & - & c(p,q) & = & 0,\ \forall q \in L^2_0(\Omega), \end{array}$$

where

$$\begin{aligned} m(\mathbf{u},\mathbf{v}) &= \int_\Omega \mathbf{u}.\mathbf{v}\, dx, \\ a(\mathbf{u},\mathbf{v}) &= \int_\Omega D(\mathbf{u}) : D(\mathbf{v})\, dx, \\ b(\mathbf{v},q) &= -\int_\Omega \mathrm{div}(\mathbf{v})\, q\, dx. \\ c(p,q) &= \frac{1}{\lambda}\int_\Omega p\, q\, dx. \\ V &= \{\mathbf{v} \in (H^1(\Omega))^2;\ \mathbf{v} = 0 \text{ on } \Gamma_{left} \cup \Gamma_{bottom}\} \end{aligned}$$

When λ becomes large, we obtain the incompressible elasticity problem, that coincides with the Stokes problem.

Approximation

As for the Stokes problem, the Talor-Hood [22] finite element approximation is considered. We introduce a mesh $\mathcal{T}_h$ of Ω and the following finite dimensional spaces:

$$\begin{aligned} X_h &= \{\mathbf{v} \in (H^1(\Omega));\ \mathbf{v}_{/K} \in (P_2)^2,\ \forall K \in \mathcal{T}_h\}, \\ V_h(\alpha) &= X_h \cap V, \\ Q_h &= \{q \in L^2(\Omega)) \cap C^0(\bar{\Omega});\ q_{/K} \in P_1,\ \forall K \in \mathcal{T}_h\}, \end{aligned}$$

The approximate problem writes:

$(VFE)_h$ *find* $\mathbf{u}_h \in V_h(1)$ *and* $p \in Q_h$ *such that:*

$$\begin{array}{rcrcl} a(\mathbf{u}_h, \mathbf{v}) & + & b(\mathbf{v}, p_h) & = & 0,\ \forall \mathbf{v} \in V_h(0), \\ b(\mathbf{u}_h, q) & - & c(p, q) & = & 0,\ \forall q \in Q_h. \end{array}$$

Example file 5.1: `incompressible-elasticity.cc`

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
#include "embankment.icc"
int main(int argc, char**argv) {
  environment rheolef (argc, argv);
  geo omega (argv[1]);
  Float inv_lambda = (argc > 2 ? atof(argv[2]) : 0);
  size_t d = omega.dimension();
  space Xh = embankment_space(omega, "P2");
  space Qh (omega, "P1");
  point f (0,0,0);
  f [d-1] = -1;
  trial u (Xh), p (Qh); test v (Xh), q (Qh);
  field lh = integrate (dot(f,v));
  form  a  = integrate (2*ddot(D(u),D(v)));
  form  b  = integrate (-div(u)*q);
  form  mp = integrate (p*q);
  form  c  = inv_lambda*mp;
  field uh (Xh, 0), ph (Qh, 0);
  solver_abtb elasticity (a.uu(), b.uu(), c.uu(), mp.uu());
  elasticity.solve (lh.u() - a.ub()*uh.b(), -(b.ub()*uh.b()),
                    uh.set_u(),             ph.set_u());
  dout << catchmark("inv_lambda")  << inv_lambda << endl
       << catchmark("u")  << uh
       << catchmark("p")  << ph;
}
```

Comments

The problem admits the following matrix form:

$$\begin{pmatrix} \mathtt{a.uu} & \mathtt{trans(b.uu)} \\ \mathtt{b.uu} & -\mathtt{c.uu} \end{pmatrix} \begin{pmatrix} \mathtt{uh.u} \\ \mathtt{ph.u} \end{pmatrix} = \begin{pmatrix} \mathtt{lh.u} - \mathtt{a.ub} * \mathtt{uh.b} \\ -\mathtt{b.ub} * \mathtt{uh.b} \end{pmatrix}$$

The problem is similar to the Stokes one (see page 59). This system is solved by:

```
  solver_abtb elasticity (a.uu(), b.uu(), c.uu(), mp.uu());
  elasticity.solve (lh.u() - a.ub()*uh.b(), -(b.ub()*uh.b()),
                    uh.set_u(),             ph.set_u());
```

For two-dimensional problems, a direct solver is used by default. In the three-dimensional case, an iterative algorithm is the default: the preconditioned conjugate gradient. The preconditioner is here the mass matrix `mp.uu` for the pressure. As showed in [24], the number of iterations need by the conjugate gradient algorithm to reach a given precision is then independent of the mesh size and is uniformly bounded when λ becomes small, i.e. in the incompressible case.

How to run the program

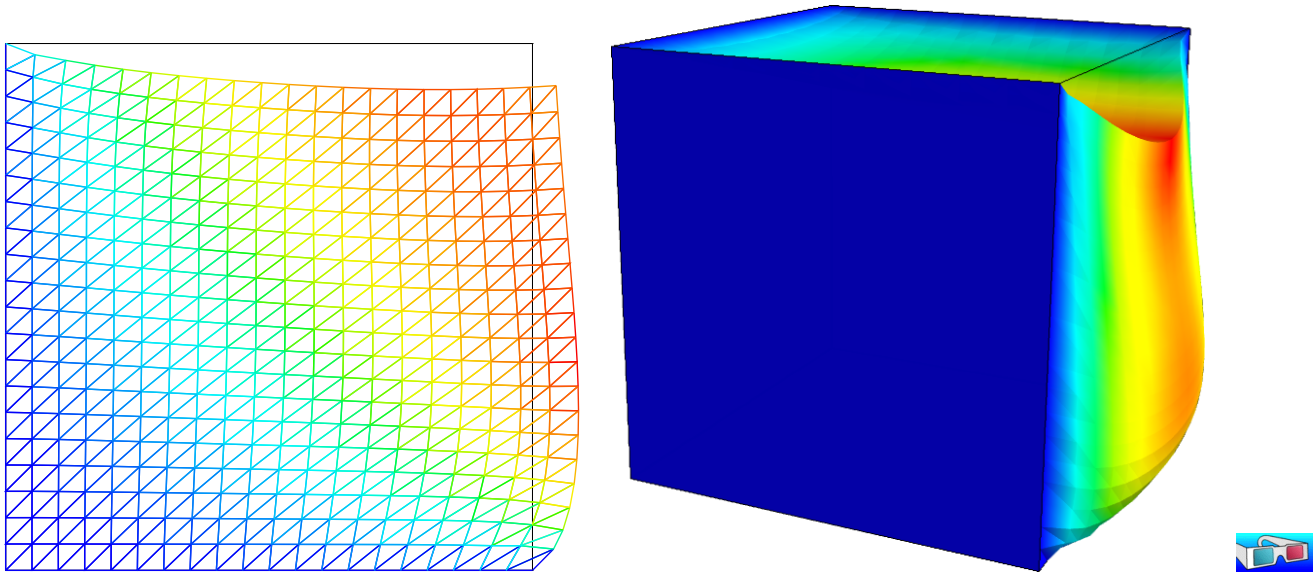


Figure 5.1: The incompressible linear elasticity ($\lambda = +\infty$) for $N = 2$ and $N = 3$.

We assume that the previous code is contained in the file '`incompressible-elasticity.cc`'. Compile the program as usual (see page 18):

```
make incompressible-elasticity
```

and enter the commands:

```
mkgeo_grid -t 10 > square.geo
./incompressible-elasticity square.geo 0 > square.field
field square.field -mayavi -nofill

mkgeo_grid -T 10 > cube.geo
./incompressible-elasticity cube.geo 0 > cube.field
field cube.field -mayavi -fill -scale 2.5
```

The visualization is performed as usual: see section 4.1, page 50. Compare the results on FIG. 5.1, obtained for $\lambda = +\infty$ with those of FIG. 4.2, page 50, obtained for $\lambda = 1$.

Finally, the stress computation and the mesh adaptation loop is left as an exercise to the reader.

5.2 The $P_1b - P_1$ element for the Stokes problem

Formulation and approximation

Let us go back to the Stokes problem. In section 4.4, page 57, the Taylor-Hood finite element was considered. Here, we turn to the mini-element [3] proposed by Arnold, Brezzi and Fortin, also well-known as the *P1-bubble* element. This element is generally less precise than the Taylor-Hood one, but becomes popular, mostly because it is easy to implement in two and three dimensions and furnishes a P_1 approximation of the velocity field. Moreover, this problem develops some links with stabilization technique and will presents some new **Rheolef** features.

We consider a mesh $\mathcal{T}_h$ of $\Omega \subset \mathbb{R}^d$, $d = 2, 3$ composed only of simplicial elements: triangles when $d = 2$ and tetrahedra when $d = 3$. The following finite dimensional spaces are introduced:

$$\begin{aligned}
\mathbf{X}_h^{(1)} &= \{\mathbf{v} \in (H^1(\Omega))^d;\ \mathbf{v}_{/K} \in (P_1)^d,\ \forall K \in \mathcal{T}_h\},\\
\mathbf{B}_h &= \{\boldsymbol{\beta} \in (C^0(\bar{\Omega}))^d;\ \boldsymbol{\beta}_{/K} \in B(K)^d, \forall K \in \mathcal{T}_h\}\\
\mathbf{X}_h &= \mathbf{X}_h^{(1)} \oplus \mathbf{B}_h\\
\mathbf{V}_h(\alpha) &= X_h \cap \mathbf{V}(\alpha),\\
Q_h &= \{q \in L^2(\Omega)) \cap C^0(\bar{\Omega});\ q_{/K} \in P_1,\ \forall K \in \mathcal{T}_h\},
\end{aligned}$$

where $B(K) = \mathrm{vect}(\lambda_1 \times \ldots \times \lambda_{d+1})$ and λ_i are the barycentric coordinates of the simplex K. The $B(K)$ space is related to the *bubble* local space. The approximate problem is similar to (4.4), page 58, up to the choice of finite dimensional spaces.

Remark that the velocity field splits in two terms: $\mathbf{u}_h = \mathbf{u}_h^{(1)} + \mathbf{u}_h^{(b)}$, where $\mathbf{u}_h^{(1)} \in \mathbf{X}_h^{(1)}$ is continuous and piecewise linear, and $\mathbf{u}_h^{(b)} \in \mathbf{B}_h$ is the bubble term.

We consider the abrupt contraction geometry:

$$\Omega =]-L_u, 0[\times]0, c[\cup [0, L_d[\times]0, 1[$$

where $c \geqslant 1$ stands for the contraction ratio, and $L_u, L_d > 0$, are the upstream and downstream tube lengths. The boundary conditions on $\mathbf{u} = (u_0, u_1)$ for this test case are:

$$\begin{aligned}
u_0 &= u_{\text{poiseuille}} \ \text{ and } \ u_1 = 0 \ \text{ on } \Gamma_{\text{upstream}}\\
\mathbf{u} &= 0 \ \text{ on } \Gamma_{\text{wall}}\\
\frac{\partial u_0}{\partial x_1} &= 0 \ \text{ and } \ u_1 = 0 \ \text{ on } \Gamma_{\text{axis}}\\
\frac{\partial \mathbf{u}}{\partial n} &= 0 \ \text{ on } \Gamma_{\text{downstream}}
\end{aligned}$$

where

$$\begin{aligned}
\Gamma_{\text{upstream}} &= \{-L_u\}\times]0, c[\\
\Gamma_{\text{downstream}} &= \{L_d\}\times]0, 1[\\
\Gamma_{\text{axis}} &=]-L_u, L_d[\times\{0\}\\
\Gamma_{\text{wall}} &=]-L_u, 0[\times\{c\} \ \cup \ \{0\}\times]1, c[\cup]0, L_d[\times\{1\}
\end{aligned}$$

The matrix structure is similar to those of the Taylor-Hood element (see section 4.4, page 57). Since $\mathbf{X}_h = \mathbf{X}_h^{(1)} \oplus \mathbf{B}_h$, any element $u_h \in X_h$ can be written as a sum $u_h = u_{1,h} + u_{b,h}$ where $u_{1,h} \in \mathbf{X}_h^{(1)}$ and $u_{b,h} \in \mathbf{B}_h$. Remark that

$$a(u_{1,h}, v_{b,h}) = 0, \quad \forall u_{1,h} \in \mathbf{X}_h^{(1)}, \quad \forall v_{b,h} \in \mathbf{B}_h.$$

Thus, the form $a(.,.)$ defined over $\mathbf{X}_h \times \mathbf{X}_h$ writes simply as the sum of the forms $a_1(.,.)$ and $a_b(.,.)$, defined over $\mathbf{X}_h^{(1)} \times \mathbf{X}_h^{(1)}$ and $\mathbf{B}_h \times \mathbf{B}_h$ respectively. Finaly, the form $b(.,.)$ defined over $\mathbf{X}_h \times Q_h$ writes as the sum of the forms $b_1(.,.)$ and $b_b(.,.)$, defined over $\mathbf{X}_h^{(1)} \times Q_h$ and $\mathbf{B}_h \times Q_h$ respectively. Then, the linear system admits the following block structure :

$$\begin{pmatrix} A_1 & 0 & B_1^T \\ 0 & A_b & B_b^T \\ B_1 & B_b & 0 \end{pmatrix} \begin{pmatrix} U_1 \\ U_b \\ P \end{pmatrix} = \begin{pmatrix} L_1 \\ L_b \\ L_p \end{pmatrix}$$

An alternative and popular implementation of this element eliminates the unknowns related to the bubble components (see e.g. [1], page 24). Remark that, on any element $K \in \mathcal{T}_h$, any bubble

function v_K that belongs to $B(K)$ vanishes on the boundary of K and have a compact support in K. Thus, the A_b matrix is block-diagonal. Moreover, A_b is invertible and U_b writes :

$$U_b = A_b^{-1}(B_b^T p - L_b)$$

As A_b is block-diagonal, its inverse can be efficiently inverted at the element level during the assembly process. Then, U_b can be easily eliminated from the system that reduces to:

$$\begin{pmatrix} A_1 & B_1^T \\ B_1 & -C \end{pmatrix} \begin{pmatrix} U_1 \\ P \end{pmatrix} = \begin{pmatrix} L_1 \\ \tilde{L}_p \end{pmatrix}$$

where $\tilde{L}_p = L_p - A_b^{-1} L_p$ and $C = B_b A_b^{-1} B_b^T$. Remarks that the matrix structure is similar to those of the nearly incompressible elasticity (see 5.1, page 5.1). This reduced matrix formulation of the $P_1 b - P1$ element is similar to the direct $P_1 - P_1$ stabilized element, proposed by Brezzi and Pitkäranta [9].

Example file 5.2: `stokes_contraction_bubble.cc`

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
#include "poiseuille.h"

int main(int argc, char**argv) {
  environment rheolef (argc, argv);
  geo  omega (argv[1]);
  space X1h (omega, "P1", "vector");
  space Bh  (omega, "bubble", "vector");
  space Qh  (omega, "P1");
  space Wh  (omega["upstream"], "P1");
  X1h.block ("wall");
  X1h.block ("upstream");
  X1h[1].block ("axis");
  X1h[1].block ("downstream");
  trial u1 (X1h), ub (Bh), p (Qh);
  test  v1 (X1h), vb (Bh), q (Qh);
  form mp = integrate (p*q);
  form b1 = integrate (-div(u1)*q);
  form bb = integrate (-div(ub)*q);
  form a1 = integrate (2*ddot(D(u1),D(v1)));
  form_option_type fopt;
  fopt.invert = true;
  form inv_ab = integrate (2*ddot(D(ub),D(vb)), fopt);
  form c = bb*inv_ab*trans(bb);
  field u1h (X1h, 0), ph (Qh, 0);
  string sys_coord = omega.coordinate_system_name();
  Float cr = omega.xmax()[1];
  u1h[0]["upstream"] = interpolate (Wh, u_poiseuille(cr,sys_coord));
  solver_abtb stokes (a1.uu(), b1.uu(), c.uu(), mp.uu());
  stokes.solve (-(a1.ub()*u1h.b()), -(b1.ub()*u1h.b()),
                u1h.set_u(),        ph.set_u());
  dout << catchmark("inv_lambda") << 0 << endl
       << catchmark("u")  << u1h
       << catchmark("p")  << ph;
}
```

Comments

First, A_b^{-1} is computed as:

```
  form_option_type fopt;
  fopt.invert = true;
  form inv_ab = integrate (2*ddot(D(ub),D(vb)), fopt);
```

Notice the usage of the optional parameter `fopt` to the `integrate` function. As the form is bloc-diagonal, its inverse is computed element-by-element during the assembly process. Next, the $C = B_b A_b^{-1} B_b^T$ form is simply computed as:

```
  form c = bb*inv_ab*trans(bb);
```

Notice also the automatic computation of the geometric coordinate system and contraction ratio c from the input mesh, as:

```
  string sys_coord = omega.coordinate_system_name();
  Float cr = omega.xmax()[1];
```

These parameters are send to the function that computes the Poiseuille input flow boundary condition:

```
  u1h[0]["upstream"] = interpolate (Wh, u(cr,sys_coord));
```

The file `poiseuille.h` contains code for the velocity and stream function boundary conditions.

Example file 5.3: `poiseuille.h`

```
struct u_poiseuille : field_functor<u_poiseuille,Float> {
  Float operator() (const point& x) const {
    return a*(c+x[1])*(c-x[1]); }
  u_poiseuille (const Float& c1, std::string sc) : c(c1)
   { a = (sc == "cartesian") ? 3/(2*pow(c,3)) : 4/pow(c,4); }
  protected: Float c, a;
};
struct psi_poiseuille : field_functor<psi_poiseuille,Float> {
  Float operator() (const point& x) const {
    return xy ? a*sqr(c-x[1])*(2*c+x[1]) : a*sqr(c-x[1])*sqr(c+x[1]); }
  psi_poiseuille (const Float& c1, std::string sc)
   : c(c1), xy(sc == "cartesian")
   { a = xy ? -1/(2*pow(c,3)) : -1/pow(c,4); }
  protected: Float c, a; bool xy;
};
```

The Poiseuille velocity upstream boundary condition $u_{\text{poiseuille}}$ has been scaled such that the total flow rate is equal to one. The stream function is equal to -1 on the axis and to zero on the wall. This file contains also a treatment of the axisymmetric variant of the geometry: this case will be presented in the next paragraphs.

How to run the program

The boundary conditions in this example are related to an abrupt contraction geometry with a free surface. The corresponding mesh '`contraction.geo`' can be easily builded from the geometry description file '`contraction.mshcad`', which is provided in the example directory of the **Rheolef** distribution. The building mesh procedure is presented with details in appendix B, page B.

```
gmsh -2 contraction.mshcad -o contraction.msh
msh2geo contraction.msh > contraction.geo
geo contraction.geo
```

The mesh is represented on Fig. 5.2.top. Then, the computation and the visualization writes:

```
make stokes_contraction_bubble
./stokes_contraction_bubble contraction.geo > contraction-P1.field
field contraction-P1.field -mayavi -velocity
```

The visualization of the velocity field brings few informations about the properties of the flow. The stream function is more relevant for stationary flow visualization.

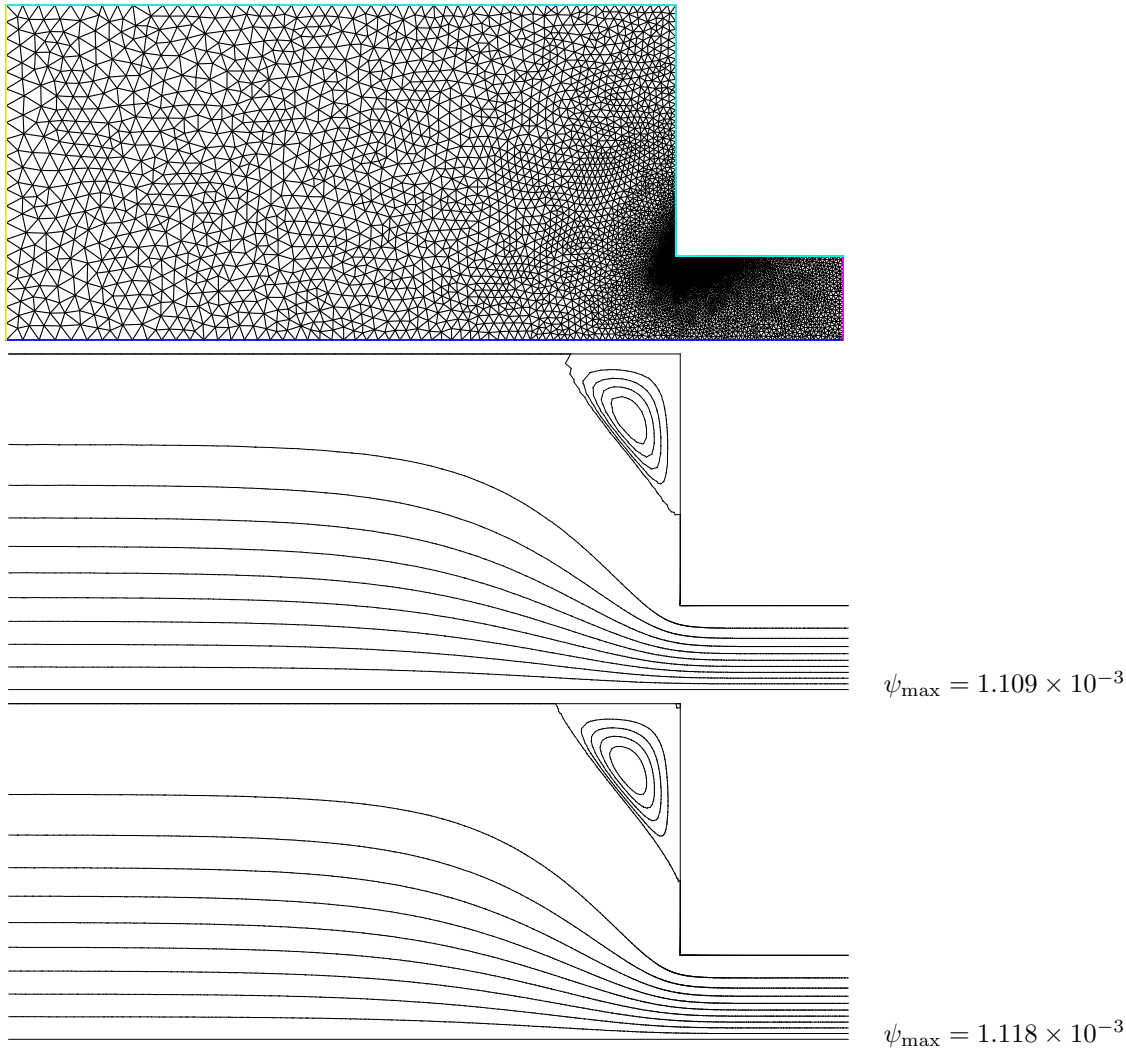


Figure 5.2: Solution of the Stokes problem in the abrupt contraction: (top) the mesh; (center) the P_1 stream function associated to the P_1b-P_1 element; (bottom) the P_2 stream function associated to the P_2-P_1 Taylor-Hood element.

Example file 5.4: `streamf_contraction.cc`

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
#include "poiseuille.h"
int main (int argc, char** argv) {
  environment rheolef (argc, argv);
  field uh;
  din >> uh;
  const geo& omega = uh.get_geo();
  size_t d = omega.dimension();
  string sys_coord = omega.coordinate_system_name();
  Float c = omega.xmax()[1];
  string approx = "P" + itos(uh.get_space().degree());
  space Ph (omega, approx);
  Ph.block("upstream");
  Ph.block("wall");
  Ph.block("axis");
  space Wh (omega["upstream"], approx);
  const space& Xh = uh.get_space();
  field psi_h (Ph, 0);
  psi_h["upstream"] = interpolate (Wh, psi_poiseuille(c,sys_coord));
  psi_h["wall"] = 0;
  psi_h["axis"] = -1;
  form_option_type fopt;
  fopt.ignore_sys_coord = true;
  trial psi (Ph), u (Xh);
  test   xi (Ph), v (Xh);
  form a = (d == 3) ? integrate (ddot(grad(psi), grad(xi)))
                    : integrate ( dot(grad(psi), grad(xi)), fopt);
  field lh = integrate (dot(uh,bcurl(xi)));
  solver sa (a.uu());
  psi_h.set_u() = sa.solve (lh.u() - a.ub()*psi_h.b());
  dout << catchmark("psi") << psi_h;
}
```

Notice the usage of the optional parameter `fopt` to the `integrate` function.

```
fopt.ignore_sys_coord = true;
```

In the axisymmetric coordinate system, there is a specific definition of the stream function, together with the use of a variant of the **curl** operator, denoted as **bcurl** in **Rheolef**.

```
field lh = integrate (dot(uh,bcurl(xi)));
```

The axisymmetric case will be presented in the next section. By this way, our code is abble to deal with both cartesian and axisymmetric geometries.

The stream function ψ (see also section 4.6) is computed and visualized as:

```
make streamf_contraction
./streamf_contraction < contraction-P1.field > contraction-P1-psi.field
field contraction-P1-psi.field -mayavi
field contraction-P1-psi.field -n-iso 15 -n-iso-negative 10 -bw
```

The P_1 stream function is represented on Fig. 5.2.center. The stream function is zero along the wall and the line separating the main flow and the vortex located in the outer corner of the contraction. Thus, the isoline associated to the zero value separates the main flow from the vortex. In order to observe this vortex, an extra `-n-iso-negative 10` option is added: ten isolines are drawn for negatives values of ψ, associated to the main flow, and `n_iso-10` for the positives values, associated to the vortex.

A similar computation based on the Taylor-Hood $P_2 - P_1$ element is implemented in `stokes_contraction.cc`. The code is similar, up to the boundary conditions, to `stokes_cavity.cc` (see page 58): thus it is not listed here but is available in the **Rheolef** example directory.

```
make stokes_contraction
./stokes_contraction contraction.geo > contraction-P2.field
field contraction-P2.field -mayavi -velocity
./streamf_contraction < contraction-P2.field > contraction-P2-psi.field
field contraction-P2-psi.field -n-iso-negative 10 -bw
```

The associated P_2 stream function is represented on Fig. 5.2.bottom. Observe that the two solutions are similar and that the vortex activity, defined as $\psi_{\max}$, is accurately computed with the two methods (see also [41], Fig. 5.11.a, page 143).

```
field contraction-P1-psi.field -max
field contraction-P2-psi.field -max
```

Recall that the stream function is negative in the main flow and positive in the vortex located in the outer corner of the contraction. Nevertheless, the Taylor-Hood based solution is more accurate : this is perceptible on the graphic, in the region where the upstream vortex reaches the boundary.

5.3 Axisymmetric geometries

Axisymmetric geometries are fully supported in **Rheolef**: the coordinate system is associated to the geometry description, stored together with the mesh in the '`.geo`' and this information is propagated in spaces, forms and fields without any change in the code. Thus, a code that works in plane a 2D plane geometry is able to support a 3D axisymmetric one without changes. A simple axisymmetric geometry writes:

```
mkgeo_grid -t 10 -zr > square-zr.geo
more square-zr.geo
```

Remark the additional line in the header:

```
coordinate_system zr
```

The axis of symmetry is denoted as z while the polar coordinates are (r, θ). By symmetry, the problem is supposed to be independent upon θ and the computational domain is described by $(x_0, x_1) = (z, r)$. Conversely, in some cases, it could be convenient to swap the order of the coordinates and use (r, z): this feature is obtained by the `-rz` option:

```
mkgeo_grid -t 10 -rz > square-rz.geo
more square-rz.geo
```

Axisymmetric problems uses L^2 functional space equipped with the following weighted scalar product

$$(f, g) = \int_\Omega f(z, r)\, g(z, r)\, r\, dr dz$$

and all usual bilinear forms support this weight. Thus, the **coordinate system can be chosen at run time** and we can expect an efficient source code reduction.

5.4 The axisymmetric stream function and stress tensor

In the axisymmetric case, the velocity field $\mathbf{u} = (u_z, u_r)$ can be expressed in terms of the Stokes stream function ψ by (see Batchelor [5, p.453] and [51]):

$$\mathbf{u} = (u_z, u_r) = \left(\frac{1}{r}\frac{\partial \psi}{\partial r}, \quad -\frac{1}{r}\frac{\partial \psi}{\partial z} \right) \tag{5.1}$$

Recall that in the axisymmetric case:

$$\mathbf{curl}\,\psi = \left(\frac{1}{r}\frac{\partial(r\psi)}{\partial r},\quad -\frac{\partial\psi}{\partial z}\right)$$

Thus, from this definition, in axisymmetric geometries $\mathbf{u} \neq \mathbf{curl}\,\psi$ and the definition of ψ differs from the 2D plane or 3D cases (see section 4.6, page 62).

Let us turn to a variational formulation in order to compute ψ from $\mathbf{u}$. For any $\xi \in H^1(\Omega)$, let us multiply (5.1) by $\mathbf{v} = (\partial_r\xi,\ -\partial_z\xi)$ and then integrate over Ω with the $r\,\mathrm{d}r\,\mathrm{d}z$ weight. For any known $\mathbf{u}$ velocity field, the problem writes:

(P): find $\psi \in \Psi(\psi_\Gamma)$ such that

$$a(\psi,\xi) = l(\xi),\quad \forall \xi \in \Psi(0)$$

where we have introduced the following bilinear forms:

$$\begin{aligned} a(\psi,\xi) &= \int_\Omega \left(\frac{\partial\psi}{\partial r}\frac{\partial\xi}{\partial r} + \frac{\partial\psi}{\partial z}\frac{\partial\xi}{\partial z}\right)\,\mathrm{d}r\,\mathrm{d}z \\ l(\xi) &= \int_\Omega \left(\frac{\partial\xi}{\partial r}u_z - \frac{\partial\xi}{\partial z}u_r\right)\,r\,\mathrm{d}r\,\mathrm{d}z \end{aligned}$$

These forms are defined in '`streamf_contraction.cc`' as:

```
form_option_type fopt;
fopt.ignore_sys_coord = true;
form a = integrate (dot(grad(psi), grad(xi)), fopt);
```

and

```
field lh = integrate (dot(uh,bcurl(xi)));
```

The `fopt.ignore_sys_coord` alows us to drops the r integration weight, i.e. replace $r\,\mathrm{d}r\,\mathrm{d}z$ by $\mathrm{d}r\,\mathrm{d}z$ when computing the $a(.,.)$ form. Conversely, l involves the **bcurl** operator defined as:

$$\mathbf{bcurl}\,\xi = \left(\frac{\partial\xi}{\partial r},\quad -\frac{\partial\xi}{\partial z}\right)$$

It is is closely related but differs from the standard **curl** operator:

$$\mathbf{curl}\,\xi = \left(\frac{1}{r}\frac{\partial(r\xi)}{\partial r},\quad -\frac{\partial\xi}{\partial z}\right)$$

The **bcurl** operator is a specific notation introduced in **Rheolef**: it coincides with the usual **curl** operator except for axisymmetric geometries. In tht case, it refers to the Batchelor trick, suitable for the computation of the stream function.

As an example, let us reconsider the contraction geometry (see section 5.2, page 67), extended in the axisymmetric case. In that case, the functional space is defined by:

$$\Psi(\psi_\Gamma) = \{\varphi \in H^1(\Omega);\ \varphi = \psi_\Gamma \ \text{ on } \Gamma_{\text{upstream}} \cup \Gamma_{\text{wall}} \cup \Gamma_{\text{axis}}\}$$

with

$$\psi_\Gamma = \begin{cases} \psi_{\text{poiseuile}} & \text{on } \Gamma_{\text{upstream}} \\ 0 & \text{on } \Gamma_{\text{wall}} \\ -1 & \text{on } \Gamma_{\text{axis}} \end{cases}$$

This space corresponds to the imposition of Dirichlet boundary conditions on Γ_{upstream}, Γ_{wall} and Γ_{axis} and a Neumann boundary condition on $\Gamma_{\text{downstream}}$.

The following unix commands generate the axisymmetric geometry:

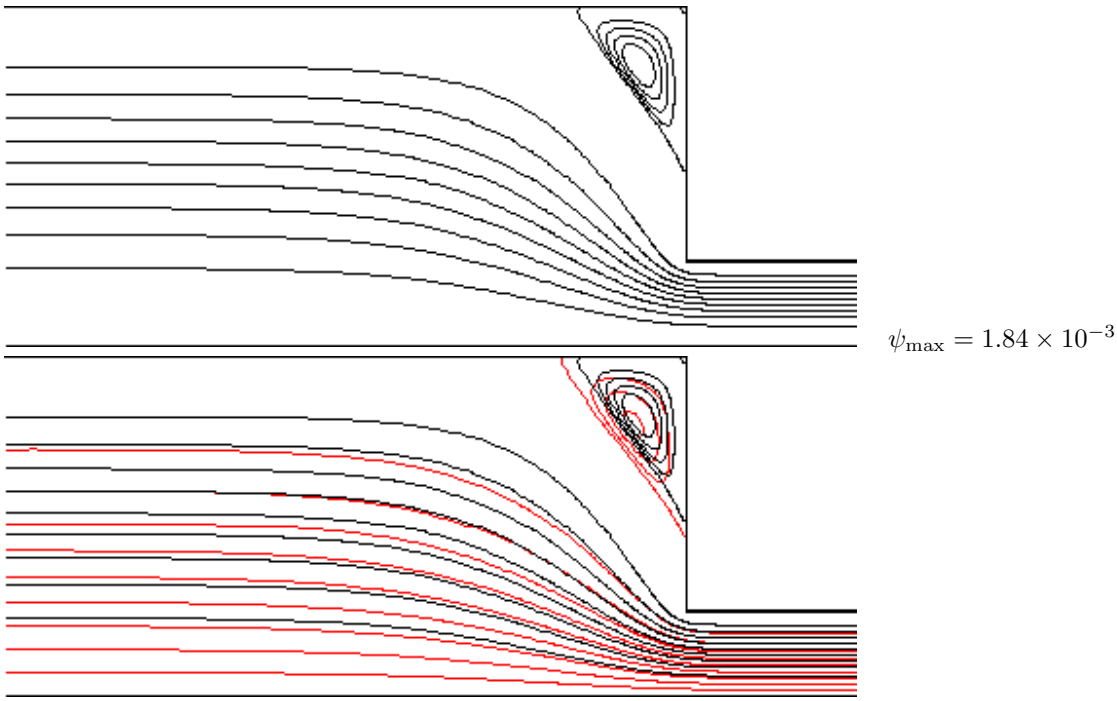


Figure 5.3: Solution of the axisymmetric Stokes problem in the abrupt contraction: (top) the P_2 stream function associated to the $P2 - P_1$ element; (bottom) comparison with the 2D Cartesian solution (in red).

```
gmsh -2 contraction.mshcad -o contraction.msh
msh2geo -zr contraction.msh > contraction-zr.geo
more contraction-zr.geo
geo contraction-zr.geo
```

The previous code `stokes_contraction.cc` and `streamf_contraction.cc` are both reused as:

```
./stokes_contraction contraction-zr.geo > contraction-zr-P2.field
./streamf_contraction < contraction-zr-P2.field > contraction-zr-P2-psi.field
field contraction-zr-P2-psi.field -n-iso-negative 10 -bw
```

The solution is represented on Fig. 5.3: it slightly differs from the 2D Cartesian solution, as computed in the previous section (see Fig. 5.2). The vortex size is smaller but its intensity $\psi_{\max} = 1.84 \times 10^{-3}$ is higher. Despite the stream functions looks like similar, the plane solutions are really different, as we can observe from a cut of the first component of the velocity along the axis (see Fig. 5.4):

```
field contraction-P2.field    -comp 0 -cut -normal 0 1 -origin 0 1e-15
field contraction-zr-P2.field -comp 0 -cut -normal 0 1 -origin 0 1e-15
```

The `1e-15` argument replace the zero value, as the mesh intersection cannot yet be done exactly on the boundary. Notice that the `stokes_contraction_bubble.cc` can be also reused in a similar way:

```
./stokes_contraction_bubble contraction-zr.geo > contraction-zr-P1.field
./streamf_contraction < contraction-zr-P1.field > contraction-zr-P1-psi.field
field contraction-zr-P1-psi.field -n-iso-negative 10 -bw
```

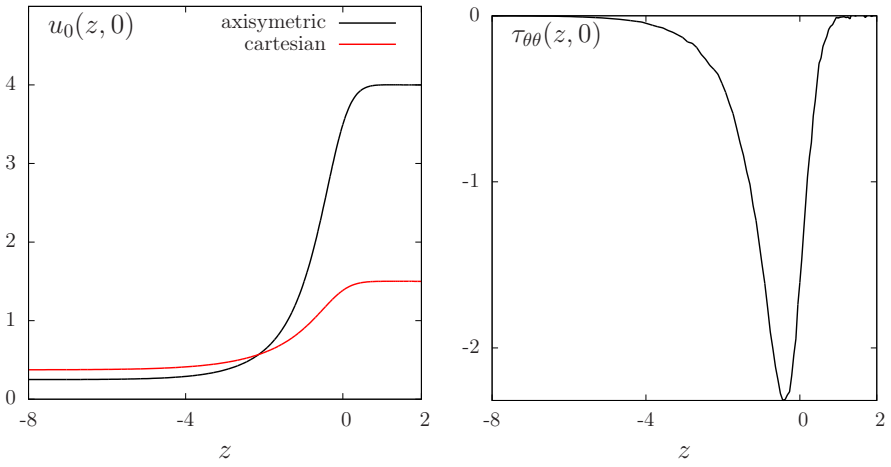


Figure 5.4: Solution of the plane and axisymmetric Stokes problem in the abrupt contraction: cut along the axis of symmetry: (left): u_0; (right) $\tau_{\theta\theta}$.

There is another major difference with axisymmetric problems: the rate of deformation tensor writes:

$$\tau = 2D(\mathbf{u}) = \begin{pmatrix} \tau_{zz} & \tau_{rz} & 0 \\ \tau_{rz} & \tau_{rr} & 0 \\ 0 & 0 & \tau_{\theta\theta} \end{pmatrix}$$

Thus, there is an additional non-zero component $\tau_{\theta\theta}$ that is automatically integrated into the computations in **Rheolef**. The incompressibility relation leads to $\mathrm{tr}(\tau) = \tau_{zz} + \tau_{rr} + \tau_{\theta\theta} = 0$. Here $\sigma_{\mathrm{tot}} = -p.I + \tau$ is the total Cauchy stress tensor (by a dimensionless procedure, the viscosity can be taken as one). By reusing the `stress.cc` code (see page 52) we are able to compute the tensor components:

```
make stress
./stress < contraction-zr-P1.field  > contraction-zr-P1-tau.field
```

The visualization along the axis of symmetry for the $\tau_{\theta\theta}$ component is obtained by (see Fig. 5.4):

```
field contraction-zr-P1-tau.field -comp 22 -proj -cut -normal 0 1 -origin 0 1e-15
```

Recall that the τ_{zz} and τ_{rr} components are obtained by the `-comp 00` and `-comp 11` options, respectively. The non-zero values along the axis of symmetry expresses the elongational effects in the entry region of the abrupt contraction.

Chapter 6

Time-dependent problems

6.1 The heat equation

Formulation

Let $T > 0$, $\Omega \subset \mathbb{R}^d$, $d = 1, 2, 3$ and $f$defined in Ω. The heat problem writes:

(P): *find u, defined in $\Omega\times]0,T[$, such that*

$$\begin{aligned} \frac{\partial u}{\partial t} - \Delta u &= f \ \text{ in } \Omega\times]0,T[, \\ u(0) &= 0 \ \text{ in } \Omega, \\ u(t) &= 0 \ \text{ on } \partial\Omega\times]0,T[. \end{aligned}$$

where f is a known function. In the present example, we consider $f = 1$.

Approximation

Let $\Delta t > 0$ and $t_n = n\Delta t$, $n \geqslant 0$. The problem is approximated with respect to time by the following first-order implicit Euler scheme:

$$\frac{u^{n+1} - u^n}{\Delta t} - \Delta u^{n+1} = f(t_{n+1}) \ \text{ in } \Omega$$

where $u^n \approx u(n\Delta t)$ and $u^{(0)} = 0$. The variational formulation of the time-discretized problem writes:
$(VF)_n$*: Let u^n being known, find $u^{n+1} \in H_0^1(\Omega)$ such that*

$$a\,(u^{n+1},\, v) = l^{(n)}(v),\ \forall v \in H_0^1(\Omega).$$

where

$$\begin{aligned} a(u,v) &= \int_\Omega (uv + \Delta t\, \nabla u.\nabla v)\ v\,\mathrm{d}x \\ l^{(n)}(v) &= \int_\Omega (u^n + \Delta t\, f(t_{n+1}))\ v\,\mathrm{d}x \end{aligned}$$

This is a Poisson-like problem. The discretization with respect to space of this problem is similar to those presented in section 1.1, page 15.

Example file 6.1: `heat.cc`

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
int main (int argc, char **argv) {
  environment rheolef (argc, argv);
  geo omega (argv[1]);
  size_t n_max = (argc > 2) ? atoi(argv[2]) : 100;
  Float delta_t = 0.5/n_max;
  space Xh (omega, "P1");
  Xh.block ("boundary");
  trial u (Xh); test v (Xh);
  form a = integrate (u*v + delta_t*dot(grad(u),grad(v)));
  solver sa = ldlt (a.uu());
  field uh (Xh, 0);
  branch event ("t","u");
  dout << event (0, uh);
  for (size_t n = 1; n <= n_max; n++) {
    field rhs = uh + delta_t;
    field lh = integrate (rhs*v);
    uh.set_u() = sa.solve (lh.u() - a.ub()*uh.b());
    dout << event (n*delta_t, uh);
  }
}
```

Comments

Notice the use of the `branch` class:

```
  branch event ("t","u");
```

this is a wrapper class that is used here to print the branch of solution $(t_n, u^n)_{n \geqslant 0}$, on the standard output in the '`.branch`' file format. An instruction as:

```
  dout << event (t,uh);
```

is equivalent to the formatted output

```
  dout << catchmark("t") << t << endl
       << catchmark("u") << uh;
```

How to run the program

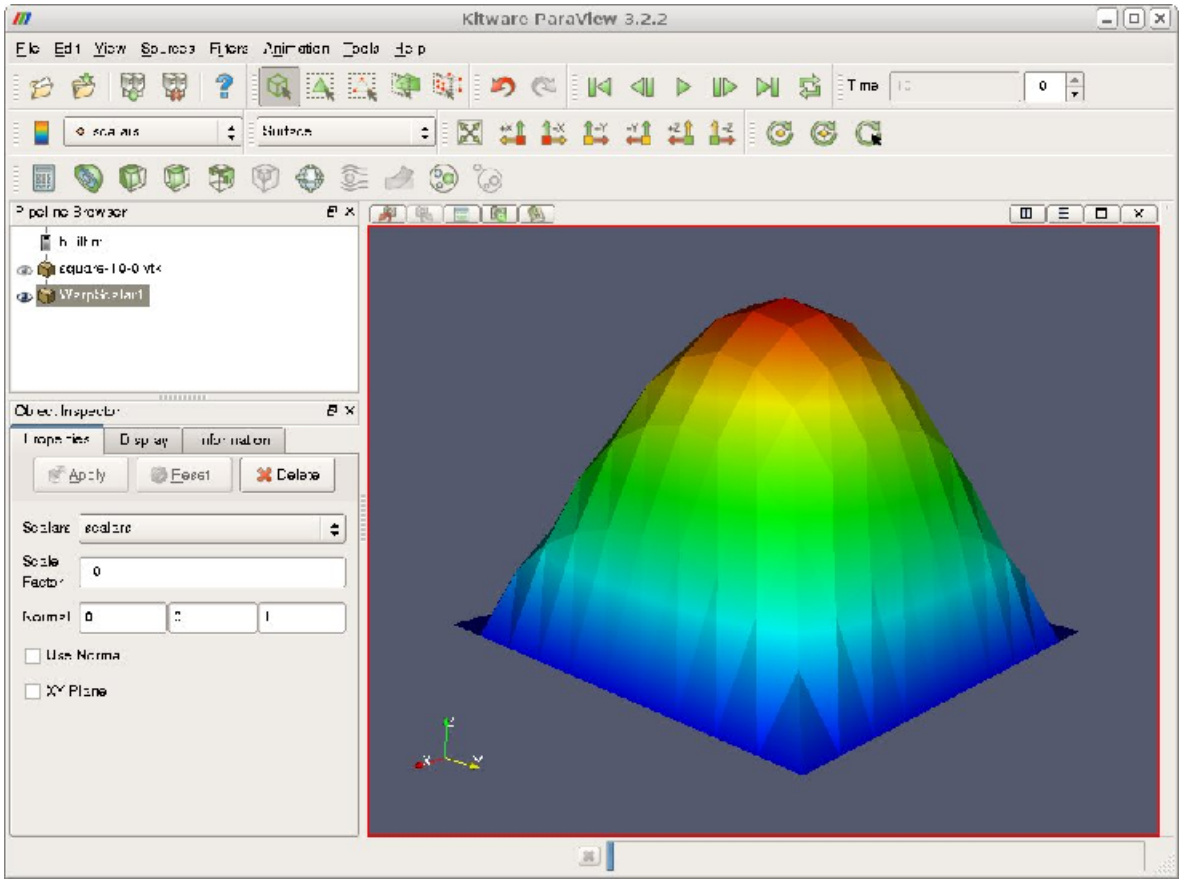


Figure 6.1: Animation of the solution of the heat problem.

We assume that the previous code is contained in the file '`heat.cc`'. Then, compile the program as usual (see page 18):

```
make heat
```

For a one dimensional problem, enter the commands:

```
mkgeo_grid -e 100 > line.geo
./heat line.geo > line.branch
```

The previous commands solve the problem for the corresponding mesh and write the solution in the field-family file format '`.branch`'. For a bidimensional one:

```
mkgeo_grid -t 10 > square.geo
./heat square.geo > square.branch
```

For a tridimensional one:

```
mkgeo_grid -T 10 > box.geo
./heat box.geo > box.branch
```

How to run the animation

```
branch line.branch -gnuplot -umax 0.125
```

A `gnuplot` window appears. Enter `q` to exit the window. For a bidimensional case, a more sophisticated procedure is required. Enter the following unix commands:

```
branch square.branch -paraview
paraview &
```

A window appears, that looks like a video player. Then, open the `File->open` menu and load `square-..vtk`. The first '.' stands for a wildcard, i.e. the time index family. Then, press the apply green button and, click a first time on the video play button, at the top of the window. Next, go to the `object inspector window`, select `display` and click on the re-scale to data range button. Then click a second time on the video play button. An elevation view can be also obtained: Select the `Filter->alphabetical->wrap(scalar)` menu, choose 10 as `scale factor` and press the `apply` green button. Then, click on the graphic window, rotate the view and finally re-play the animation

To generate an animation file[1], go to the `File->save animation` menu and enter as file name `square` and as file type `jpeg`. A collection of `jpeg` files are generated by `paraview`. Then, run the unix command:

```
ffmpeg -r 2 -i 'square.%04d.jpg' square.mov
```

The animation file `square.mov` can now be started from any video player, such as `vlc`:

```
vlc --loop square.mov
```

For the tridimensional case, the animation feature is similar.

6.2 The convection-diffusion problem

Formulation

Let $T > 0$ and $\nu > 0$. The convection-diffusion problem writes:

(P): *find* ϕ*, defined in* $\Omega\times]0,T[$*, such that*

$$\begin{aligned} \frac{\partial\phi}{\partial t} + \mathbf{u}.\nabla\phi - \nu\Delta\phi + \sigma\phi &= 0 \ \text{ in } \Omega\times]0,T[\\ \phi(0) &= \phi_0 \ \text{ in } \Omega \\ \phi(t) &= \phi_\Gamma(t) \ \text{ on } \partial\Omega\times]0,T[\end{aligned}$$

where $\mathbf{u}$, $\sigma \geqslant 0$, ϕ_0 and ϕ_Γ being known. Notice the additional $\mathbf{u}.\nabla$ operator.

Time approximation

This problem is approximated by the following first-order implicit Euler scheme:

$$\frac{\phi^{n+1} - \phi^n \circ X^n}{\Delta t} - \nu\Delta\phi^{n+1} + \sigma\phi^{n+1} = 0 \ \text{ in } \Omega$$

[1] At this time, the `avi` output feature is broken in `paraview`, and an alternate `mpeg` output is here suggested.

where $\Delta t > 0$, $\phi^n \approx \phi(n\Delta t)$ and $\phi^{(0)} = \phi_0$.

Let $t_n = n\Delta t$, $n \geqslant 0$. The term $X^n(x)$ is the position at t_n of the particle that is in x at t_{n+1} and is transported by $\mathbf{u}^n$. Thus, $X^n(x) = X(t_n, x)$ where $X(t, x)$ is the solution of the differential equation

$$\left\{ \begin{array}{rcl} \dfrac{\mathrm{d}X}{\mathrm{d}t} & = & \mathbf{u}(X(t,x),t) \quad p.p. \quad t \in]t_n, t_{n+1}[, \\ X(t_{n+1}, x) & = & x. \end{array} \right.$$

Then $X^n(x)$ is approximated by the first-order Euler approximation

$$X^n(x) \approx x - \Delta t\, \mathbf{n}^n(x).$$

This algorithm has been introduced by O. Pironneau (see e.g. [35]), and is known as the method of characteristic in the finite difference context and as the Lagrange-Galerkin in the finite element one. The efficient evaluation of $\phi_h \circ X^n(x)$ in an unstructured mesh involves a hierarchical d-tree (quadtree, octree) data structure for the localization of the element K of the mesh that contains x. When $d = 3$ requires also sophisticated geometric predicates to test whether $x \in K$ without rounding errors, and avoid to conclude that no elements contains a point x close to ∂K up to rounding errors. This problems is addressed in **Rheolef** based on the `cgal` library.

The following code implements the classical rotating Gaussian hill test case (see e.g. [40]).

Example file 6.2: `convect.cc`

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
#include "rotating-hill.h"
int main (int argc, char **argv) {
  environment rheolef (argc,argv);
  geo omega (argv[1]);
  string approx = (argc > 2) ? argv[2] : "P1";
  Float  nu     = (argc > 3) ? atof(argv[3]) : 1e-2;
  size_t n_max  = (argc > 4) ? atoi(argv[4]) : 50;
  size_t d = omega.dimension();
  Float delta_t = 2*acos(-1.)/n_max;
  space Vh (omega, approx, "vector");
  field uh = interpolate (Vh, u(d));
  space Xh (omega, approx);
  Xh.block ("boundary");
  field phi_h = interpolate (Xh, phi(d,nu,0));
  characteristic X (-delta_t*uh);
  quadrature_option_type qopt;
  qopt.set_family (quadrature_option_type::gauss_lobatto);
  qopt.set_order (Xh.degree());
  trial phi (Xh); test psi (Xh);
  branch event ("t","phi");
  dout << catchmark("nu") << nu << endl
       << event (0, phi_h);
  for (size_t n = 1; n <= n_max; n++) {
    Float t = n*delta_t;
    Float c1 = 1 + delta_t*phi::sigma(d,nu,t);
    Float c2 = delta_t*nu;
    form  a  = integrate (c1*phi*psi + c2*dot(grad(phi),grad(psi)), qopt);
    field lh = integrate (compose(phi_h, X)*psi, qopt);
    solver sa (a.uu());
    phi_h.set_u() = sa.solve (lh.u() - a.ub()*phi_h.b());
    dout << event (t, phi_h);
  }
}
```

Comments

The `characteristic` variable `X` implements the localizer $X^n(x)$:

```
  characteristic X (-delta_t*uh);
```

Combined with the `compose` function, it perform the composition $\phi_h \circ X^n$. The right-hand side is then computed by using the `integrate` function:

```
field lh = integrate (compose(phi_h, X)*psi, qopt);
```

Notice the additional `qopt` argument to the `integrate` function. By default, when this argument is omitted, a Gauss quadrature formulae is assumed, and the number of point is computed such that it integrate exactlty $2k+1$ polynoms, where k is the degree of polynoms in X_h. The Gauss-Lobatto quadrature formule is recommended for Lagrange-Galerkin methods. Recall that this choice of quadrature formulae guaranties inconditional stability at any polynomial order. Here, we specifies a Gauss-Lobatto quadrature formulae that should be exact for k order polynoms. The bilinear form is computed via the same quadrature formulae:

```
form  a  = integrate (c1*phi*psi + c2*dot(grad(phi),grad(psi)), qopt);
```

A test case is described in [36]: we take $\Omega =]-2,2[^d$ and $T = 2\pi$. This problem provides an example for a convection-diffusion equation and a known analytical solution:

$$\phi(t,x) = \exp\left(-\lambda t - r(t)|x - x_0(t)|^2\right)$$

where $\lambda = 4\nu t_0 \geqslant 0$ with $t_0 > 0$ and $\nu \geqslant 0$, $x_0(t)$ is the moving center of the hill and $r(t) = 1/(t_0 + 4\nu t)$. The source term is time-dependent: $\sigma(t) = \lambda - 2d\nu r(t)$ and has been adjusted such that the right-hand side is zero. The moving center of the hill $x_0(t)$ is associated to the velocity field $\mathbf{u}(t,x)$ as:

d	$\mathbf{u}(t,x)$	$x_0(t)$
1	$1/(2\pi)$	$t/(2\pi) - 1/2$
2	$(y,-x)$	$(-\cos(t)/2, \sin(t)/2)$
3	$(y,-x,0)$	$(-\cos(t)/2, \sin(t)/2, 0)$

Example file 6.3: `rotating-hill.h`

```
struct u : field_functor<u,point> {
  point operator() (const point & x) const {
    return (d == 1) ? point(u0) : point(x[1], -x[0]); }
  u (size_t d1) : d(d1), u0 (0.5/acos(Float(-1))) {}
  protected: size_t d; Float u0;
};
struct phi : field_functor<phi,Float> {
  static Float sigma(size_t d, Float nu1, Float t) {
    const Float t0 = 0.2;
    return 4*nu1/t0 - 2*d*nu1/(t0 + 4*nu1*t); }
  Float operator() (const point& x) const {
    point x0t;
    if (d == 1) { x0t = point(x0[0] + u0*t); }
    else {        x0t = point( x0[0]*cos(t) + x0[1]*sin(t),
                              -x0[0]*sin(t) + x0[1]*cos(t));
    }
    return exp(-4*nu*(t/t0) - dist2(x,x0t)/(t0+4*nu*t));
  }
  phi (size_t d1, Float nu1, Float t1) : d(d1), nu(nu1), t(t1),
    t0(0.2), u0 (0.5/acos(Float(-1))), x0(-0.5,0) {}
  protected: size_t d; Float nu, t, t0, u0; point x0;
};
```

Notice the use of a class-function `phi` for the implementation of $\phi(t)$ as a function of x. Such programming style has been introduced in the *standard template library* [30], which is a part of the standard `C++` library. By this way, for a given t, $\phi(t)$ can be interpolated as an usual function on a mesh.

How to run the program

We assume that the previous code is contained in the file '`convect.cc`'. Then, compile the program as usual (see page 18):

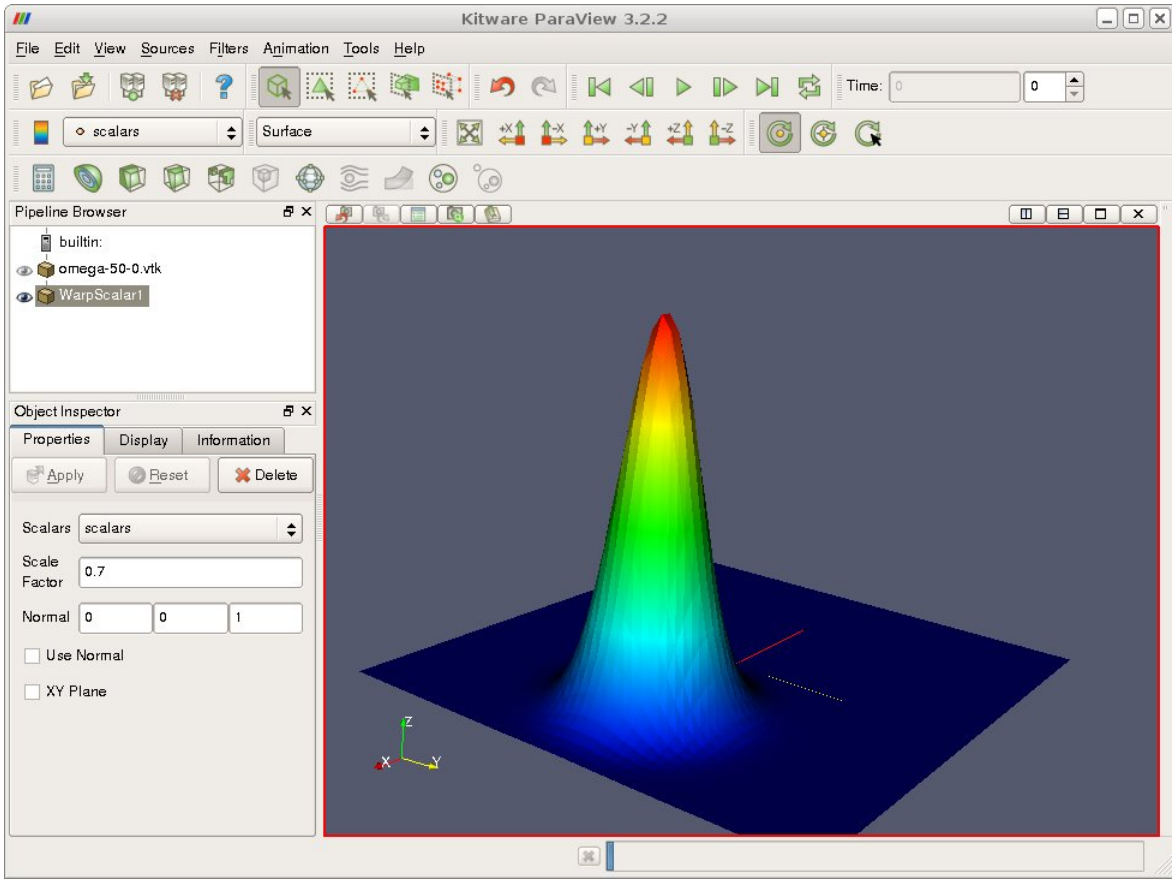


Figure 6.2: Animation of the solution of the rotating hill problem.

```
make convect
```

and enter the commands: Running the one-dimensional test case:

```
mkgeo_grid -e 500 -a -2 -b 2 > line2.geo
./convect line2.geo P1 > line2.branch
branch line2.branch -gnuplot
```

Notice the hill that moves from $x = -1/2$ to $x = 1/2$. Since the exact solution is known, it is possible to analyze the error:

Example file 6.4: `convect_error.cc`

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
#include "rotating-hill.h"
int main (int argc, char **argv) {
  environment rheolef (argc,argv);
  Float tol = (argc > 1) ? atof(argv[1]) : 1e-10;
  Float nu;
  din >> catchmark("nu") >> nu;
  branch get ("t","phi");
  branch put ("t","phi_h","pi_h_phi");
  derr << "# t\terror_l2\terror_linf" << endl;
  field phi_h;
  Float err_l2_l2 = 0;
  Float err_linf_linf = 0;
  for (Float t = 0, t_prec = 0; din >> get (t, phi_h); t_prec = t) {
    const space& Xh = phi_h.get_space();
    size_t d = Xh.get_geo().dimension();
    field pi_h_phi = interpolate (Xh, phi(d,nu,t));
    trial phi (Xh); test psi (Xh);
    form m = integrate (phi*psi);
    field eh = phi_h - pi_h_phi;
    Float err_l2 = sqrt(m(eh,eh));
    Float err_linf = eh.max_abs();
    err_l2_l2 += sqr(err_l2)*(t - t_prec);
    err_linf_linf = max(err_linf_linf, err_linf);
    dout << put (t, phi_h, pi_h_phi);
    derr << t << "\t" << err_l2 << "\t" << err_linf << endl;
  }
  derr << "# error_l2_l2     = " << sqrt(err_l2_l2) << endl;
  derr << "# error_linf_linf = " << err_linf_linf << endl;
  return (err_linf_linf <= tol) ? 0 : 1;
}
```

The numerical error $\phi_h - \pi_h(\phi)$ is computed as:

```
  field pi_h_phi = interpolate (Xh, phi(d,nu,t));
  field eh = phih - pi_h_phi;
```

and its L^2 norm is printed on the standard error. Observe the use of the `branch` class as both input and output field stream.

```
make convect_error
./convect_error < line2.branch > line2-cmp.branch
branch line2-cmp.branch -gnuplot
```

The instantaneous $L^2(\Omega)$ norm is printed at each time step and the total error in $L^2(]0,T[;L^2(\Omega))$ is finally printed at the end of the stream.

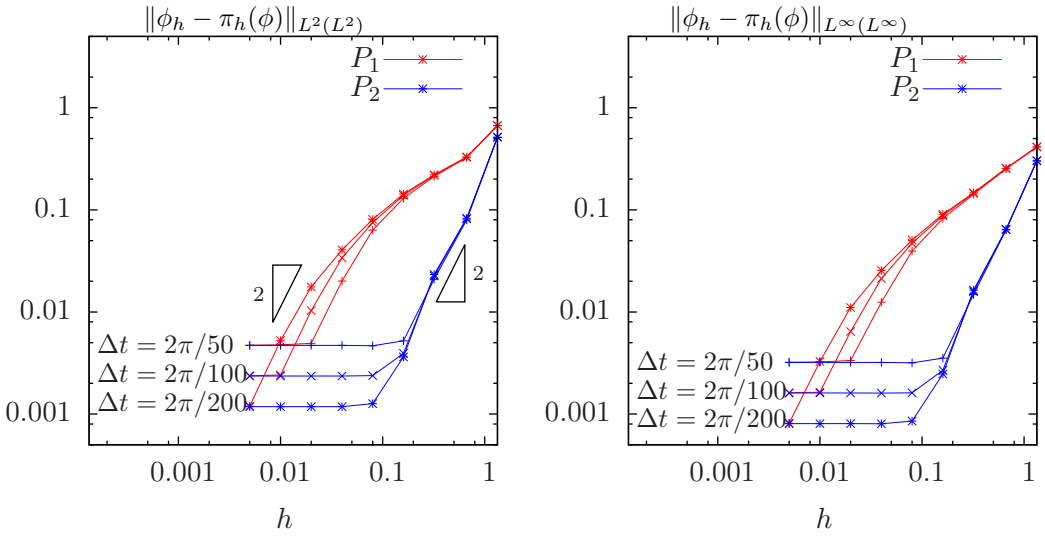


Figure 6.3: Diffusion-convection when $d = 1$ and $\nu = 10^{-2}$: convergence versus h and Δt for P_1 and P_2 elements: (left) in $L^2(L^2)$ norm; (right) in $L^\infty(L^\infty)$ norm.

A `P2` approximation can be used as well:

```
./convect line2.geo P2 > line2.branch
branch line2.branch -gnuplot
./convect_error < line2.branch > line2-cmp.branch
```

On Fig. 6.3.left we observe the $L^2(L^2)$ convergence versus h for the P_1 and P_2 elements when $d = 1$: the errors reaches a plateau that decreases versus Δt. On Fig. 6.3.right the $L^\infty(L^\infty)$ norm of the error presents a similar behavior. Since the plateau are equispaced, the convergence versus Δt is of first order.

These computation was performed for a convection-diffusion problem with $\nu = 10^{-2}$. The pure transport problem ($\nu = 0$, without diffusion) computation is obtained by:

```
./convect line2.geo P1 0 > line2.branch
branch line2.branch -gnuplot
```

Let us turn to the two-dimensional test case:

```
mkgeo_grid -t 80 -a -2 -b 2 -c -2 -d 2 > square2.geo
./convect square2.geo P1 > square2.branch
branch square2.branch -paraview
paraview &
```

The visualization and animation are similar to those of the head problem previously presented in paragraph 6.1. Observe the rotating hill. The result is shown on Fig. 6.2. The error analysis writes:

```
./convect_error < square2.branch > square2-cmp.branch
branch square2-cmp.branch -paraview
```

From the paraview menu, you can visualize simultaneously both the approximate solution and the Lagrange interpolate of the exact one. Finally, the three-dimensional case:

```
mkgeo_grid -T 15 -a -2 -b 2 -c -2 -d 2 -f -2 -g 2 > cube2.geo
./convect cube2.geo P1 > cube2.branch
```

The visualization is similar to the two-dimensional case.

6.3 The Navier-Stokes problem

Formulation

This longer example combines most functionalities presented in the previous examples.
Let us consider the Navier-Stokes problem for the driven cavity in $\Omega =]0,1[^d$, $d = 2, 3$. Let $Re > 0$ be the Reynolds number, and $T > 0$ a final time. The problem writes:

(NS): *find* $\mathbf{u} = (u_0, \ldots, u_{d-1})$ *and* p *defined in* $\Omega\times]0,T[$ *such that:*

$$
\begin{array}{rcrcl}
Re\left(\dfrac{\partial \mathbf{u}}{\partial t} + \mathbf{u}.\nabla\mathbf{u}\right) & - \mathbf{div}(2D(\mathbf{u})) & + \quad \boldsymbol{\nabla} p & = & 0 \text{ in } \Omega\times]0,T[, \\
& - \operatorname{div} \mathbf{u} & & = & 0 \text{ in } \Omega\times]0,T[, \\
& \mathbf{u}(t=0) & & = & 0 \text{ in } \Omega \times \{0,T\}, \\
& \mathbf{u} & & = & (1,0) \text{ on } \Gamma_{\text{top}}\times]0,T[, \\
& \mathbf{u} & & = & 0 \text{ on } (\Gamma_{\text{left}} \cup \Gamma_{\text{right}} \cup \Gamma_{\text{bottom}})\times]0,T[, \\
& \dfrac{\partial u_0}{\partial \mathbf{n}} = \dfrac{\partial u_1}{\partial \mathbf{n}} & = & = & u_2 = 0 \text{ on } (\Gamma_{\text{back}} \cup \Gamma_{\text{front}})\times]0,T[\text{ when } d = 3,
\end{array}
$$

where $D(\mathbf{u}) = (\boldsymbol{\nabla}\mathbf{u} + \boldsymbol{\nabla}\mathbf{u}^T)/2$. This nonlinear problem is the natural extension of the linear Stokes problem, as presented in paragraph 6.3, page 86. The boundaries are represented on Fig. 4.1, page 48.

Time approximation

Let $\Delta t > 0$. Let us consider the following backward second order scheme, for all $\phi \in C^2([0,T])$:

$$
\frac{\mathrm{d}\phi}{\mathrm{d}t}(t) = \frac{3\phi(t) - 4\phi(t-\Delta t) + \phi(t-2\Delta t)}{2\Delta t} + \mathcal{O}(\Delta t^2)
$$

The problem is approximated by the following second-order implicit scheme (BDF2):

$$
\begin{array}{rcrcl}
Re\dfrac{3\mathbf{u}^{n+1} - 4\mathbf{u}^n \circ X^n + \mathbf{u}^{n-1} \circ X^{n-1}}{2\Delta t} & - \mathbf{div}(2D(\mathbf{u}^{n+1})) & + \quad \boldsymbol{\nabla} p^{n+1} & = & 0 \text{ in } \Omega, \\
& - \operatorname{div} \mathbf{u}^{n+1} & & = & 0 \text{ in } \Omega, \\
& \mathbf{u}^{n+1} & & = & (1,0) \text{ on } \Gamma_{\text{top}}, \\
& \mathbf{u}^{n+1} & & = & 0 \text{ on } \Gamma_{\text{left}} \cup \Gamma_{\text{right}} \cup \Gamma_{\text{bottom}}, \\
& \dfrac{\partial u_0^{n+1}}{\partial \mathbf{n}} = \dfrac{\partial u_1^{n+1}}{\partial \mathbf{n}} & = & u_2^{n+1} & = 0 \text{ on } \Gamma_{\text{back}} \cup \Gamma_{\text{front}} \text{ when } d = 3,
\end{array}
$$

where, following [7, 15]:

$$
\begin{aligned}
X^n(x) &= x - \Delta t\, \mathbf{u}^*(x) \\
X^{n-1}(x) &= x - 2\Delta t\, \mathbf{u}^*(x) \\
\mathbf{u}^* &= 2\mathbf{u}^n - \mathbf{u}^{n-1}
\end{aligned}
$$

It is a second order extension of the method previously introduced in paragraph 6.2 page 80. The scheme defines a second order recurrence for the sequence $(\mathbf{u}^n)_{n\geqslant -1}$, that starts with $\mathbf{u}^{-1} = \mathbf{u}^0 = 0$.

Variational formulation

The variational formulation of this problem expresses:

$(NS)_{\Delta t}$: *find* $\mathbf{u}^{n+1} \in \mathbf{V}(1)$ *and* $p^{n+1} \in L^2_0(\Omega)$ *such that:*

$$
\begin{array}{rcrcll}
a(\mathbf{u}^{n+1}, \mathbf{v}) & + \quad b(\mathbf{v}, p^{n+1}) & = & m(\mathbf{f}^n, \mathbf{v}), & \forall \mathbf{v} \in \mathbf{V}(0), \\
b(\mathbf{u}^{n+1}, q) & & = & 0, & \forall q \in L^2_0(\Omega),
\end{array}
$$

where

$$\mathbf{f}^n = \frac{Re}{2\Delta t}\left(4\,\mathbf{u}^n \circ X^n - \mathbf{u}^{n-1} \circ X^n\right)$$

and

$$a(\mathbf{u},\mathbf{v}) \quad = \quad \frac{3Re}{2\Delta t}\int_\Omega \mathbf{u}.\mathbf{v}\,dx + \int_\Omega 2D(\mathbf{u}) : D(\mathbf{v})\,dx$$

and $b(.,.)$ and $\mathbf{V}(\alpha)$ was already introduced in paragraph 4.4, page 57, while studying the Stokes problem.

Space approximation

The Taylor-Hood [22] finite element approximation of this generalized Stokes problem was also considered in paragraph 4.4, page 57. We introduce a mesh $\mathcal{T}_h$ of Ω and the finite dimensional spaces $\mathbf{X}_h$, $\mathbf{V}_h(\alpha)$ and Q_h. The approximate problem writes:

$(NS)_{\Delta t,h}$: *find* $\mathbf{u}_h^{n+1} \in \mathbf{V}_h(1)$ *and* $p^{n+1} \in Q_h$ *such that:*

$$\begin{array}{lcccll} a(\mathbf{u}_h^{n+1},\mathbf{v}) & + & b(\mathbf{v},p_h^{n+1}) & = & m(\mathbf{f}_h^n,\mathbf{v}), & \forall \mathbf{v} \in \mathbf{V}_h(0), \\ b(\mathbf{u}_h^{n+1},q) & & & = & 0, & \forall q \in Q_h. \end{array} \tag{6.1}$$

where

$$\mathbf{f}_h^n = \frac{Re}{2\Delta t}\left(4\,\mathbf{u}_h^n \circ X^n - \mathbf{u}_h^{n-1} \circ X^n\right)$$

The problem reduces to a sequence resolution of a generalized Stokes problems.

Example file 6.5: `navier_stokes_solve.icc`

```
using namespace std;
int navier_stokes_solve (
    Float Re, Float delta_t, field l0h, field& uh, field& ph,
    size_t& max_iter, Float& tol, odiststream *p_derr=0) {
  const space& Xh = uh.get_space();
  const space& Qh = ph.get_space();
  string label = "navier-stokes-" + Xh.get_geo().name();
  quadrature_option_type qopt;
  qopt.set_family(quadrature_option_type::gauss_lobatto);
  qopt.set_order(Xh.degree());
  trial u (Xh), p (Qh);
  test  v (Xh), q (Qh);
  form mp = integrate (p*q, qopt);
  form m  = integrate (dot(u,v), qopt);
  form a  = integrate (2*ddot(D(u),D(v)) + 1.5*(Re/delta_t)*dot(u,v), qopt);
  form b  = integrate (-div(u)*q, qopt);
  solver sa (a.uu());
  solver_abtb stokes (a.uu(), b.uu(), mp.uu());
  if (p_derr != 0) *p_derr << "[" << label << "] #n |du/dt|" << endl;
  field uh1 = uh;
  for (size_t n = 0; true; n++) {
    field uh2 = uh1;
    uh1  = uh;
    field uh_star = 2.0*uh1 - uh2;
    characteristic X1 (    -delta_t*uh_star);
    characteristic X2 (-2.0*delta_t*uh_star);
    field l1h = integrate (dot(compose(uh1,X1),v), qopt);
    field l2h = integrate (dot(compose(uh2,X2),v), qopt);
    field lh  = l0h + (Re/delta_t)*(2*l1h - 0.5*l2h);
    stokes.solve (lh.u() - a.ub()*uh.b(), -(b.ub()*uh.b()),
                  uh.set_u(), ph.set_u());
    field duh_dt = (3*uh - 4*uh1 + uh2)/(2*delta_t);
    Float residual = sqrt(m(duh_dt,duh_dt));
    if (p_derr != 0) *p_derr << "[" << label << "] "<< n << " " << residual << endl;
    if (residual < tol) {
      tol = residual;
      max_iter = n;
      return 0;
    }
    if (n == max_iter-1) {
      tol = residual;
      return 1;
    }
  }
}
```

Comments

The `navier_stokes_solve` function is similar to the '`stokes_cavity.cc`'. It solves here a generalized Stokes problem and manages a right-hand side $\mathbf{f}_h$:

```
  characteristic X1 (    -delta_t*uh_star);
  characteristic X2 (-2.0*delta_t*uh_star);
  field l1h = integrate (compose(uh1,X1)*v, qopt);
  field l2h = integrate (compose(uh2,X2)*v, qopt);
  field lh  = l0h + (Re/delta_t)*(2*l1h - 0.5*l2h);
```

This last computation is similar to those done in the '`convect.cc`' example. The generalized Stokes problem is solved by the `solver_abtb` class. The stopping criterion is related to the stationary solution or the maximal iteration number.

Example file 6.6: `navier_stokes_cavity.cc`

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
#include "navier_stokes_solve.icc"
#include "navier_stokes_criterion.icc"
#include "cavity.icc"
int main (int argc, char**argv) {
  environment rheolef (argc, argv);
  if (argc < 2) {
    cerr << "usage: " << argv[0] << " <geo> <Re> <err> <n_adapt>" << endl;
    exit (1);
  }
  geo    omega (argv[1]);
  adapt_option_type options;
  Float  Re       = (argc > 2) ? atof(argv[2]) : 100;
  options.err     = (argc > 3) ? atof(argv[3]) : 1e-2;
  size_t n_adapt  = (argc > 4) ? atoi(argv[4]) : 5;
  Float  delta_t  = 0.05;
  options.hmin    = 0.004;
  options.hmax    = 0.1;
  space Xh = cavity_space (omega, "P2");
  space Qh (omega, "P1");
  field uh = cavity_field (Xh, 1.0);
  field ph (Qh, 0);
  field fh (Xh, 0);
  for (size_t i = 0; true; i++) {
    size_t max_iter = 1000;
    Float tol = 1e-5;
    navier_stokes_solve (Re, delta_t, fh, uh, ph, max_iter, tol, &derr);
    odiststream o (omega.name(), "field");
    o << catchmark("Re") << Re << endl
      << catchmark("delta_t") << delta_t << endl
      << catchmark("u")  << uh
      << catchmark("p")  << ph;
    o.close();
    if (i >= n_adapt) break;
    field ch = navier_stokes_criterion (Re,uh);
    omega = adapt (ch, options);
    o.open (omega.name(), "geo");
    o << omega;
    o.close();
    Xh = cavity_space (omega, "P2");
    Qh = space (omega, "P1");
    uh = cavity_field (Xh, 1.0);
    ph = field (Qh, 0);
    fh = field (Xh, 0);
  }
}
```

Example file 6.7: `navier_stokes_criterion.icc`

```
field navier_stokes_criterion (Float Re, const field& uh) {
  space T0h (uh.get_geo(), "P1d");
  return interpolate (T0h, sqrt(Re*norm2(uh) + 4*norm2(D(uh))));
}
```

Comments

The code performs a computation by using adaptive mesh refinement, in order to capture recirculation zones. The `adapt_option_type` declaration is used by `rheolef` to send options to the mesh generator. The code reuse the file '`cavity.icc`' introduced page 58. This file contains two functions that defines boundary conditions associated to the cavity driven problem.

The `criteria` function computes the adaptive mesh refinement criteria:

$$c_h = (Re|\mathbf{u}_h|^2 + 2|D(\mathbf{u}_h)|^2)^{1/2}$$

The `criteria` function is similar to those presented in the '`embankment_adapt.cc`' example.

How to run the program

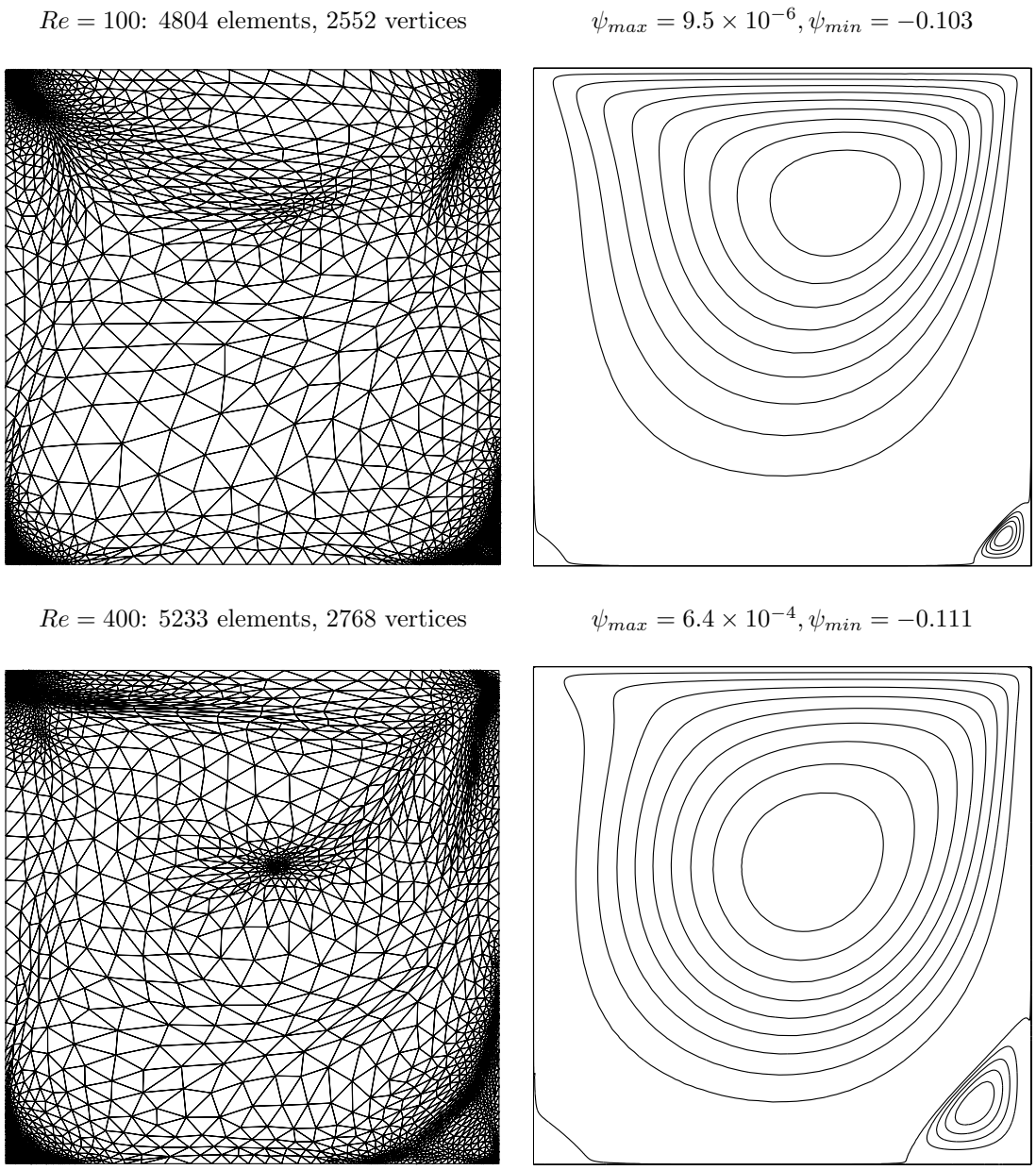


Figure 6.4: Meshes and stream functions associated to the solution of the Navier-Stokes equations for $Re = 100$ (top) and $Re = 400$ (bottom).

$Re = 1000$: 5873 elements, 3106 vertices $\psi_{max} = 1.64 \times 10^{-3}, \psi_{min} = -0.117$

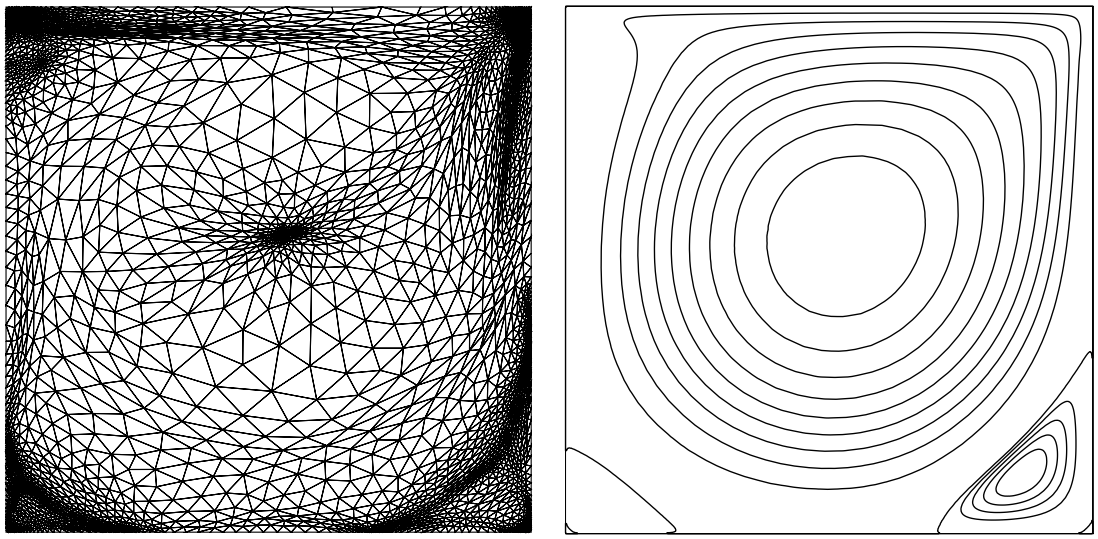


Figure 6.5: Meshes and stream functions associated to the solution of the Navier-Stokes equations for $Re = 1000$.

The mesh loop adaptation is initiated from a `bamg` mesh (see also appendix B.1).

```
bamg -g square.bamgcad -o square.bamg
bamg2geo square.bamg square.dmn > square.geo
```

Then, compile and run the Navier-Stokes solver for the driven cavity for $Re = 100$:

```
make navier_stokes_cavity
./navier_stokes_cavity square.geo 100
```

The program performs a computation with $Re = 100$. By default the time step is $\Delta t = 0.05$ and the computation loops for five mesh adaptations. At each time step, the program prints an approximation of the time derivative, and stops when a stationary solution is reached. Then, we visualize the '`square-5.geo`' adapted mesh and its associated solution:

```
geo   square-5.geo
field square-5.field.gz -velocity -scale 4 -mayavi
```

Notice the `-scale` option that applies a multiplicative factor to the arrow length when plotting. The representation of the stream function writes:

```
make streamf_cavity
zcat square-5.field.gz | ./streamf_cavity | field -bw -n-iso-negative 10 -
```

The programs '`streamf_cavity.cc`', already introduced page 63, is here reused. The last options of the `field` program draws isocontours of the stream function using lines, as shown on Fig. 6.4. The zero isovalue separates the main flow from recirculations, located in corners at the bottom of the cavity.

For $Re = 400$ and 1000 the computation writes:

```
./navier_stokes_cavity square.geo  400
./navier_stokes_cavity square.geo 1000
```

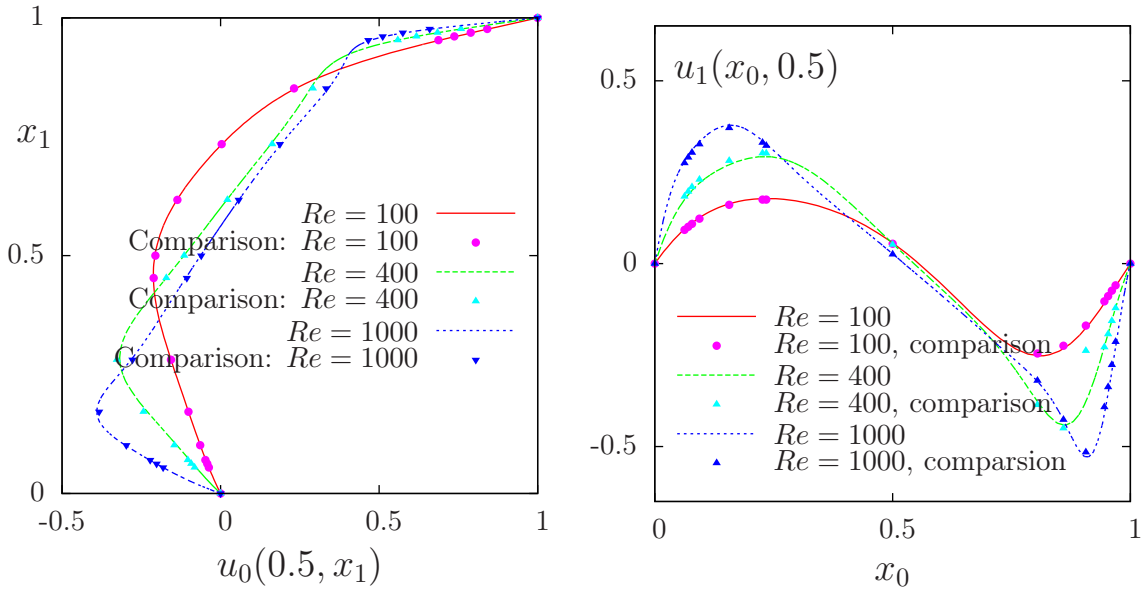


Figure 6.6: Navier-Stokes: velocity profiles along lines passing thought the center of the cavity, compared with data from [18]: (a) u_0 along the vertical line; (b) u_1 along the horizontal line line.

The visualization of the cut of the horizontal velocity along the vertical median line writes:

```
field square-5.field.gz -comp 0 -cut -normal -1 0 -origin 0.5 0
field square-5.field.gz -comp 1 -cut -normal  0 1 -origin 0   0.5
```

Fig. 6.6 compare the cuts with data from [18], table 1 and 2 (see also [20]). Observe that the solution is in good agreement with these previous computations.

Re		x_c	y_c	$-\psi_{\min}$	$\psi_{\max}$
100	present	0.613	0.738	0.103	9.5×10^{-6}
	Labeur and Wells [26]	0.608	0.737	0.104	-
	Donea and Huerta [13]	0.62	0.74	0.103	-
400	present	0.554	0.607	0.111	5.6×10^{-4}
	Labeur and Wells [26]	0.557	0.611	0.115	-
	Donea and Huerta [13]	0.568	0.606	0.110	-
1000	present	0.532	0.569	0.117	1.6×10^{-3}
	Labeur and Wells [26]	0.524	0.560	0.121	-
	Donea and Huerta [13]	0.540	0.573	0.110	-

Figure 6.7: Cavity flow: primary vortex position and stream function value.

Finally, table 6.7 compares the primary vortex position and its associated stream function value. Notice also the good agreement with previous simulations. The stream function extremal values are obtained by:

```
zcat square-5.field.gz | ./streamf_cavity | field -min -
zcat square-5.field.gz | ./streamf_cavity | field -max -
```

The maximal value has not yet been communicated to our knowledge and is provided in table 6.7 for cross validation purpose. The small program that computes the primary vortex position is showed below.

```
make vortex_position
zcat square-5.field.gz | ./streamf_cavity | ./vortex_position
```

Example file 6.8: `vortex_position.cc`

```
#include "rheolef.h"
using namespace rheolef;
int main (int argc, char** argv) {
  environment rheolef (argc, argv);
  check_macro (communicator().size() == 1, "please, use sequentially");
  field psi_h;
  din >> psi_h;
  size_t idof_min = 0;
  Float psi_min = std::numeric_limits<Float>::max();
  for (size_t idof = 0, ndof = psi_h.ndof(); idof < ndof; idof++) {
    if (psi_h.dof(idof) >= psi_min) continue;
    psi_min = psi_h.dof(idof);
    idof_min = idof;
  }
  const array<point>& xdof = psi_h.get_space().get_xdofs();
  point xmin = xdof [idof_min];
  dout << "xc\t\tyc\t\tpsi" << std::endl
       << xmin[0] << "\t" << xmin[1] << "\t" << psi_min << std::endl;
}
```

For higher Reynolds number, Shen [47] showed in 1991 that the flow converges to a stationary state for Reynolds numbers up to 10 000; for Reynolds numbers larger than a critical value $10\,000 < Re_1 < 10\,500$ and less than another critical value $15\,000 < Re_2 < 16\,000$, these authors founded that the flow becomes periodic in time which indicates a Hopf bifurcation; the flow loses time periodicity for $Re \geq Re_2$. In 1998, Ould Salihi [33] founded a loss of stationarity between 10 000 and 20 000. In 2002, Auteri et al. [4] estimated the critical value for the apparition of the first instability to $Re_1 \approx 8018$. In 2005, Erturk et al. [14] computed steady driven cavity solutions up to $Re \leqslant 21\,000$. Also in 2005, this result was infirmed by [16]: these authors estimated Re_1 close to 8000, in agreement with [4]. The 3D driven cavity has been investigated in [28] by the method of characteristic (see also [27] for 3D driven cavity computations). In conclusion, the exploration of the driven cavity at large Reynolds number is a fundamental challenge in computational fluid dynamics.

Part III

Advanced and highly nonlinear problems

Chapter 7

Equation defined on a surface

This chapter deals with equations defined on a closed hypersurface. We present three different numerical methods: the direct resolution of the problem on an explicit surface mesh generated independently of **Rheolef**, the direct resolution on a surface mesh generated by **Rheolef** from a volume mesh, and finally a level set type method based on a volume mesh in an h-narrow band containing the surface. This last method allows to define hybrid operators between surface and volume-based finite element fields. These methods are demonstrated on two model problems and two different surfaces.

Let us consider a closed surface $\Gamma \in \mathbb{R}^d$, $d = 2$ or 3 and Γ is a connected C^2 surface of dimension $d-1$ with $\partial\Gamma = 0$. We first consider the following problem:

($P1$) *find u, defined on Γ such that:*

$$u - \Delta_s u = f \text{ on } \Gamma \tag{7.1}$$

where $f \in L^2(\Gamma)$. For all function u defined on Γ, Δ_s denotes the Laplace-Beltrami operator:

$$\Delta_s u = \mathrm{div}_s(\nabla_s u)$$

where ∇_s and div_s are the tangential derivative and the surface divergence along Γ, defined respectively, for all scalar field φ and vector field $\mathbf{v}$ by:

$$\begin{aligned} \nabla_s \varphi &= (I - \mathbf{n} \otimes \mathbf{n}) \nabla \varphi \\ \mathrm{div}_s \mathbf{v} &= (I - \mathbf{n} \otimes \mathbf{n}) : \nabla \mathbf{v} \end{aligned}$$

Here, $\mathbf{n}$ denotes a unit normal on Γ.

We also consider the following variant of this problem:

($P2$) *find u, defined on Γ such that:*

$$-\Delta_s u = f \text{ on } \Gamma \tag{7.2}$$

This second problem is similar to the first one: the Helmholtz operator $I - \Delta_s$ has been replaced by the Laplace-Beltrami one $-\Delta_s$. In that case, the solution is defined up to a constant: if u is a solution, then $u + c$ is also a solution for any constant $c \in \mathbb{R}$. Thus, we refers to ($P1$) as the Helmholtz-Beltrami problem and to ($P2$) as the Laplace-Beltrami one.

7.1 Approximation on an explicit surface mesh

The Helmholtz-Beltrami problem

Tanks to the surface Green formula (see appendix A.3), the variational formulation of problem ($P1$) writes:

$(VF1)$: *find* $u \in H^1(\Gamma)$ *such that:*

$$a(u,v) = l(v),\ \forall v \in H^1(\Gamma)$$

where for all $u, v \in H^1(\Gamma)$,

$$\begin{aligned} a(u,v) &= \int_\Gamma (u\,v + \nabla_s u.\nabla_s v)\ \mathrm{d}s \\ l(v) &= \int_\Gamma f\,v\,\mathrm{d}s \end{aligned}$$

Let $k \geqslant 1$ and consider a k-th order curved surface finite element mesh Γ_h of Γ. We define the space W_h:

$$W_h = \left\{ v_h \in H^1(\Gamma_h); v_{|S} \in P_k, \forall S \in \Gamma_h \right\}$$

The approximate problem writes:

$(VF1)_h$: *find* $u_h \in W_h$ *such that:*

$$a(u_h, v_h) = l(v_h),\quad \forall v_h \in W_h$$

Example file 7.1: `helmholtz_s.cc`

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
#include "sphere.icc"
int main(int argc, char**argv) {
  environment rheolef(argc, argv);
  geo gamma (argv[1]);
  size_t d = gamma.dimension();
  space Wh (gamma, argv[2]);
  trial u (Wh); test v (Wh);
  form  a  = integrate (u*v + dot(grad_s(u),grad_s(v)));
  field lh = integrate (f(d)*v);
  field uh (Wh);
  solver sa (a.uu());
  uh.set_u() = sa.solve (lh.u() - a.ub()*uh.b());
  dout << uh;
}
```

Comments

The problem involves the Helmholtz operator and thus, the code is similar to '`neumann-nh.cc`' presented page 32. Let us comments the only differences:

```
  form  a  = integrate (u*v + dot(grad_s(u),grad_s(v)));
```

The form refers to the `grad_s` operator instead of the `grad` one, since only the coordinates related to the surface are involved.

```
  field lh = integrate (f(d)*v);
```

The right-hand-side does not involve any boundary term, since the surface Γ is closed: the boundary domain $\partial\Gamma = \emptyset$. As test problem, the surface Γ is the unit circle when $d = 2$ and the unit sphere when $d = 3$. The data f has been chosen as in [11, p. 17]. This choice is convenient since the exact solution is known. Recall that the spherical coordinates (ρ, θ, ϕ) are defined from the artesian ones (x_0, x_1, x_2) by:

$$\rho = \sqrt{x_0^2 + x_1^2 + x_2^2},\quad \phi = \arccos\left(x_2/\rho\right),\quad \theta = \begin{cases} \arccos\left(x_0/\sqrt{x_0^2 + x_1^2}\right) & \text{when } x_1 \geqslant 0 \\ 2\pi - \arccos\left(x_0/\sqrt{x_0^2 + x_1^2}\right) & \text{otherwise} \end{cases}$$

Example file 7.2: `sphere.icc`

```
struct p : field_functor<p,Float> {
  Float operator() (const point& x) const {
    if (d == 2) return 26*(pow(x[0],5) - 10*pow(x[0],3)*sqr(x[1])
                                       + 5*x[0]*pow(x[1],4));
    else        return 3*sqr(x[0])*x[1] - pow(x[1],3);
  }
  p (size_t d1) : d(d1) {}
  protected: size_t d;
};
struct f : field_functor<f,Float> {
  Float operator() (const point& x) const {
    if (d == 2) return _p(x)/pow(norm(x),5);
    else        return alpha*_p(x);
  }
  f (size_t d1) : d(d1), _p(d1) {
    Float pi = acos(Float(-1));
    alpha = -(13./8.)*sqrt(35./pi);
  }
  protected: size_t d; p _p; Float alpha;
};
struct u_exact : field_functor<u_exact,Float> {
  Float operator() (const point& x) const {
    if (d == 2) return _f(x)/(25+sqr(norm(x)));
    else        return sqr(norm(x))/(12+sqr(norm(x)))*_f(x);
  }
  u_exact (size_t d1) : d(d1), _f(d1) {}
  protected: size_t d; f _f;
};
Float phi (const point& x) { return norm(x) - 1; }
```

How to run the program

The program compile as usual:

```
make helmholtz_s
```

A mesh of a circle is generated by:

```
mkgeo_ball -s -e 100 > circle.geo
geo circle -mayavi
```

The `mkgeo_ball` is a convenient script that generates a mesh with the `gmsh` mesh generator. Then, the problem resolution writes:

```
./helmholtz_s circle P1 > circle.field
field circle.field -mayavi
field circle.field -mayavi -elevation
```

The tridimensional case is similar:

```
mkgeo_ball -s -t 10 > sphere.geo
geo sphere.geo -mayavi -stereo
./helmholtz_s sphere.geo P1 > sphere.field
field sphere.field -mayavi -stereo
```

The solution is represented on Fig .7.1.left.

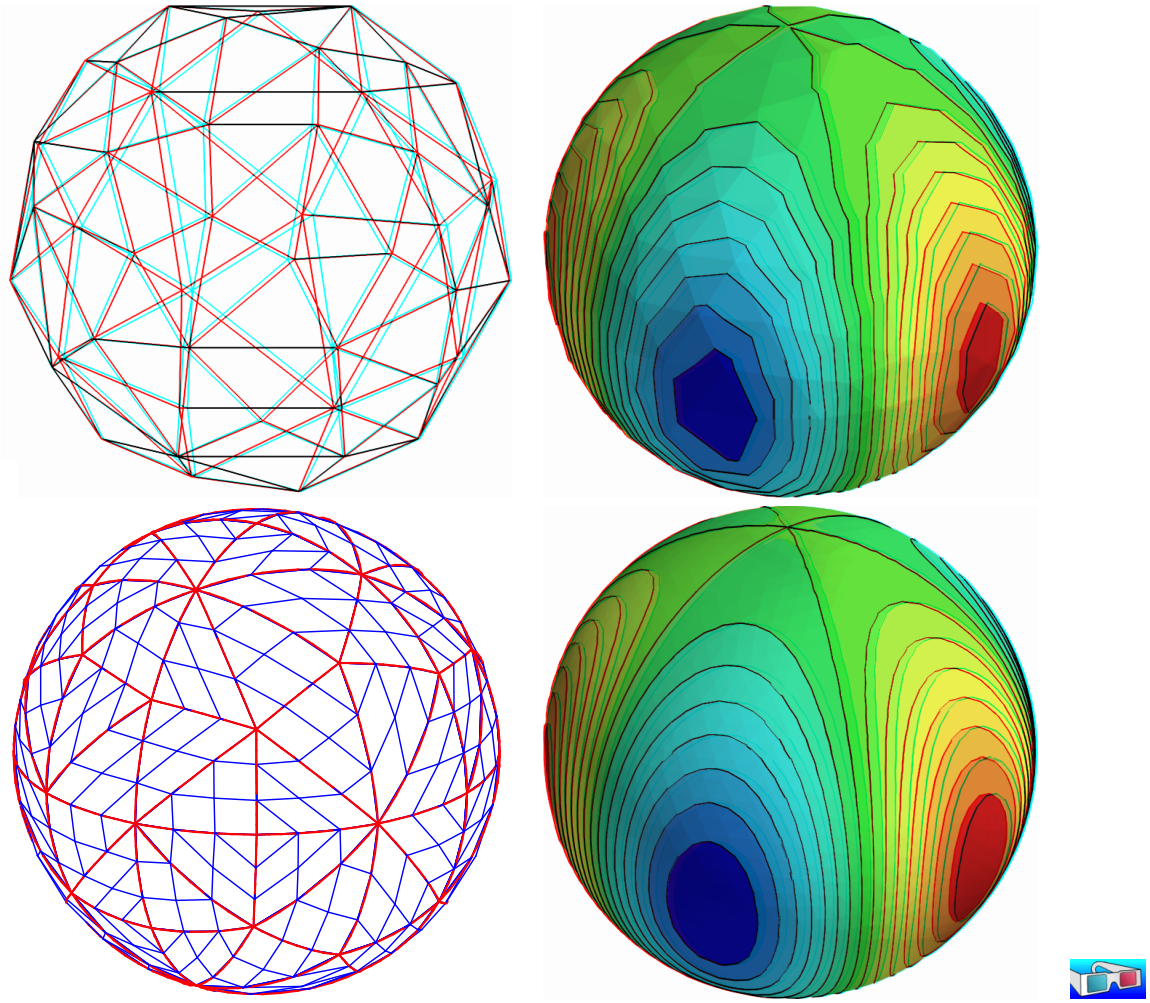


Figure 7.1: Helmholtz-Beltrami problem: high-order curved surface mesh and its corresponding isoparametric solution: (top) *order* = 1; (bottom) *order* = 3.

Higher-order isoparametric finite elements can be considered for the curved geometry:

```
mkgeo_ball -s -e 30 -order 3 > circle-P3.geo
geo circle-P3.geo -subdivide 10
```

Observe the curved edges (see Fig .7.1). The `-subdivide` option allows a graphical representation of the curved edges by subdividing each edge in ten linear parts, since graphical softwares are not yet able to represent curved elements. The computation with the P_3 isoparametric approximation writes:

```
./helmholtz_s circle-P3 P3 > circle-P3.field
field circle-P3.field -mayavi -elevation
```

Notice that both the curved geometry and the finite element are second order. The tridimensional counterpart writes simply:

```
mkgeo_ball -s -t 10 -order 3 > sphere-P3.geo
geo sphere-P3.geo
./helmholtz_s sphere-P3 P3 > sphere-P3.field
field sphere-P3.field -mayavi -stereo
```

The solution is represented on Fig .7.1).right-bottom. The graphical representation is not yet able to represent the high-order approximation: each elements is subdivided and a piecewise linear representation is used in each sub-elements.

Since the exact solution is known, the error can be computed: this is done by the program `helmholtz_s_error.cc`. This file is not presented here, as it is similar to some others examples, but can be founded in the **Rheolef** example directory. Figure 7.2 plots the error in various norms versus element size for different isoparametric approximations.

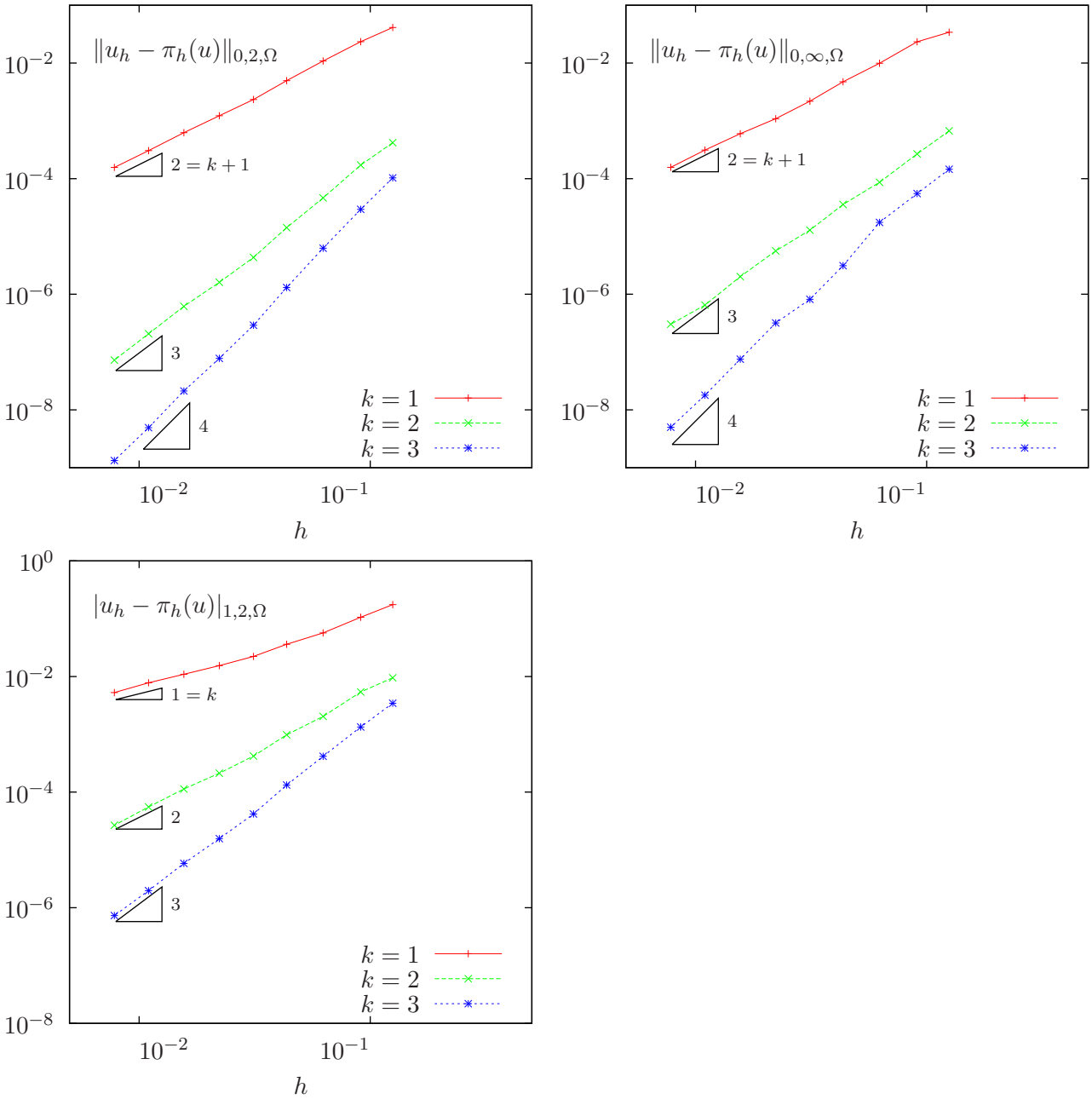


Figure 7.2: Curved non-polynomial surface: error analysis in L^2, L^∞ and H^1 norms.

The Laplace-Beltrami problem

This problem has been introduced in (7.2), page 97. While the treatment of the Helmholtz-Beltrami problem was similar to the Helmholtz problem with Neumann boundary conditions, here, the treatment of the Laplace-Beltrami problem is similar to the Laplace problem with Neumann boundary conditions: see section 2.4, page 35. Notice that for both problems, the solution is defined up to a constant. Thus, the linear problem has a singular matrix. The '`laplace_s.cc`' code is similar to the '`neumann-laplace.cc`' one, as presented in section 2.4. The only change lies one the definition of the right-hand side.

Example file 7.3: `laplace_s.cc`

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
#include "torus.icc"
int main (int argc, char**argv) {
  environment rheolef (argc, argv);
  geo gamma (argv[1]);
  size_t d = gamma.dimension();
  space Wh (gamma, argv[2]);
  trial u (Wh); test v (Wh);
  form m = integrate (u*v);
  form a = integrate (dot(grad_s(u),grad_s(v)));
  field b = m*field(Wh,1);
  field lh = integrate (f(d)*v);
  csr<Float> A = {{  a.uu(),      b.u()},
                  {trans(b.u()),  0   }};
  vec<Float> B =  {  lh.u(),      0   };
  solver sa (A);
  vec<Float> U = sa.solve (B);
  field uh(Wh);
  uh.set_u() = U [range(0,uh.u().size())];
  dout << uh;
}
```

Example file 7.4: `torus.icc`

```
static const Float R = 1;
static const Float r = 0.6;
Float phi (const point& x) {
 return sqr(sqrt(sqr(x[0])+sqr(x[1]))-sqr(R)) + sqr(x[2])-sqr(r);
}
void get_torus_coordinates (const point& x,
                            Float& rho, Float& theta, Float& phi) {
 static const Float pi = acos(Float(-1));
 rho = sqrt(sqr(x[2]) + sqr(sqrt(sqr(x[0]) + sqr(x[1])) - sqr(R)));
 phi = atan2(x[1], x[0]);
 theta = atan2(x[2], sqrt(sqr(x[0]) + sqr(x[1])) - R);
}
struct u_exact : field_functor<u_exact,Float> {
 Float operator() (const point& x) const {
  Float rho, theta, phi;
  get_torus_coordinates (x, rho, theta, phi);
  return sin(3*phi)*cos(3*theta+phi);
 }
 u_exact (size_t d=3) {}
};
struct f : field_functor<f,Float> {
 Float operator() (const point& x) const {
  Float rho, theta, phi;
  get_torus_coordinates (x, rho, theta, phi);
  Float fx = (9*sin(3*phi)*cos(3*theta+phi))/sqr(r)
    - (-10*sin(3*phi)*cos(3*theta+phi) - 6*cos(3*phi)*sin(3*theta+phi))
      /sqr(R + r*cos(theta))
    - (3*sin(theta)*sin(3*phi)*sin(3*theta+phi))
      /(r*(R + r*cos(theta)));
  return fx;
 }
 f (size_t d=3) {}
};
```

As test problem, the surface Γ is the a torus when $d = 3$. The data f has been chosen as in [32, p. 3355]. This choice is convenient since the exact solution is known. Let R and r denotes the large and small torus radii, respectively. The torus coordinates (ρ, θ, ϕ) are defined linked to the Cartesian ones by:

$$\begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = R \begin{pmatrix} \cos(\phi) \\ \sin(\phi) \\ 0 \end{pmatrix} + \rho \begin{pmatrix} \cos(\phi)\cos(\theta) \\ \sin(\phi)\cos(\theta) \\ \sin(\theta) \end{pmatrix}$$

Here ρ is the distance from the point to the circle in the x_0x_1 plane around 0 with radius R, θ is the angle from the positive $(x_0, x_1, 0)$ to x_0 and ϕ is the angle from the positive x_0 axis to $(x_0, x_1, 0)$.

How to run the program ?

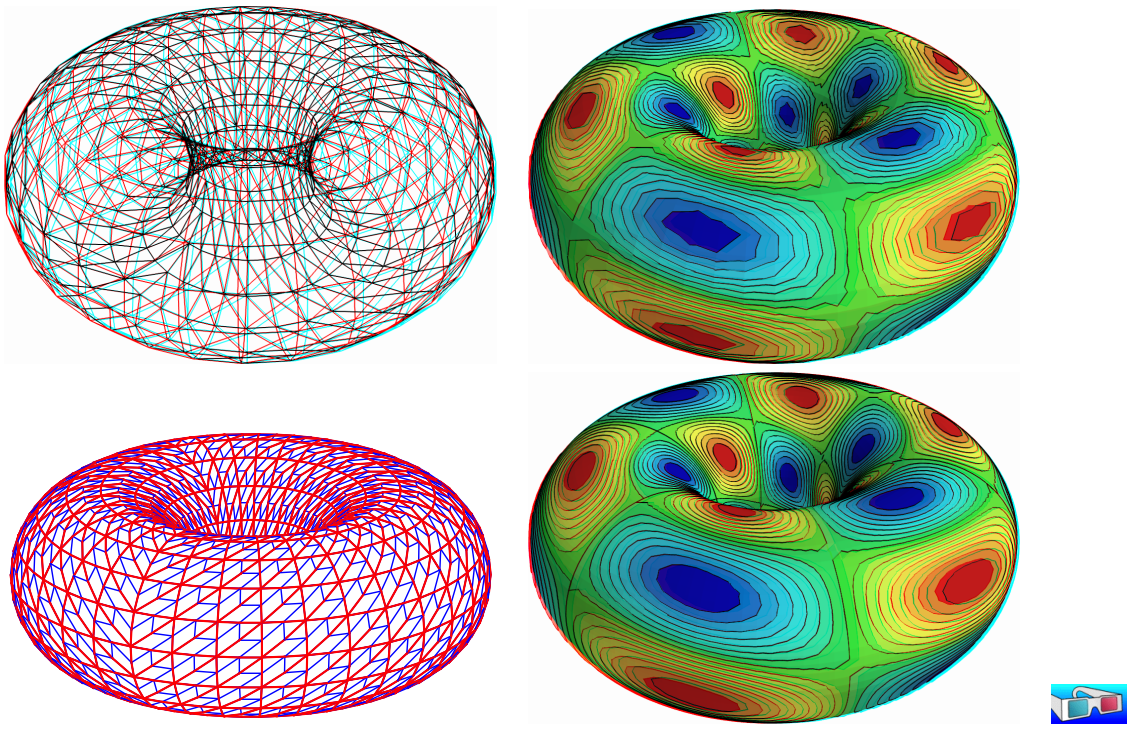


Figure 7.3: Laplace-Beltrami problem on a torus: high-order curved surface mesh and its corresponding isoparametric solution: (top) $order = 1$; (bottom) $order = 2$.

The surface mesh of the torus is generated by:

```
gmsh -2 torus.mshcad -o torus.msh
msh2geo torus.msh > torus.geo
geo torus.geo -mayavi -stereo
```

The '`torus.mshcad`' is not presented here: it can be founded in the **Rheolef** example directory. Then, the computation and visualization writes:

```
make laplace_s
./laplace_s torus.geo P1 > torus.field
field torus.field -mayavi -stereo
```

For a higher-order approximation:

```
gmsh -2 -order 2 torus.mshcad -o torus-P2.msh
msh2geo torus-P2.msh > torus-P2.geo
geo torus-P2.geo
./laplace_s torus-P2.geo P2 > torus-P2.field
field torus-P2.field -mayavi
```

The solution is represented on Fig. 7.3. By editing '`torus.mshcad`' and changing the density of discretization, we can improve the approximate solution and converge to the exact solution. Due to a bug [46] in the current gmsh version 2.5.1 the convergence is not optimal $\mathcal{O}(h^k)$ for higher values of k.

7.2 Building a surface mesh from a level set function

The previous method is limited to not-too-complex surface Γ, that can be described by a regular finite element surface mesh Γ_h. When the surface change, as in a time-dependent process, complex change of topology often occurs and the mesh Γ_h can degenerate or be too complex to be efficiently meshed. In that case, the surface is described implicitly as the zero isosurface, or zero *level set*, of a function:

$$\Gamma = \{x \in \Lambda; \quad \phi(x) = 0\}$$

where $\Lambda \subset \mathbb{R}^d$ is a bounding box of the surface Γ.

The following code automatically generates the mesh Γ_h of the surface described by the zero isosurface of a discrete $\phi_h \in X_h$ level set function:

$$\Gamma_h = \{x \in \Lambda; \quad \phi_h(x) = 0\}$$

where X_h is a piecewise affine functional space over a mesh $\mathcal{T}_h$ of Λ:

$$X_h = \{\varphi \in L^2(\Lambda) \cap C^0(\Lambda);\ \varphi_{/K} \in P_1,\ \forall K \in \mathcal{T}_h\}$$

The polynomial approximation is actually limited here to first order: building higher order curved finite element surface meshes from a level set function is planed for the future versions of **Rheolef**.

Finally, a computation, as performed in the previous paragraph can be done using Γ_h. We also point out the limitations of this approach.

Example file 7.5: `level_set_sphere.cc`

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
#include "sphere.icc"
int main (int argc, char**argv) {
  environment rheolef (argc,argv);
  geo lambda (argv[1]);
  level_set_option_type opts;
  opts.split_to_triangle
    = (argc > 2 && argv[2] == std::string("-tq")) ? false : true;
  space Xh (lambda, "P1");
  field phi_h = interpolate(Xh, phi);
  geo gamma = level_set (phi_h, opts);
  dout << gamma;
}
```

Comments

All the difficult work of building the intersection mesh Γ_h, defined as the zero level set of the ϕ_h function, is performed by the `level_set` function:

```
  geo gamma = level_set (phi_h, opts);
```

When $d = 3$, intersected tetrahedra leads to either triangular or quadrangular faces. By default, quadrangular faces are split into two triangles. An optional `-tq` program flag allows to conserve quadrangles in the surface mesh: it set the `split_to_triangle` optional field to false.

How to run the program ?

After the compilation, generates the mesh of a bounding box $\Lambda = [-2, 2]^d$ of the surface and run the program:

```
make level_set_sphere
mkgeo_grid -t 20 -a -2 -b 2 -c -2 -d 2 > square2.geo
./level_set_sphere square2.geo > circle.geo
geo circle.geo -mayavi -stereo
```

The computation of the previous paragraph can be reused:

```
./helmholtz_s circle.geo P1 | field -mayavi -stereo -
```

Notice that, while the bounding box mesh was uniform, the intersected mesh could present arbitrarily small edge length (see also Fig. 7.4):

```
geo -min-element-measure circle.geo
geo -max-element-measure circle.geo
```

Let us turn to the $d = 3$ case:

```
mkgeo_grid -T 20 -a -2 -b 2 -c -2 -d 2 -f -2 -g 2 > cube2.geo
./level_set_sphere cube2.geo | geo -upgrade - > sphere.geo
geo sphere.geo -mayavi -stereo
./helmholtz_s sphere.geo P1 | field -mayavi -stereo -
```

This approach can be extended to the Laplace-Beltrami problem on a torus:

```
sed -e 's/sphere/torus/' < level_set_sphere.cc > level_set_torus.cc
make level_set_torus
./level_set_torus cube2.geo | geo -upgrade - > torus.geo
geo torus.geo -mayavi -stereo
./laplace_s torus.geo P1 | field -mayavi -stereo -
```

While the bounding box mesh was uniform, the triangular elements obtained by intersecting the 3D bounding box mesh with the level set function can present arbitrarily irregular sizes nd shapes (see also Fig. 7.4):

```
geo -min-element-measure -max-element-measure sphere.geo
geo -min-element-measure -max-element-measure torus.geo
```

Thus, there is no theoretical guaranties for the finite element method to converge on these irregular families of meshes, despite, most of the time, the computations run well. This is the major drawback of this method.

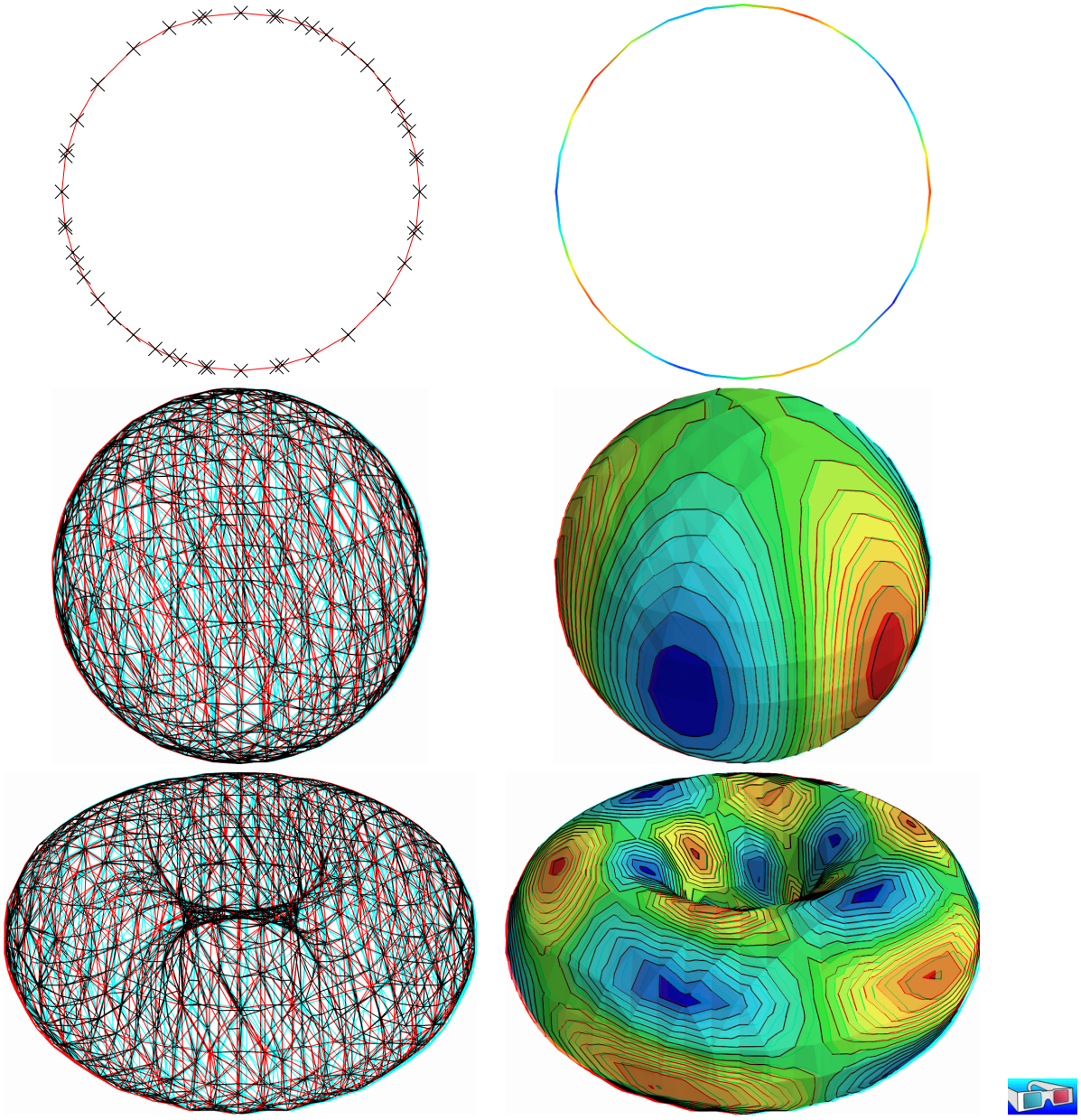


Figure 7.4: Building an explicit surface mesh from level set: (top) circle; (center) sphere; (bottom) torus.

7.3 The banded level set method

The banded level set method presents the advantages of the two previous methods without their drawback: it applies to very general geometries, as described by a level set funtion, and has theoretical fundations, as usual finite element methods. The previous drawback of the intersection mesh can be circumvented by enlarging the surface Γ_h to a band β_h containing all the intersected elements of $\mathcal{T}_h$ (see [2,12,32]):

$$\beta_h = \{K \in \mathcal{T}_h; K \cap \Gamma_h \neq \emptyset\}$$

Then, we introduce B_h the piecewise affine functional space over β_h:

$$B_h = \{v \in L^2(\beta_h) \cap C^0(\beta_h);\ v_{/K} \in P_1,\ \forall K \in \mathcal{T}_h\}$$

The problem is extended from Γ_h to β_h as:
$(VF)_h$: *find* $u_h \in B_h$ *such that:*

$$a(u_h, v_h) = l(v_h),\ \forall v_h \in B_h$$

where, for all $u, v \in B_h$,

$$\begin{aligned} a(u,v) &= \int_{\Gamma_h} (u\,v + \nabla_s u.\nabla_s v)\ \mathrm{d}s \\ l(v) &= \int_{\Gamma_h} f\,v\,\mathrm{d}s \end{aligned}$$

for all $u_h, v_h \in B_h$. Notice that while u_h and v_h are defined over β_h, the summations in the variational formulations are restricted only to $\Gamma_h \subset \beta_h$.

Example file 7.6: `helmholtz_band_iterative.cc`

```
#include "rheolef.h"
using namespace std;
using namespace rheolef;
#include "sphere.icc"
int main (int argc, char**argv) {
  environment rheolef(argc, argv);
  geo lambda (argv[1]);
  size_t d = lambda.dimension();
  space Xh (lambda, "P1");
  field phi_h = interpolate(Xh, phi);
  band gamma_h (phi_h);
  space Bh (gamma_h.band(), "P1");
  trial u (Bh); test v (Bh);
  form  a  = integrate (gamma_h, u*v + dot(grad_s(u),grad_s(v)));
  field lh = integrate (gamma_h, f(d)*v);
  field uh (Bh,0);
  size_t max_iter = 10000;
  Float tol = 1e-10;
  pminres (a.uu(), uh.set_u(), lh.u(), eye(), max_iter, tol, &derr);
  dout << catchmark("phi") << phi_h
       << catchmark("u")   << uh;
}
```

Comments

The band is build directly from the level set function as:

```
  band gamma_h (phi_h);
```

The band structure is a small class that groups the surface mesh Γ_h, available as `gamma_h.level_set()`, and the β_h mesh, available as `gamma_h.band()`. It also manages some

correspondance between both meshes. Then, the space of piecewise affine functions over the band is introduced:

```
space Bh (gamma_h.band(), "P1");
```

Next, two forms are computed by using the `integrate` function, with the band `gamma_h` as a domain-like argument:

```
form m = integrate (gamma_h, u*v);
form a = integrate (gamma_h, dot(grad_s(u),grad_s(v)));
```

The right-hand side also admits the `gamma_h` argument:

```
field lh = integrate (gamma_h, f(d)*v);
```

Recall that summations for both forms and right-hand side will be performed on Γ_h, represented by `gamma_h.level_set()`, while the approximate functional space is B_h. Due to this summation on Γ_h instead of β_h, the matrix of the system is singular [2, 31, 32] and the MINRES algorithm has been chosen to solve the linear system:

```
pminres (a.uu(), uh.set_u(), lh.u(), eye(), max_iter, tol, &derr);
```

The `eye()` argument represents here the identity preconditioner, i.e. no preconditioner at all. It has few influence of the convergence properties of the matrix and could be replaced by another simple one: the diagonal of the matrix `diag(a.uu())` without sensible gain of performance:

```
pminres (a.uu(), uh.set_u(), lh.u(), diag(a.uu()), max_iter, tol, &derr);
```

How to run the program

The compilation and run writes:

```
make helmholtz_band_iterative
mkgeo_grid -T 20 -a -2 -b 2 -c -2 -d 2 -f -2 -g 2 > cube-20.geo
./helmholtz_band_iterative cube-20.geo > sphere-band.field
```

The run generates also two meshes (see Fig. 7.5): the intersection mesh and the band around it. The solution is here defined on this band: this extension has no interpretation in terms of the initial problem and can be restricted to the intersection mesh for visualization purpose:

```
make proj_band
./proj_band < sphere-band.field | field -mayavi -stereo -
```

The '`proj_band.cc`' is presented below. The run generates also the Γ_h mesh (see Fig. 7.5), required for the visualization. The two-dimensional case is obtained simply by replacing the 3D bounding box by a 2D one:

```
mkgeo_grid -t 20 -a -2 -b 2 -c -2 -d 2 > square-20.geo
./helmholtz_band_iterative square-20.geo > circle-band.field
./proj_band < circle-band.field | field -mayavi -
./proj_band < circle-band.field | field -mayavi -elevation -
```

Example file 7.7: `proj_band.cc`

```
#include "rheolef.h"
using namespace std;
using namespace rheolef;
int main (int argc, char**argv) {
  environment rheolef (argc, argv);
  field phi_h;
  din >> catchmark("phi") >> phi_h;
  const space& Xh = phi_h.get_space();
  band gamma_h (phi_h);
  space Bh (gamma_h.band(), "P1");
  field uh(Bh);
  din >> catchmark("u") >> uh;
  space Wh (gamma_h.level_set(), "P1");
  gamma_h.level_set().save();
  dout << interpolate (Wh, uh);
}
```

7.4 A direct solver for the banded level set method

The iterative algorithm previously used for solving the linear system is not optimal: for 3D problems on a surface, the bidimensionnal connectivity of the sparse matrix suggests that a direct sparse factorisation would be much more efficent.

Recall that $\phi_h = 0$ on Γ_h. Thus, if $u_h \in B_h$ is solution of the problem, then $u_h + \alpha\phi_{h|\beta_h} \in B_h$ is also solution for any $\alpha \in \mathbb{R}$, where $\phi_{h|\beta_h} \in B_h$ denotes the restriction of the level set function $\phi_h \in X_h$ on the band β_h. Thus there is multiplicity of solutions and the matrix of the problem is singular. The direct resolution is still possible on a modified linear system with additional constraints in order to recover the unicity of the solution. We impose the constraint that the solution u_h should be othogonal to $\phi_{h|\beta_h} \in B_h$. In some special cases, the band is composed of several connected components (see Fig. 7.6): this appends when a vertex of the bounding box mesh belongs to Γ_h. In that case, the constaint sould be expressed on each connected component. Fig. 7.6 shows also the case when a full side of an element is included in Γ_h: such an element of the band is called *isolated*.

Example file 7.8: `helmholtz_band.cc`

```
#include "rheolef.h"
using namespace std;
using namespace rheolef;
#include "sphere.icc"
int main (int argc, char**argv) {
  environment rheolef(argc, argv);
  geo lambda (argv[1]);
  size_t d = lambda.dimension();
  space Xh (lambda, "P1");
  field phi_h = interpolate(Xh, phi);
  band gamma_h (phi_h);
  field phi_h_band = phi_h [gamma_h.band()];
  space Bh (gamma_h.band(), "P1");
  Bh.block ("isolated");
  Bh.unblock ("zero");
  trial u (Bh); test v (Bh);
  form  a  = integrate (gamma_h, u*v + dot(grad_s(u),grad_s(v)));
  field lh = integrate (gamma_h, f(d)*v);
  vector<vec<Float> > b (gamma_h.n_connected_component());
  vector<Float>       z (gamma_h.n_connected_component(), 0);
  for (size_t i = 0; i < b.size(); i++) {
    const domain& cci = gamma_h.band() ["cc"+itos(i)];
    field phi_h_cci (Bh, 0);
    phi_h_cci [cci] = phi_h_band [cci];
    b[i] = phi_h_cci.u();
  }
  csr<Float> A = { { a.uu(), trans(b)},
                   { b,          0    } };
  vec<Float> F =   { lh.u(),    z    };
  A.set_symmetry(true);
  solver sa = ldlt(A);
  vec<Float> U = sa.solve (F);
  field uh(Bh,0);
  uh.set_u() = U [range(0,uh.u().size())];
  dout << catchmark("phi") << phi_h
       << catchmark("u")   << uh;
}
```

Comments

The management of the special sides and vertices that are fully included in Γ_h is perfomed by:

```
  Bh.block ("isolated");
  Bh.unblock ("zero");
```

The addition of linear constraints is similar to the '`neumann-laplace.cc`' code, as presented in section 2.4:

```
  csr<Float> A = { { a.uu(), trans(b)},
                   { b,          0    } };
```

Here `b` is a `vector<vec<Float> >`, i.e. a vector of linear constraints, one per connected component of the band β_h.

How to run the program

The commands are similar to the previous iterative implementation, just replacing `helmholtz_band_iterative` by `helmholtz_band`.

This approach could be also adapted to the Laplace-Beltrami problem on the torus.

Example file 7.9: `laplace_band.cc`

```
#include "rheolef.h"
using namespace std;
using namespace rheolef;
#include "torus.icc"
int main (int argc, char**argv) {
  environment rheolef(argc, argv);
  geo lambda (argv[1]);
  size_t d = lambda.dimension();
  space Xh (lambda, "P1");
  field phi_h = interpolate(Xh, phi);
  band gamma_h (phi_h);
  field phi_h_band = phi_h [gamma_h.band()];
  space Bh (gamma_h.band(), "P1");
  Bh.block ("isolated");
  Bh.unblock ("zero");
  trial u (Bh); test v (Bh);
  form  m  = integrate (gamma_h, u*v);
  form  a  = integrate (gamma_h, dot(grad_s(u),grad_s(v)));
  field lh = integrate (gamma_h, f(d)*v);
  vector<vec<Float> > b (gamma_h.n_connected_component());
  vector<Float>       z (gamma_h.n_connected_component(), 0);
  for (size_t i = 0; i < b.size(); i++) {
    const domain& cci = gamma_h.band() ["cc"+itos(i)];
    field phi_h_cci (Bh, 0);
    phi_h_cci [cci] = phi_h_band [cci];
    b[i] = phi_h_cci.u();
  }
  field c = m*field(Bh,1);
  csr<Float> A = { { a.uu(),       trans(b), c.u()},
                   { b,            0,        0    },
                   { trans(c.u()), 0,        0    } };
  vec<Float> F =   { lh.u(),       z,        0};
  A.set_symmetry(true);
  solver sa = ldlt(A);
  vec<Float> U = sa.solve (F);
  field uh(Bh,0);
  uh.set_u() = U [range(0,uh.u().size())];
  dout << catchmark("phi") << phi_h
       << catchmark("u")   << uh;
}
```

Comments

The code is simlar to the previous one `helmholtz_band.cc`. Since the solution is defined up to a constant, an additional linear constraint has to be inserted:

$$\int_{\Gamma_h} u_h \, \mathrm{d}x = 0$$

This writes:

```
  field c = m*field(Bh,1);
  csr<Float> A = { { a.uu(),       trans(b), c.u()},
                   { b,            0,        0    },
                   { trans(c.u()), 0,        0    } };
```

How to run the program

```
make laplace_band
mkgeo_grid -T 20 -a -2 -b 2 -c -2 -d 2 -f -2 -g 2 > cube-20.geo
./laplace_band cube-20.geo > torus-band.field
```

```
geo cube-20.band.geo -stereo -cut
./proj_band < torus-band.field | field -mayavi -stereo -
```

The solution is represented on Fig. 7.5.bottom.

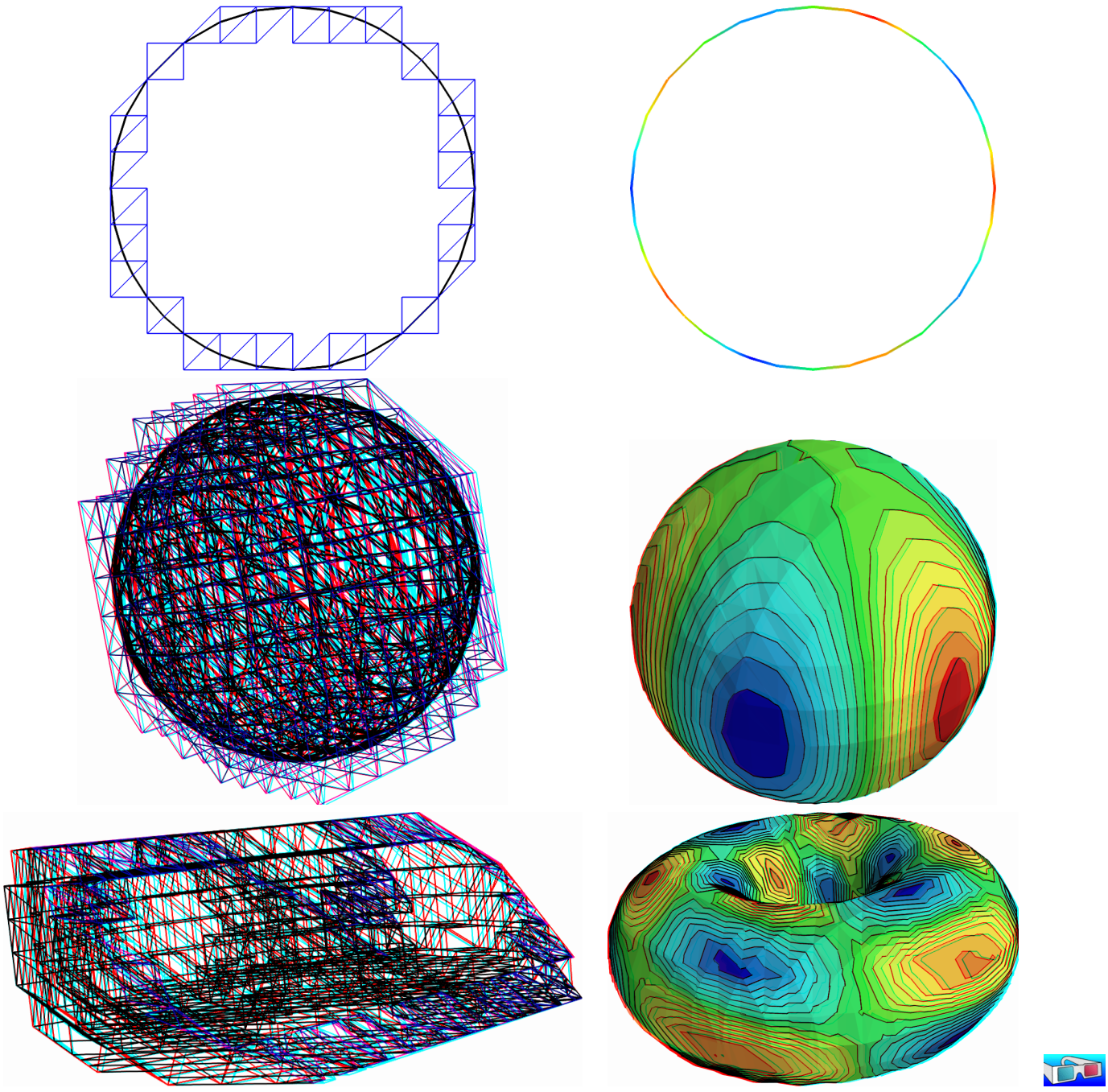


Figure 7.5: The banded level set method: (top) circle; (center) sphere; (bottom) torus.

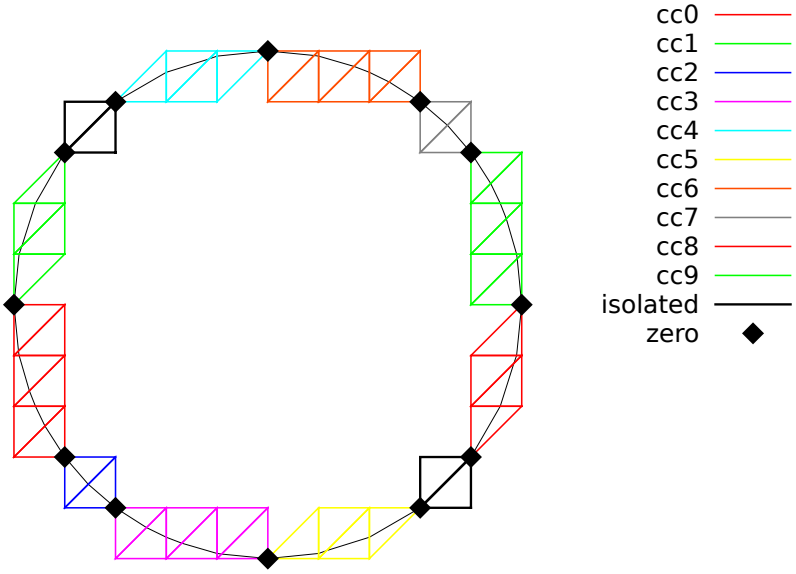


Figure 7.6: The banded level set method: the band is composed of several connected components.

Chapter 8

The highly nonlinear p-laplacian problem

8.1 Problem statement

Let us consider the classical p-Laplacian problem with homogeneous Dirichlet boundary conditions in a domain bounded $\Omega \subset \mathbb{R}^d$, $d = 1, 2, 3$:

(P): find u, defined in Ω such that:

$$\begin{aligned} -\text{div}\left(\eta\left(|\boldsymbol{\nabla} u|^2\right)\boldsymbol{\nabla} u\right) &= f \text{ in } \Omega \\ u &= 0 \text{ on } \partial\Omega \end{aligned}$$

where $\eta : z \in \mathbb{R}^+ \longmapsto z^{\frac{p-2}{2}} \in \mathbb{R}^+$. Several variants of the η can be considered: see [44] for practical and usefull examples. Here $p \in]1, +\infty[$ and f are known. For the computational examples, we choose $f = 1$. When $p = 2$, this problem reduces to the linear Poisson problem with homogeneous Dirichlet boundary conditions. Otherwise, for any $p > 1$, the nonlinear problem is equivalent to the following minimization problem:

(MP): find $u \in W_0^{1,p}(\Omega)$ such that:

$$u = \underset{v \in W_0^{1,p}(\Omega)}{\arg\min} \quad \frac{1}{2}\int_\Omega H\left(|\nabla v|^2\right)\,\mathrm{d}x - \int_\Omega f\, v\,\mathrm{d}x,$$

where H denotes the primitive of η:

$$H(z) = \int_0^z \eta(z)\,\mathrm{d}z = \frac{2z^p}{p}$$

Here $W_0^{1,p}(\Omega)$ denotes the usual Sobolev spaces of functions in $W^{1,p}(\Omega)$ We also assume that $f \in W^{-1,p}(\Omega)$, where $W_0^{-1,p}(\Omega)$ denotes the dual space of $W_0^{1,p}(\Omega)$ that vanishes on the boundary [8, p. 118]. The variational formulation of this problem expresses:

(VF): find $u \in W_0^{1,p}(\Omega)$ such that:

$$a(u; u, v) = l(v),\ \forall v \in W_0^{1,p}(\Omega)$$

where $a(.,.)$ and $l(.)$ are defined for any $u_0, u, v \in W^{1,p}(\Omega)$ by

$$a(u_0; u, v) = \int_\Omega \eta\left(|\nabla u_0|^2\right)\nabla u.\nabla v\,\mathrm{d}x, \quad \forall u, v \in W_0^{1,p}(\Omega) \tag{8.1}$$

$$l(v) = \int_\Omega f\, v\,\mathrm{d}x, \quad \forall u, v \in L^2(\Omega) \tag{8.2}$$

The quantity $a(u; u, u)^{1/p} = \|\boldsymbol{\nabla} u\|_{0,p,\Omega}$ induces a norm in $W_0^{1,p}$, equivalent to the standard norm. The form $a(.; ., .)$ is bilinear with respect to the two last variable and is related to the *energy* form.

8.2 The fixed-point algorithm

8.2.1 Principe of the algorithm

This nonlinear problem is then reduced to a sequence of linear subproblems by using the fixed-point algorithm. The sequence $\left(u^{(n)}\right)_{n\geqslant 0}$ is defined by recurrence as:

- $n=0$: let $u^{(0)} \in W_0^{1,p}(\Omega)$ be known.

- $n \geqslant 0$: suppose that $u^{(n)} \in W_0^{1,p}(\Omega)$ is known and find $u^* \in W_0^{1,p}(\Omega)$ such that:

$$a\left(u^{(n)}; u^*, v\right) = l(v),\ \forall v \in W_0^{1,p}(\Omega)$$

  and then set

$$u^{(n+1)} = \omega u^* + (1-\omega) * u^{(n)}$$

Here $\omega > 0$ is the relaxation parameter: when $\omega = 1$ we obtain the usual un-relaxed fixed point algorithm. For stiff nonlinear problems, we will consider the under-relaxed case $0 < \omega < 1$. Let $u^{(n+1)} = G\left(u^{(n)}\right)$ denotes the operator that solve the previous linear subproblem for a given $u^{(n)}$. Since the solution u satisfies $u = G(u)$, it is a fixed-point of G.

Let us introduce a mesh $\mathcal{T}_h$ of Ω and the finite dimensional space X_h of continuous piecewise polynomial functions and V_h, the subspace of X_h containing elements that vanishes on the boundary of Ω:

$$\begin{aligned} X_h &= \{v_h \in C_0^0\left(\overline{\Omega}\right);\ v_{h/K} \in P_k,\ \forall K \in \mathcal{T}_h\} \\ V_h &= \{v_h \in X_h;\ v_h = 0 \text{ on } \partial\Omega\} \end{aligned}$$

where $k = 1$ or 2. The approximate problem expresses: suppose that $u_h^{(n)} \in V_h$ is known and find $u_h^* \in V_h$ such that:

$$a\left(u_h^{(n)}; u_h^*, v_h\right) = l(v_h),\ \forall v_h \in V_h$$

By developing u_h^* on a basis of V_h, this problem reduces to a linear system.

Example file 8.1: `p_laplacian_fixed_point.cc`

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
#include "eta.icc"
#include "dirichlet.icc"
int main(int argc, char**argv) {
  environment rheolef (argc,argv);
  geo omega (argv[1]);
  Float  eps    = std::numeric_limits<Float>::epsilon();
  string approx = (argc > 2) ?      argv[2]  : "P1";
  Float  p      = (argc > 3) ? atof(argv[3]) : 1.5;
  Float  w      = (argc > 4) ? (is_float(argv[4]) ? atof(argv[4]) :2/p) :1;
  Float  tol    = (argc > 5) ? atof(argv[5]) : 1e5*eps;
  size_t max_it = (argc > 6) ? atoi(argv[6]) : 500;
  derr << "# P-Laplacian problem by fixed-point:" << endl
       << "# geo    = " << omega.name() << endl
       << "# approx = " << approx << endl
       << "# p      = " << p << endl
       << "# w      = " << w << endl
       << "# tol    = " << tol << endl;
  space Xh (omega, approx);
  Xh.block ("boundary");
  trial u (Xh); test v (Xh);
  form m = integrate (u*v);
  solver sm (m.uu());
  quadrature_option_type qopt;
  qopt.set_family (quadrature_option_type::gauss);
  qopt.set_order  (2*Xh.degree()-1);
  field uh (Xh);
  uh ["boundary"] = 0;
  field lh = integrate (v);
  dirichlet (lh, uh);
  derr << "# n r v" << endl;
  Float r = 1, r0 = 1;
  size_t n = 0;
  do {
    form a = integrate(compose(eta(p),norm2(grad(uh)))*dot(grad(u),grad(v)),
                       qopt);
    field mrh = a*uh - lh;
    field rh (Xh, 0);
    rh.set_u() = sm.solve (mrh.u());
    r = rh.max_abs();
    if (n == 0) { r0 = r; }
    Float v = (n == 0) ? 0 : log10(r0/r)/n;
    derr << n << " " << r << " " << v << endl;
    if (r <= tol || n++ >= max_it) break;
    solver sa (a.uu());
    vec<Float> u_star = sa.solve (lh.u() - a.ub()*uh.b());
    uh.set_u() = w*u_star + (1-w)*uh.u();
  } while (true);
  dout << catchmark("p") << p << endl
       << catchmark("u") << uh;
  return (r <= tol) ? 0 : 1;
}
```

8.2.2 Comments

The implementation with **Rheolef** involves a weighted forms: the tensor-valued weight $\eta\left(\left|\nabla u_h^{(n)}\right|^2\right)$ is inserted in the variationnal expression passed to the `integrate` function. The construction of the weighted form $a(.;.,.)$ writes:

```
  form a = integrate(compose(eta(p),norm2(grad(uh)))*dot(grad(u),grad(v)),
                     qopt);
```

Remarks the usage of the `compose`, `norm2` and `grad` libray functions. The weight $\eta\left(\left|\nabla u_h^{(n)}\right|^2\right)$ is represented by the `compose(eta(p),norm2(grad(uh)))` sub-expression. This weight is evaluated on the fly at the quadrature nodes during the assembly process implemented by the `integrate` function. Also, notice the distinction between u_h, that represents the value of the solution at step n, and the trial u and test v functions, that represents any elements of the function space X_h. These functions appear in the `dot(grad(u),grad(v))` sub-expression. As the integrals involved by this weighted form cannot be computed exactly for a general η function, a quadrature formula is used:

$$\int_K f(x)\,\mathrm{d}x = \sum_{q=0}^{n_K-1} f(x_{K,q})\,\omega_{K,q} + \mathcal{O}(h^{k'+1})$$

where $(x_{K,q},\omega_{K,q})_{0\leqslant q<n_K}$ are the quadrature nodes and weights on K and k' is the order of the quadrature: when f is a polynomial of degree less than k', the integral is exact. The bilinear form $a(.,.)$ introduced in (8.1) is then re-defined for all $u_0, u, v \in X_h$ by:

$$a(u_0;u,v) = \sum_{K\in\mathcal{T}_h}\sum_{q=0}^{n_K-1} \eta\left(|\nabla u_0(x_{K,q})|^2\right)\ \nabla u(x_{K,q}).\nabla v(x_{K,q})\ \omega_{K,q} \tag{8.3}$$

We choose the Gauss quadrature formula and the order k' is choosen as $k' = 2k-1$: the number n_K of nodes and weights in K is adjusted correspondingly. This choice writes:

```
quadrature_option_type qopt;
qopt.set_family (quadrature_option_type::gauss);
qopt.set_order  (2*Xh.degree()-1);
```

while the `qopt` variable is send as an optional argument to the weighted form $a(.,.)$ declaration. Remark that the integral would be exact for a constant weight. For a general weight, this choice also guarantee that the approximate solution u_h converges optimaly with mesh refinements to the exact solution u (see [38, p. 129]). Notice also that the Gauss quadrature formula is convenient here, as quadrature nodes are internal to the elements: evaluation of η does not occurs at the domain boundaries, where the weight function could be singular when $p < 2$ and where the gradient vanishes, e.g. at corners.

Example file 8.2: `eta.icc`

```
struct eta : std::unary_function<Float,Float> {
  Float operator() (const Float& z) const {
   check_macro(z != 0 || p > 2, "eta: division by zero (HINT: check mesh)");
   return pow(z, (p-2)/2);
  }
  Float derivative (const Float& z) const {
   check_macro(z != 0 || p > 4, "eta': division by zero (HINT: check mesh)");
   return 0.5*(p-2)*pow(z, (p-4)/2);
  }
  eta (const Float& q) : p(q) {}
  Float p;
};
```

The η function is implemented separately, in file named `eta.icc` in order to easily change its definition. The `derivative` member function is not yet used here: it is implemented for a forthcoming application (the Newton method). Notice the guards that check for division by zero and send a message related to the mesh: this will be commentated in the next paragraph. Finally, the fixed-point algorithm is initiated with $u^{(0)}$ as the solution of the linear problem associated to $p = 2$, i.e. the standard Poisson problem with Dirichlet boundary conditions.

Example file 8.3: `dirichlet.icc`

```
void dirichlet (const field& lh, field& uh) {
  const space& Xh = lh.get_space();
  trial u (Xh); test v (Xh);
  form a = integrate (dot(grad(u),grad(v)));
  solver sa (a.uu());
  uh.set_u() = sa.solve (lh.u() - a.ub()*uh.b());
}
```

8.2.3 Running the program

Compile the program, as usual:

```
make p_laplacian_fixed_point
```

and enter the commands:

```
mkgeo_ugrid -t 50 > square.geo
geo square.geo
```

The triangular mesh has a boundary domain named `boundary`.

```
./p_laplacian_fixed_point square.geo P1 1.5 > square.field
field square.field -elevation -stereo
```

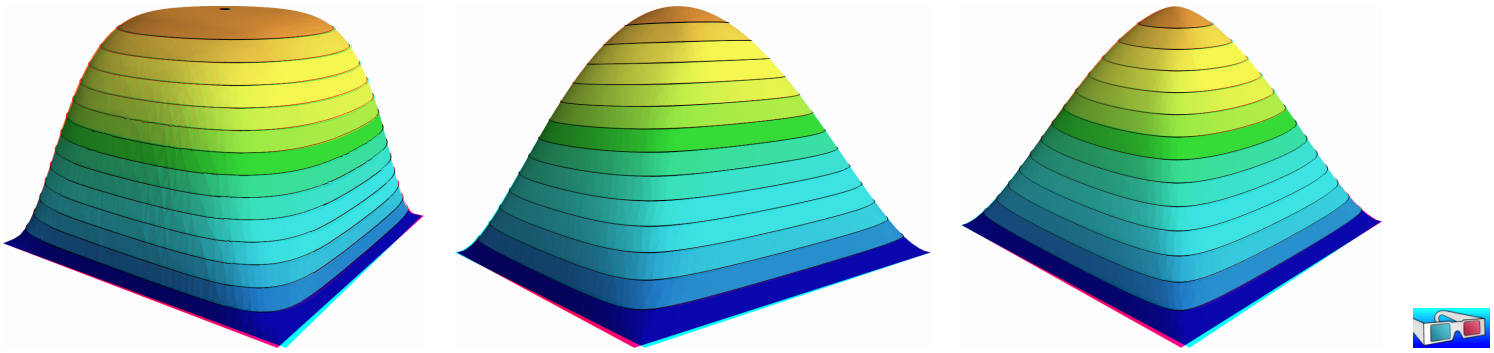


Figure 8.1: The p-Laplacian for $d = 2$: elevation view for $p = 1.25$ (left), $p = 2$ (center) and $p = 2.5$ (right).

Run the field visualization:

```
field square.field -elevation -stereo
field square.field -cut -origin 0.5 0.5 -normal 1 1
```

The first command shows an elevation view of the solution (see 8.1) while the second one shows a cut along the first bisector $x_0 = x_1$. Observe that the solution becomes flat at the center when p decreases. The $p = 2$ case, corresponding to the linear case, is showed for the purpose of comparison.

There is a technical issue concerning the mesh: the computation could failed on some mesh that presents at least one triangle with two edges on the boundary:

```
mkgeo_grid -t 50 > square-bedge.geo
geo square-bedge.geo
./p_laplacian_fixed_point square-bedge.geo P1 1.5 > square-bedge.field
```

The computation stops and claims a division by zero: the three nodes of such a triangle, the three nodes are on the boundary, where $u_h = 0$ is prescribed: thus $\boldsymbol{\nabla} u_h = 0$ uniformly inside this element. Notice that this failure occurs only for linear approximations: the computation works well on such meshes for P_k approximations with $k \geqslant 2$. While the `mkgeo_grid` generates uniform meshes that have such triangles, the `mkgeo_ugrid` calls the `gmsh` generator that automatically splits the triangles with two boundary edges. When using `bamg`, you should consider the `-splitpbedge`.

8.2.4 Convergence properties of the fixed-point algorithm

The fixed-point algorithm prints also r_n, the norm of the residual term, at each iteration n, and the convergence rate $v_n = \log_{10}(r_n/r_0)/n$. The residual term of the non-linear variational formulation is defined by:

$$r_h^{(n)} \in V_h \quad \text{and} \quad m\left(r_h^{(n)}, v_h\right) = a\left(u_h^{(n)}; u_h^{(n)}, v_h\right) - l(v_h), \quad \forall v_h \in V_h$$

where $m(.,.)$ denotes the L^2 scalar product. Clearly, $u_h^{(n)}$ is a solution if and only if $r_h^{(n)} = 0$.

For clarity, let us drop temporarily the n index of the current iteration. The field $r_h \in V_h$ can be extended as a field $r_h \in X_h$ with vanishing components on the boundary. The previous relation writes, after expansion of the bilinear forms and fields on the unknown and blocked parts (see page 17 for the notations):

```
m.uu*rh.u = a.uu*uh.u + a.ub*ub.b - lh.u
     rh.b = 0
```

This relation expresses that the residual term r_h is obtained by solving a linear system involving the mass matrix.

It remains to choose a good norm for estimating this residual term. For the corresponding continuous formulation, we have:

$$r = -\mathrm{div}\left(\eta\left(|\boldsymbol{\nabla} u|^2\right)\boldsymbol{\nabla} u\right) - f \ \in W^{-1,p}(\Omega)$$

Thus, for the continuous formulation, the residual term may be measured with the $W^{-1,p}(\Omega)$ norm. It is defined, for all $\varphi \in W^{-1,p}(\Omega)$, by duality:

$$\|\varphi\|_{-1,p,\Omega} = \sup_{\substack{\varphi \in W_0^{1,p}(\Omega) \\ v \neq 0}} \frac{\langle \varphi, v \rangle}{\|v\|_{1,p,\Omega}} = \sup_{\substack{v \in W_0^{1,p}(\Omega) \\ \|v\|_{1,p,\Omega}=1}} \langle \varphi, v \rangle$$

where $\langle .,. \rangle$ denotes the duality bracked between $W_0^{1,p}(\Omega)$ and $W^{-1,p}(\Omega)$.

By analogy, let us introduce the discrete $W^{-1,p}(\Omega)$ norm, denoted as $\|.\|_{-1,h}$, defined by duality for all $\varphi_h \in V_h$ by:

$$\|\varphi_h\|_{-1,h} = \sup_{\substack{v_h \in V_h \\ \|v_h\|_{1,p,\Omega}=1}} \langle \varphi_h, v_h \rangle$$

The dual of space of the finite element space V_h is identified to V_h and the duality bracked is the Euclidian scalar product of $\mathbb{R}^{\dim(V_h)}$. Then, $\|\varphi_h\|_{-1,h}$ is the largest absolue value of components of φ_h considered as a vector of $\mathbb{R}^{\dim(V_h)}$. With the notations of the **Rheolef** library, it simply writes:

```
Float r = rh.u().max_abs()
```

Fig 8.2.top-left shows that the residual term decreases exponentially versus n, since the slope of the plot in semi-log scale tends to be strait. Moreover, observe that the slope is independent of

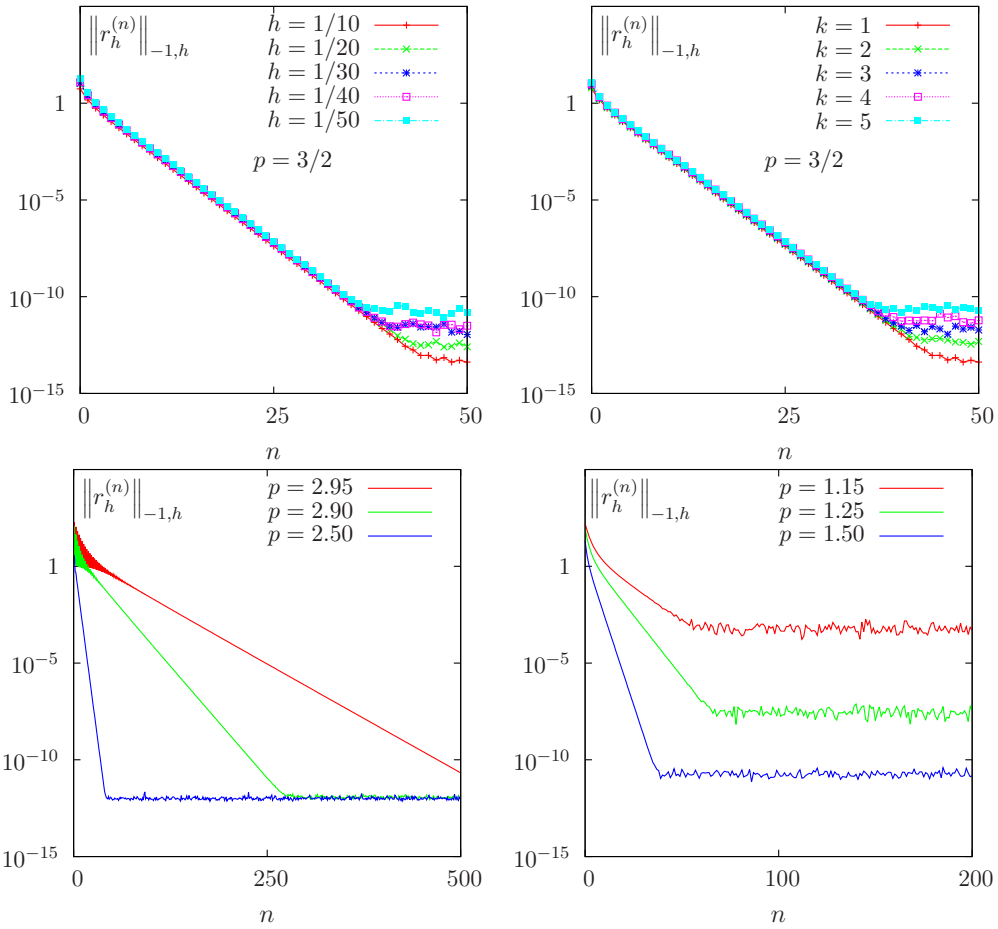


Figure 8.2: The fixed-point algorithm on the p-Laplacian for $d = 2$: when $p = 3/2$, independence of the convergence properties of the residue (top-left) with mesh refinement; (top-right) with polynomial order P_k; when $h = 1/50$ and $k = 1$, convergence (bottom-left) for $p > 2$ and (bottom-right) for $p < 2$.

the mesh size h. Also, by vertue of the previous carreful definition of the residual term and its corresponding norm, all the slopes falls into a master curve.

These invariance properties applies also to the polynomial approximation P_k : Fig 8.2.top-right shows that all the curves tends to collapse when k increases. Thus, the convergence properties of the algorithm are now investigated on a fixed mesh $h = 1/50$ and for a fixed polynomial approximation $k = 1$.

Fig 8.2.bottom-left and 8.2.bottom-right show the convergence vesus the power-law index p: observe that the convergence becomes easier when p approaches $p = 2$, where the problem is linear. In that case, the convergence occurs in one iteration. Nevertheless, it appears two limitations. From one hand, when $p \to 3$ the convergence starts to slow down and $p \geqslant 3$ cannot be solved by this algorithm (it will be solved later in this chapter). From other hand, when $p \to 1$, the convergence slows down too and numerical rounding effets limits the convergence: the machine precision canot be reached. Let us introduce the convergence rate $v_n = \log_{10}(r_n/r_0)/n$ it tends to a constant, denoted as $\bar{v}$ and: $r_n \approx r_0 \times 10^{-\bar{v}\,n}$. Observe on Fig 8.3.left that $\bar{v}$ tends to $+\infty$ when $p = 2$, since the system becomes linear and the algorithm converge in one iteration. Observe also that $\bar{v}$ tends to zero for $p = 1$ and $p = 3$ since the algorithm diverges. Fig 8.3.right shows the same plot in semi-log scale and shows that $\bar{v}$ behaves as: $\bar{v} \approx -\log_{10} |p - 2|$. This study shows

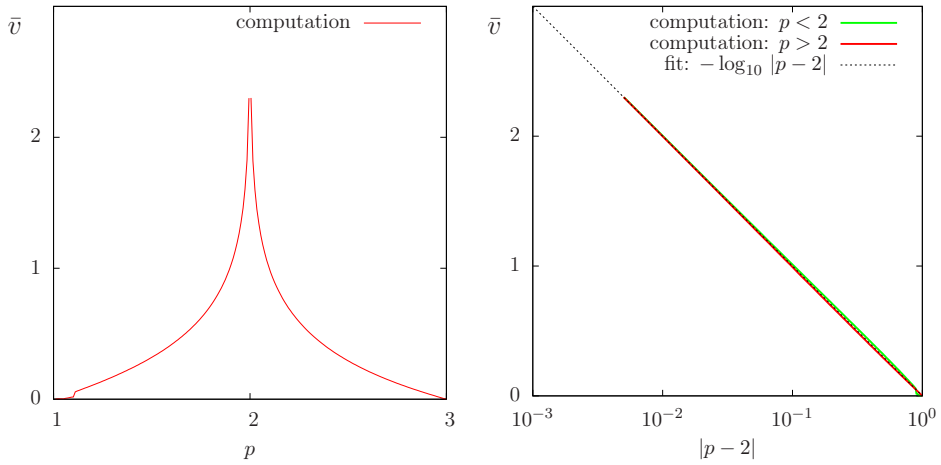


Figure 8.3: The fixed-point algorithm on the p-Laplacian for $d = 2$: (left) convergence rate versus p; (right) convergence rate versus p in semi-log scale.

that the residual term of the fixed point algorithm behaves as:

$$r_n \approx r_0 \, |p-2|^n$$

8.2.5 Improvement by relaxation

The relaxation parameter can improve the fixed-point algorithm: for instance, for $p = 3$ and $\omega = 0.5$ we get a convergent sequence:

```
./p_laplacian_fixed_point square.geo P1 3 0.5 > square.field
```

Observe on Fig. 8.4 the effect on the relaxation parameter ω upon the convergence rate $\bar{v}$: for $p < 2$ it can improve it and for $p > 2$, it can converge when $p > 3$. For each p, there is clearly an optimal relaxation parameter, denoted by ω_{opt}. A simple fit shows that (see Fig. 8.4.bottom-left):

$$\omega_{\text{opt}} = 2/p$$

Let us denote $\bar{v}_{\text{opt}}$ the corresponding rate of convergence. Fig. 8.4.top-right shows that the convergence is dramatically improved when $p > 2$ while the gain is less prononced when $p < 2$. Coveniently replacing the extra parameter ω on the command line by - leads to compute automatically $\omega = \omega_{\text{opt}}$: the fixed-point algorithm is always convergent with an optimal convergent rate, e.g.:

```
./p_laplacian_fixed_point square.geo P1 4.0 - > square.field
```

There is no way to improve more the fixed point algorithm: the next paragraph shows a different algorithm that dramatically accelerates the computation of the solution.

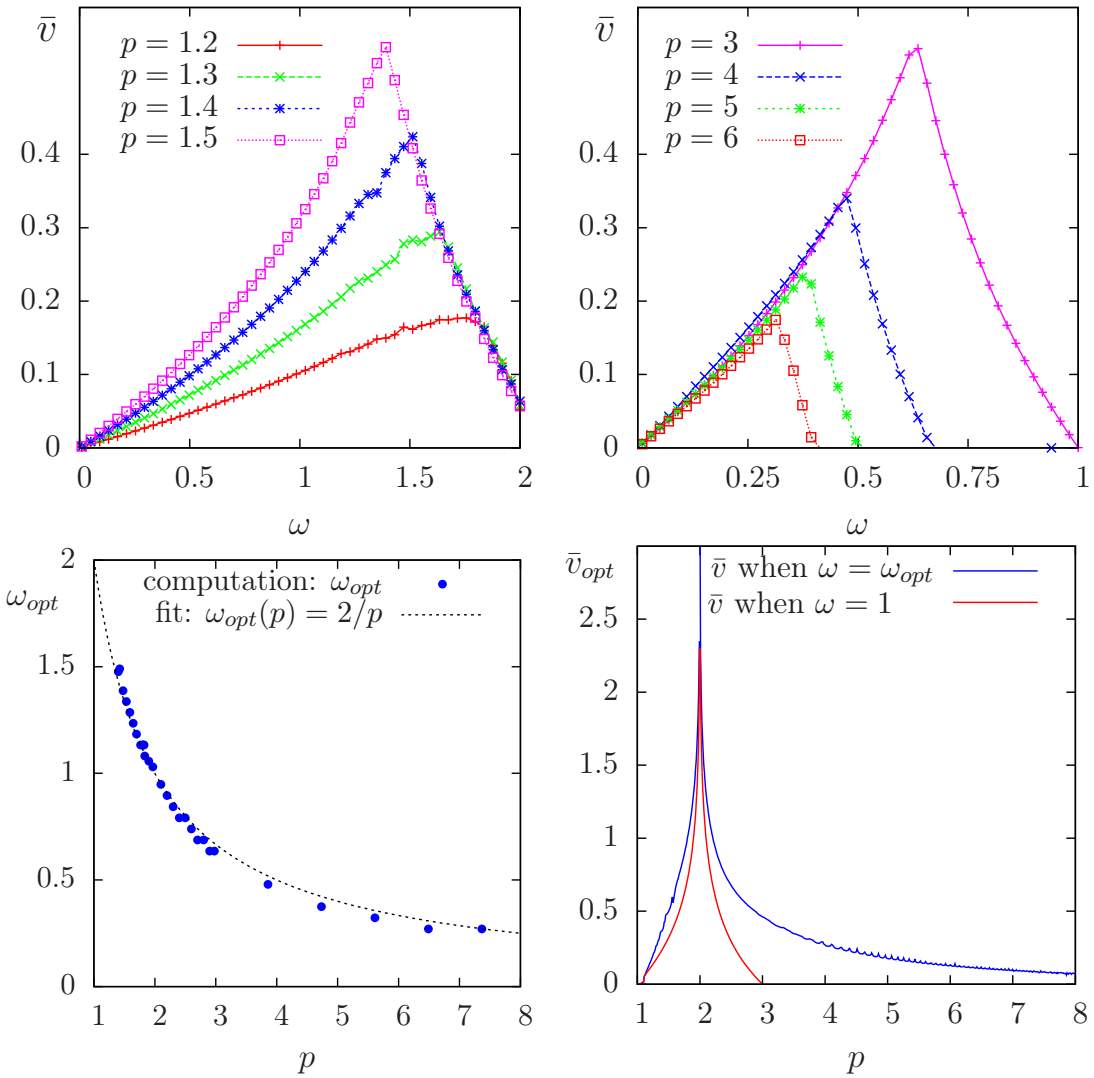


Figure 8.4: The fixed-point algorithm on the p-Laplacian for $d = 2$: effect of the relaxation parameter ω (top-left) when $p < 2$; (top-right) when $p > 2$; (bottom-left) optimal ω_{opt}; (bottom-right) optimal $\bar{v}_{\mathrm{opt}}$.

8.3 The Newton algorithm

8.3.1 Principe of the algorithm

An alternative to the fixed-point algorithm is to solve the nonlinear problem (P) by using the Newton algorithm. Let us consider the following operator:

$$\begin{array}{rcccl} F & : & W_0^{1,p}(\Omega) & \longrightarrow & W^{-1,p}(\Omega) \\ & & u & \longmapsto & F(u) = -\mathrm{div}\left(\eta\left(|\boldsymbol{\nabla} u|^2\right)\boldsymbol{\nabla} u\right) - f \end{array}$$

The F operator computes simply the residual term and the problem expresses now as: find $u \in W_0^{1,p}(\Omega)$ such that $F(u) = 0$.
The Newton algorithm reduces the nonlinear problem into a sequence of linear subproblems: the sequence $\left(u^{(n)}\right)_{n\geqslant 0}$ is classically defined by recurrence as:

- $n = 0$: let $u^{(0)} \in W_0^{1,p}(\Omega)$ be known.
- $n \geqslant 0$: suppose that $u^{(n)}$ is known, find $\delta u^{(n)}$, defined in Ω, such that:

$$F'\left(u^{(n)}\right)\ \delta u^{(n)} = -F\left(u^{(n)}\right)$$

  and then compute explicitly:

$$u^{(n+1)} := u^{(n)} + \delta u^{(n)}$$

The notation $F'(u)$ stands for the Fréchet derivative of F, as an operator from $W^{-1,p}(\Omega)$ into $W_0^{1,p}(\Omega)$. For any $r \in W^{-1,p}(\Omega)$, the linear tangent problem writes:
find $\delta u \in W_0^{1,p}(\Omega)$ such that:

$$F'(u)\,\delta u = -r$$

After the computation of the Fréchet derivative, we obtain the strong form of this problem:
(LT): find δu, defined in Ω, such that

$$\begin{array}{rcll} -\mathrm{div}\left(\eta\left(|\boldsymbol{\nabla} u|^2\right)\boldsymbol{\nabla}(\delta u) + 2\eta'\left(|\boldsymbol{\nabla} u|^2\right)\left\{\boldsymbol{\nabla} u.\boldsymbol{\nabla}(\delta u)\right\}\boldsymbol{\nabla} u\right) & = & -r & \text{in } \Omega \\ \delta u & = & 0 & \text{on } \partial\Omega \end{array}$$

where

$$\eta'(z) = \frac{1}{2}(p-2)z^{\frac{p-4}{2}},\ \forall z > 0$$

This is a Poisson-like problem with homogeneous Dirichlet boundary conditions and a non-constant tensorial coefficient. The variational form of the linear tangent problem writes:
(VLT): find $\delta u \in W_0^{1,p}(\Omega)$ such that

$$a_1(u;\delta u,\delta v) = l_1(v),\quad \forall \delta v \in W_0^{1,p}(\Omega)$$

where the $a_1(.;.,.)$ is defined for any $u, \delta u, \delta v \in W_0^{1,p}(\Omega)$ by:

$$\begin{array}{rcl} a_1(u;\delta u,\delta v) & = & \displaystyle\int_\Omega \left(\eta\left(|\boldsymbol{\nabla} u|^2\right)\boldsymbol{\nabla}(\delta u).\boldsymbol{\nabla}(\delta v) + 2\eta'\left(|\boldsymbol{\nabla} u|^2\right)\left\{\boldsymbol{\nabla} u.\boldsymbol{\nabla}(\delta u)\right\}\left\{\boldsymbol{\nabla} u.\boldsymbol{\nabla}(\delta v)\right\}\right)\mathrm{d}x \\ l_1(v) & = & \displaystyle -\int_\Omega r\,v\,\mathrm{d}x \end{array}$$

For any $\boldsymbol{\xi} \in \mathbb{R}^d$ let us denote by $\nu(\boldsymbol{\xi})$ the following $d \times d$ matrix:

$$\nu(\boldsymbol{\xi}) = \eta\left(|\boldsymbol{\xi}|^2\right)\ I + 2\eta'\left(|\boldsymbol{\xi}|^2\right)\ \boldsymbol{\xi}\otimes\boldsymbol{\xi}$$

where I stands for the d-order identity matrix. Then the a_1 expresses in a more compact form:

$$a_1(u;\delta u,\delta v) = \int_\Omega \left(\nu(\boldsymbol{\nabla} u)\boldsymbol{\nabla}(\delta u)\right).\boldsymbol{\nabla}(\delta v)\,\mathrm{d}x$$

Clearly a_1 is linear and symmetric with respect to the two last variables.

Example file 8.4: `p_laplacian_newton.cc`

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
#include "p_laplacian.h"
int main(int argc, char**argv) {
  environment rheolef (argc, argv);
  geo omega_h (argv[1]);
  Float  eps      = std::numeric_limits<Float>::epsilon();
  string approx   = (argc > 2) ?      argv[2]  : "P1";
  Float  p        = (argc > 3) ? atof(argv[3]) : 1.5;
  Float  tol      = (argc > 4) ? atof(argv[4]) : 1e5*eps;
  size_t max_iter = (argc > 5) ? atoi(argv[5]) : 500;
  derr << "# P-Laplacian problem by Newton:" << endl
       << "# geo = " << omega_h.name() << endl
       << "# approx = " << approx << endl
       << "# p = " << p << endl
       << "# tol = " << tol << endl
       << "# max_iter = " << max_iter << endl;
  p_laplacian F (p, omega_h, approx);
  field uh = F.initial ();
  int status = newton (F, uh, tol, max_iter, &derr);
  dout << setprecision(numeric_limits<Float>::digits10)
       << catchmark("p") << p << endl
       << catchmark("u") << uh;
  return status;
}
```

Example file 8.5: `p_laplacian.h`

```
class p_laplacian {
public:
  typedef field value_type;
  typedef Float float_type;
  p_laplacian (Float p, const geo& omega, string approx);
  field initial() const;
  field residue          (const field& uh) const;
  void update_derivative (const field& uh) const;
  field derivative_solve      (const field& mrh) const;
  field derivative_trans_mult (const field& mrh) const;
  Float space_norm       (const field& uh) const;
  Float dual_space_norm  (const field& mrh) const;
  Float duality_product  (const field& mrh, const field& msh) const;
  Float p;
  space Xh;
  field lh;
  form  m;
  solver sm;
  quadrature_option_type qopt;
  mutable form a1;
  mutable solver sa1;
};
#include "p_laplacian1.icc"
#include "p_laplacian2.icc"
```

8.3.2 Comments

The Newton algorithm is implemented in a generic way, for any F function, by the `newton` function from the **Rheolef** librarys. The reference manual for the `newton` generic function is available online:

```
man newton
```

The function F and its derivative F' are provided by a template class argument. Here, the `p_laplacian` class describes our F function, i.e. our problem to solve: its interface is defined in the file '`p_laplacian.h`' and its implementation in '`p_laplacian1.icc`' and '`p_laplacian2.icc`'. The introduction of the class `p_laplacian` will allow an easy exploration of some variants of the Newton algorithm for this problem, as we will see in the next section.

Example file 8.6: `p_laplacian1.icc`

```
#include "eta.icc"
#include "nu.icc"
#include "dirichlet.icc"
p_laplacian::p_laplacian (Float p1, const geo& omega, string approx)
 : p(p1), Xh(), lh(), m(), sm(), qopt(), a1(), sa1() {
  Xh = space (omega, approx);
  Xh.block ("boundary");
  qopt.set_family(quadrature_option_type::gauss);
  qopt.set_order(2*Xh.degree()-1);
  trial u (Xh); test v (Xh);
  lh = integrate (v);
  m = integrate (u*v);
  sm = solver (m.uu());
}
field p_laplacian::initial() const {
  field uh (Xh, 0);
  dirichlet (lh, uh);
  return uh;
}
field p_laplacian::residue (const field& uh) const {
  trial u (Xh); test v (Xh);
  form a = integrate (compose(eta(p), norm2(grad(uh)))*dot(grad(u),grad(v)),
                      qopt);
  field mrh = a*uh - lh;
  mrh.set_b() = 0;
  return mrh;
}
void p_laplacian::update_derivative (const field& uh) const {
  size_t d = Xh.get_geo().dimension();
  trial u (Xh); test v (Xh);
  a1 = integrate (dot(compose(nu<eta>(eta(p),d), grad(uh))*grad(u),grad(v)),
                  qopt);
  sa1 = ldlt (a1.uu());
}
field p_laplacian::derivative_solve (const field& rh) const {
  field delta_uh (Xh,0);
  delta_uh.set_u() = sa1.solve(rh.u());
  return delta_uh;
}
```

The residual term $F(u_h)$ is computed by the member function `residual` while the resolution of $F'(u_h)\delta u_h = Mr_h$ is performed by the function `derivative_solve`. The derivative $F'(u_h)$ is computed separately by the function `update_derivative`:

```
a1 = integrate(dot(compose(nu<eta>(eta(p),d),grad(uh))*grad(u),grad(v)),
               qopt);
```

Notice that the $a_1(u;.,.)$ bilinear form is a tensorial weighted form, where $\nu = \nu(\boldsymbol{\nabla} u)$ is the weight tensor. The tensorial weight ν is inserted as $(\nu\nabla u).\nabla v$ in the variationnal expression for the `integrate` function. As the tensor ν is symmetric, the bilinear form $a_1(.,.)$ is also symmetric. As the weight is non-polynomial for general η function and a quadrature formula is used:

$$a_1(u_0;u,v) = \sum_{K\in\mathcal{T}_h} \sum_{q=0}^{n_K-1} \left(\nu\left(\nabla u_0(x_{K,q})\right)\ \nabla u(x_{K,q}).\nabla v(x_{K,q})\right)\ \omega_{K,q} \tag{8.4}$$

By using exactly the same quadrature for computing both $a_1(.,.)$ and $a(.,.)$ in (8.4), then we have that F' is always the derivative of F at the discrete level: while, in general, the derivation and

the discretization of problems does not commute, it is the case when using the same quadrature formulae on both problems. This is an important aspect of the Newton method at discrete level, for conservating the optimal convergence rate of the residual terms versus n.

The linear system involving the derivative $F'(u_h)$ is solved by the `p_laplacian` member function `derivative_solve`. Finally, applying the generic Newton method requires a stopping criteria on the residual term: this is the aim of the member function `dual_space_norm`. The three last member functions are not used by the Newton algorithm, but by its extension, the damped Newton method, that will be presented later.

Example file 8.7: `p_laplacian2.icc`

```
Float p_laplacian::dual_space_norm (const field& mrh) const {
  field rh (Xh, 0);
  rh.set_u() = sm.solve (mrh.u());
  return rh.max_abs();
}
Float p_laplacian::space_norm (const field& uh) const {
  return sqrt (m(uh,uh));
}
Float p_laplacian::duality_product (const field& mrh, const field& msh) const {
  field rh (Xh, 0);
  rh.set_u() = sm.solve (mrh.u());
  return dual (rh, msh);
}
field p_laplacian::derivative_trans_mult (const field& mrh) const {
  field rh (Xh, 0);
  rh.set_u() = sm.solve(mrh.u());
  field mgh = a1*rh;
  mgh.set_b() = 0;
  return mgh;
}
```

The ν function is implemented for a generic η function, as a class-function that accept as template agument another class-function.

Example file 8.8: `nu.icc`

```
template<class Function>
struct nu : std::unary_function<point,tensor> {
  tensor operator() (const point& grad_u) const {
    Float x2 = norm2 (grad_u);
    Float a = f(x2);
    Float b = 2*f.derivative(x2);
    tensor value;
    for (size_t i = 0; i < d; i++) {
      value(i,i)   = a + b*grad_u[i]*grad_u[i];
      for (size_t j = 0; j < i; j++)
        value(j,i) = value(i,j) = b*grad_u[i]*grad_u[j];
    }
    return value;
  }
  nu (const Function& f1, Float d1) : f(f1), d(d1) {}
  Function f;
  size_t   d;
};
```

8.3.3 Running the program

Enter:

```
make p_laplacian_newton
mkgeo_ugrid -t 50 > square.geo
./p_laplacian_newton square.geo P1 3 > square.field
field square.field -elevation -stereo
```

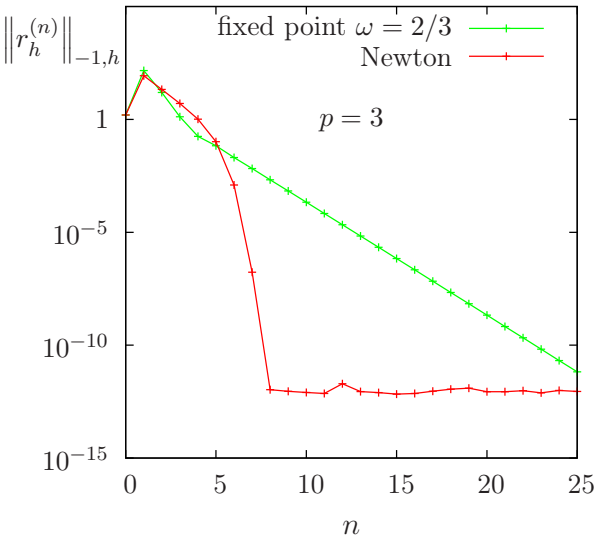


Figure 8.5: The Newton algorithm on the p-laplacian for $d = 2$: comparison with the fixed-point algorithm.

The program prints at each iteration n, the residual term r_n in discrete $L^2(\Omega)$ norm. Convergence occurs in less than ten iterations: it dramatically improves the previous algorithm (see Fig. 8.5). Observe that the slope is no more constant in semi-log scale: the convergence rate accelerates and the slope tends to be vertical, the so-called super-linear convergence. This is the major advantage of the Newton method. Figs. 8.6.top-left and. 8.6.top-bottom shows that the algorithm converge when $p \geqslant 3$ and that the convergence properties are independant of the mesh size h and the polynomial order k. There are still two limitations of the method. From one hand, the Newton algorithm is no more independant of h and k when $p \leqslant 3/2$ and to tends to diverges in that case when h tends to zero (see Fig. 8.6.bottom-left). From other hand, when p becomes large (see Fig. 8.6.bottom-right), an overshoot in the convergence tends to increases and distroys the convergence, due to rounding problems. In order to circumvent these limitations, another strategy is considered in the next section: the damped Newton algorithm.

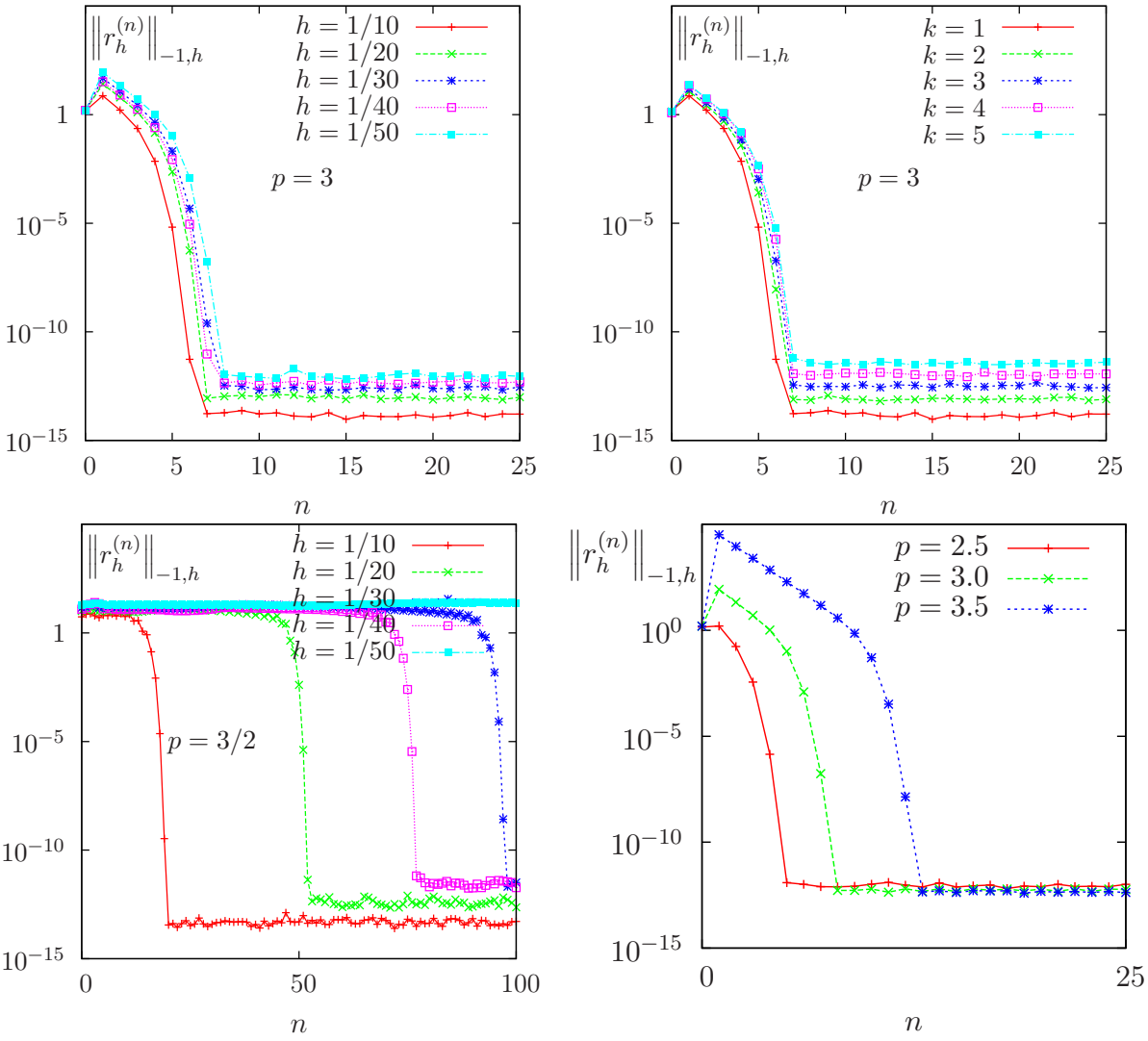


Figure 8.6: The Newton algorithm on the p-laplacian for $d = 2$: (top-left) comparison with the fixed-point algorithm; when $p = 3$, independence of the convergence properties of the residue (top-left) with mesh refinement; (top-right) with polynomial order P_k; (bottom-left) mesh-dependence convergence when $p < 2$; (bottom-right) overshoot when $p > 2$.

8.4 The damped Newton algorithm

8.4.1 Principe of the algorithm

The Newton algorithm diverges when the initial $u^{(0)}$ is too far from a solution, e.g. when p is not at the vicinity of 2. Our aim is to modify the Newton algorithm and to obtain a *globally convergent algorithm*, i.e to converge to a solution for any initial $u^{(0)}$. By this way, the algorithm should converge for any value of $p \in]1, +\infty[$. The basic idea is to decrease the step length while maintaining the direction of the original Newton algorithm:

$$u^{(n+1)} := u^{(n)} + \lambda_n \, \delta u^{(n)}$$

where $\lambda^{(n)} \in]0, 1]$ and $\delta u^{(n)}$ is the direction from the Newton algorithm, given by:

$$F'\left(u^{(n)}\right) \; \delta u^{(n)} = -F\left(u^{(n)}\right)$$

Let V a Banach space and let $T : V \to \mathbb{R}$ defined for any $v \in V$ by:

$$T(v) = \frac{1}{2} \| C^{-1} F(v) \|_V^2,$$

where C is some non-singular operator, easy to invert, used as a non-linear preconditioner. The simplest case, without preconditioner, is $C = I$. The T function furnishes a measure of the residual term in L^2 norm. The convergence is global when for any initial $u^{(0)}$, we have for any $n \geqslant 0$:

$$T\left(u^{(n+1)}\right) \leqslant T\left(u^{(n)}\right) + \alpha \left\langle T'\left(u^{(n)}\right), \; u^{(n+1)} - u^{(n)} \right\rangle_{V',V} \tag{8.5}$$

where $\langle ., . \rangle_{V',V}$ is the duality product between V and its dual V', and $\alpha \in]0, 1[$ is a small parameter. Notice that

$$T'(u) \quad = \quad \{C^{-1} F'(u)\}^* C^{-1} F(u)$$

where the superscript * denotes the adjoint operator, i.e. the transpose matrix the in finite dimensional case. In practice we consider $\alpha = 10^{-4}$ and we also use a minimal step length $\lambda_{\min} = 1/10$ in order to avoid too small steps. Let us consider a fixed step $n \geqslant 0$: for convenience the n superscript is dropped in $u^{(n)}$ and $\delta u^{(n)}$. Let $g : \mathbb{R} \to \mathbb{R}$ defined for any $\lambda \in \mathbb{R}$ by:

$$g(\lambda) = T\left(u + \lambda \delta u\right)$$

Then :

$$\begin{aligned} g'(\lambda) &= \langle T'(u + \lambda \delta u), \, \delta u \rangle_{V',V} \\ &= \langle C^{-1} F(u + \lambda \delta u), \; F'(u + \lambda \delta u) C^{-1} \delta u \rangle_{V,V'} \end{aligned}$$

where the superscript * denotes the adjoint operator, i.e. the transpose matrix the in finite dimensional case. The practical algorithm for obtaining λ was introduced first in [23] and is also presented in [37, p. 385]. The step length λ that satisfy (8.5) is computed by using a finite sequence λ_k, $k = 0, 1 \ldots$ with a second order recurrence:

- $k = 0$: initialization $\lambda_0 = 1$. If (8.5) is satisfied with $u + \lambda_0 \, d$ then let $\lambda := \lambda_0$ and the sequence stop here.
- $k = 1$: first order recursion. The quantities $g(0) = f(u)$ et $g'(0) = \langle f'(u), \, d \rangle$ are already computed at initialization. Also, we already have computed $g(1) = f(u + d)$ when verifying whether (8.5) was satisfied. Thus, we consider the following approximation of $g(\lambda)$ by a second order polynomial:

$$\tilde{g}_1(\lambda) = \{g(1) - g(0) - g'(0)\} \lambda^2 + g'(0) \lambda + g(0)$$

After a short computation, we find that the minimum of this polynomial is:

$$\tilde{\lambda}_1 = \frac{-g'(0)}{2\{g(1) - g(0) - g'(0)\}}$$

Since the initialization at $k = 0$ does not satisfy (8.5), it is possible to show that, when α is small enough, we have $\tilde{\lambda}_1 \leqslant 1/2$ and $\tilde{\lambda}_1 \approx 1/2$. Let $\lambda_1 := \max(\lambda_{\min}, \tilde{\lambda}_1)$. If (8.5) is satisfied with $u + \lambda_1 d$ then let $\lambda := \lambda_1$ and the sequence stop here.

- $k \geqslant 2$: second order recurrence. The quantities $g(0) = f(u)$ et $g'(0) = \langle f'(u), d \rangle$ are available, together with λ_{k-1}, $g(\lambda_{k-1})$, λ_{k-2} and $g(\lambda_{k-2})$. Then, $g(\lambda)$ is approximated by the following third order polynomial:

$$\tilde{g}_k(\lambda) = a\lambda^3 + b\lambda^2 + g'(0)\lambda + g(0)$$

where a et b are expressed by:

$$\begin{pmatrix} a \\ b \end{pmatrix} = \frac{1}{\lambda_{k-1} - \lambda_{k-2}} \begin{pmatrix} \frac{1}{\lambda_{k-1}^2} & -\frac{1}{\lambda_{k-2}^2} \\ -\frac{\lambda_{k-2}}{\lambda_{k-1}^2} & \frac{\lambda_{k-1}}{\lambda_{k-2}^2} \end{pmatrix} \begin{pmatrix} g(\lambda_{k-1}) - g'(0)\lambda_{k-1} - g(0) \\ g(\lambda_{k-2}) - g'(0)\lambda_{k-2} - g(0) \end{pmatrix}$$

The minimum of $\tilde{g}_k(\lambda)$ is

$$\tilde{\lambda}_k = \frac{-b + \sqrt{b^2 - 3ag'(0)}}{3a}$$

Let $\lambda_k = \min(1/2\,\lambda_k, \max(\tilde{\lambda}_k/10, \tilde{\lambda}_{k+1})$ in order for λ_k to be at the same order of magnitude as λ_{k-1}. If (8.5) is satisfied with $u + \lambda_k d$ then let $\lambda := \lambda_k$ and the sequence stop here.

The sequence $(\lambda_k)_{k \geqslant 0}$ is strictly decreasing: when the stopping criteria is not satisfied until λ_k reaches the machine precision $\varepsilon_{\text{mach}}$ then the algorithm stops with an error.

Example file 8.9: `p_laplacian_damped_newton.cc`

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
#include "p_laplacian.h"
int main(int argc, char**argv) {
  environment rheolef (argc,argv);
  geo omega_h (argv[1]);
  Float eps = numeric_limits<Float>::epsilon();
  string approx   = (argc > 2) ?      argv[2]  : "P1";
  Float  p        = (argc > 3) ? atof(argv[3]) : 1.5;
  Float  tol      = (argc > 4) ? atof(argv[4]) : eps;
  size_t max_iter = (argc > 5) ? atoi(argv[5]) : 500;
  derr << "# P-Laplacian problem by damped Newton:" << endl
       << "# geo = " << omega_h.name() << endl
       << "# approx = " << approx << endl
       << "# p = " << p << endl;
  p_laplacian F (p, omega_h, approx);
  field uh = F.initial ();
  int status = damped_newton (F, uh, tol, max_iter, &derr);
  dout << catchmark("p") << p << endl
       << catchmark("u") << uh;
  return status;
}
```

8.4.2 Comments

The `damped_newton` function implements the damped Newton algorithm for a generic $T(u)$ function, i.e. a generic nonlinear preconditioner. This algorithms use a backtrack strategy implemented

in the file '`newton-backtrack.h`' of the **Rheolef** library. The simplest choice of the identity preconditioner $C = I$ i.e. $T(u) = \|F(u)\|^2_{V'}/2$ is showed in file `damped-newton.h`. The gradient at $\lambda = 0$ is

$$T'(u) = F'(u)^* F(u)$$

and the slope at $\lambda = 0$ is:

$$\begin{aligned} g'(0) &= \langle T'(u),\, \delta u \rangle_{V',V} \\ &= \langle F(u),\, F'(u)\delta u \rangle_{V',V'} \\ &= -\|F(u)\|^2_{V'} \end{aligned}$$

The '`p_laplacian_damped_newton.cc`' is the application program to the p-Laplacian problem together with the $\|.\|_{L^2(\Omega)}$ discrete norm for the function T.

8.4.3 Running the program

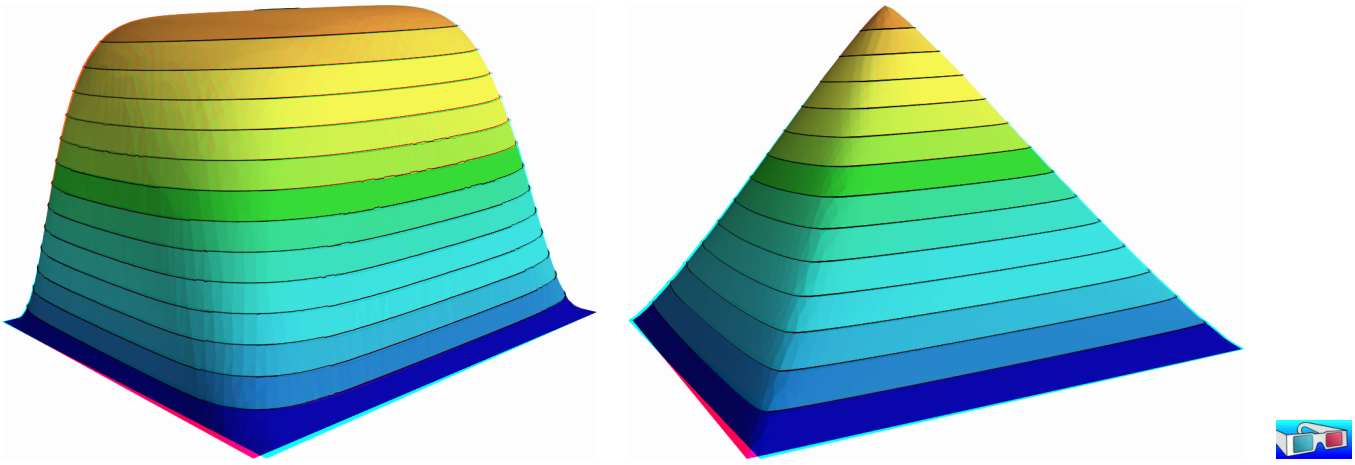


Figure 8.7: The p-Laplacian for $d = 2$: elevation view for $p = 1.15$ (left) and $p = 7$ (right).

As usual, enter:

```
make p_laplacian_damped_newton
mkgeo_ugrid -t 50 > square.geo
./p_laplacian_damped_newton square.geo P1 1.15 | field -stereo -elevation -
./p_laplacian_damped_newton square.geo P1 7    | field -stereo -elevation -
```

See Fig. 8.7 for the elevation view of the solution. The algorithm is now quite robust: the convergence occurs for quite large range of $p > 1$ values and extends the range previously presented on Fig. 8.1. The only limitation is now due to machine roundoff on some architectures.

Figs. 8.8.top shows that the convergence properties seems to slightly depend on the mesh refinement. Nevertheless, there are quite good and support both mesh refinement and high order polynomial degree. When p is far from $p = 2$, i.e. either close to one or large, Figs. 8.8.bottom shows that the convergence becomes slower and that the first linear regime, corresponding to the line search, becomes longer. This first regime finishes by a brutal super-linear regime, where the residual terms fall in few iterations to the machine precision.

8.5 Error analysis

While there is no simple explicit expression for the exact solution in the square $\Omega =]0,1[^2$, there is one when considering Ω as the unit circle:

$$u(x) = \frac{(p-1)\ 2^{-\frac{1}{p-1}}}{p} \left(1 - \left(x_0^2 + x_1^2\right)^{\frac{p}{p-1)}}\right)$$

Example file 8.10: `p_laplacian_circle.icc`

```
struct u_exact : field_functor<u_exact,Float> {
  Float operator() (const point& x) const {
    return (1 - pow(norm2(x), p/(2*p-2)))/((p/(p-1))*pow(2.,1/(p-1)));
  }
  u_exact (Float q) : p(q) {}
  protected: Float p;
};
struct grad_u : field_functor<grad_u,point> {
  point operator() (const point& x) const {
    return - (pow(norm2(x), p/(2*p-2) - 1)/pow(2.,1/(p-1)))*x;
  }
  grad_u (Float q) : p(q) {}
  protected: Float p;
};
```

Example file 8.11: `p_laplacian_error.cc`

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
#include "p_laplacian_circle.icc"
int main(int argc, char**argv) {
  environment rheolef (argc,argv);
  Float tol = (argc > 1) ? atof(argv[1]) : 1e-15;
  Float p;
  field uh;
  din >> catchmark("p") >> p
      >> catchmark("u") >> uh;
  const geo& omega = uh.get_geo();
  const space& Xh = uh.get_space();
  field pi_h_u = interpolate (Xh, u_exact(p));
  field eh = pi_h_u - uh;
  quadrature_option_type qopt;
  qopt.set_family(quadrature_option_type::gauss);
  qopt.set_order(2*Xh.degree());
  Float err_lp  = pow(integrate (omega,
    pow(fabs(uh - u_exact(p)), p), qopt), 1./p);
  Float err_w1p = pow(integrate (omega,
    pow(norm(grad(uh) - grad_u(p)), p), qopt), 1./p);
  Float err_linf = eh.max_abs();
  dout << "err_linf = " << err_linf << endl
       << "err_lp   = " << err_lp << endl
       << "err_w1p  = " << err_w1p << endl;
  return (err_linf < tol) ? 0 : 1;
}
```

Notice, in the file '`p_laplacian_error.cc`', the usage of the `integrate` function, together with a quadrature formula specificiation, for computing the errors in L^p norm and $W^{1,p}$ semi-norm. Notice also the flexibility of expressions, mixing together `fields` as `uh` and `field_functors`, as `u_exact`. The whole expression is evaluated by the `integrate` function at quadrature points inside each element of the mesh.

By this way, the error analysis investigation becomes easy:

```
make p_laplacian_error
mkgeo_ball -t 10 -order 2 > circle-10-P2.geo
./p_laplacian_damped_newton circle-10-P2.geo P2 1.5 | ./p_laplacian_error
```

We can vary both the mesh size and the polynomial order and the error plots are showed on Fig. 8.9 for both the L^2, L^∞ norms and the $W^{1,p}$ semi-norm. Observe the optimal error behavior: the slopes in the log-log scale are the same as those obtained by a direct Lagrange interpolation of the exact solution.

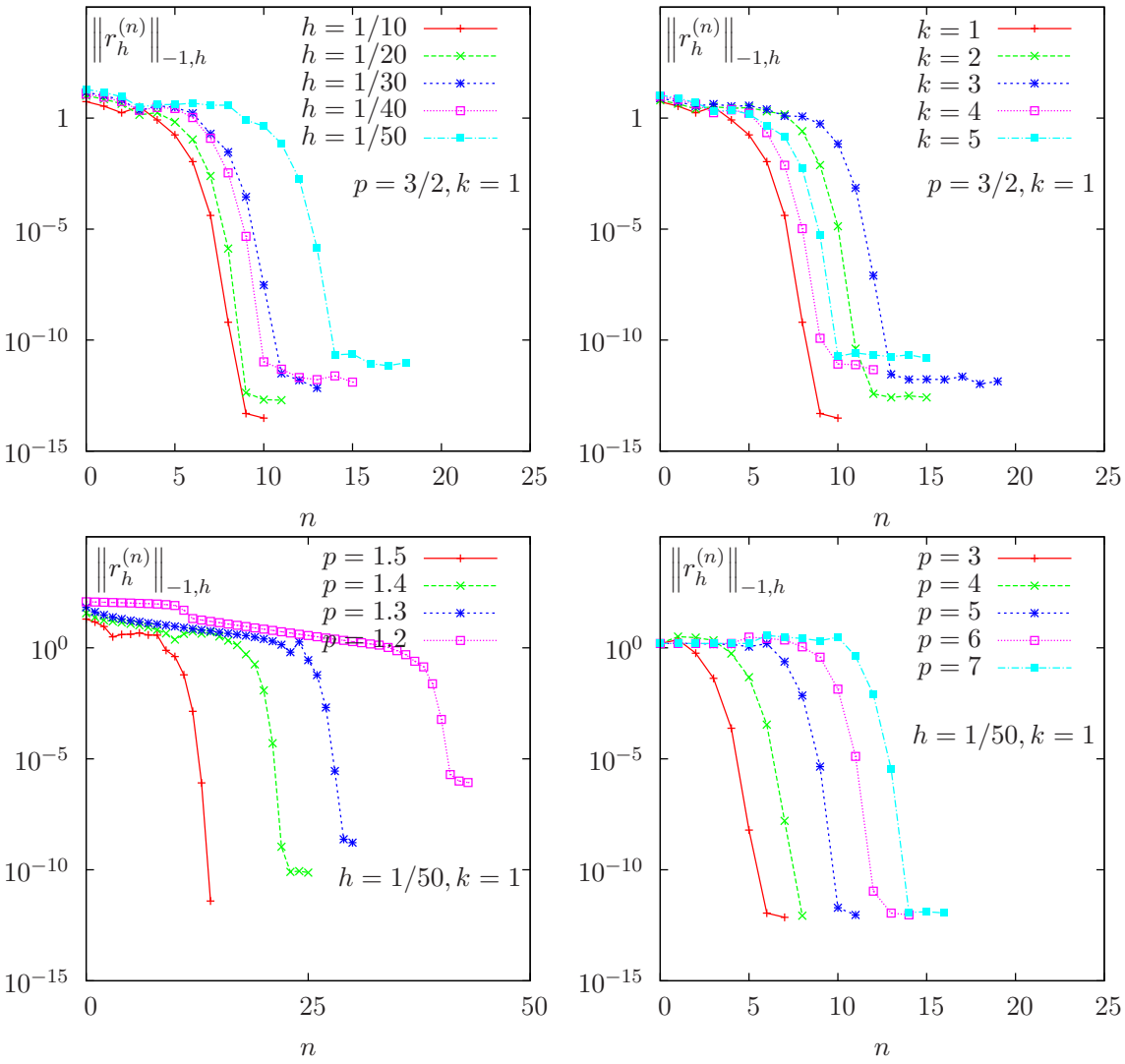


Figure 8.8: The damped Newton algorithm on the p-Laplacian for $d = 2$: when $p = 1.5$ and $h = 1/50$, convergence properties of the residue (top-left) with mesh refinement; (top-right) with polynomial order P_k; (bottom-left) convergence when $p < 2$; (bottom-right) when $p > 2$.

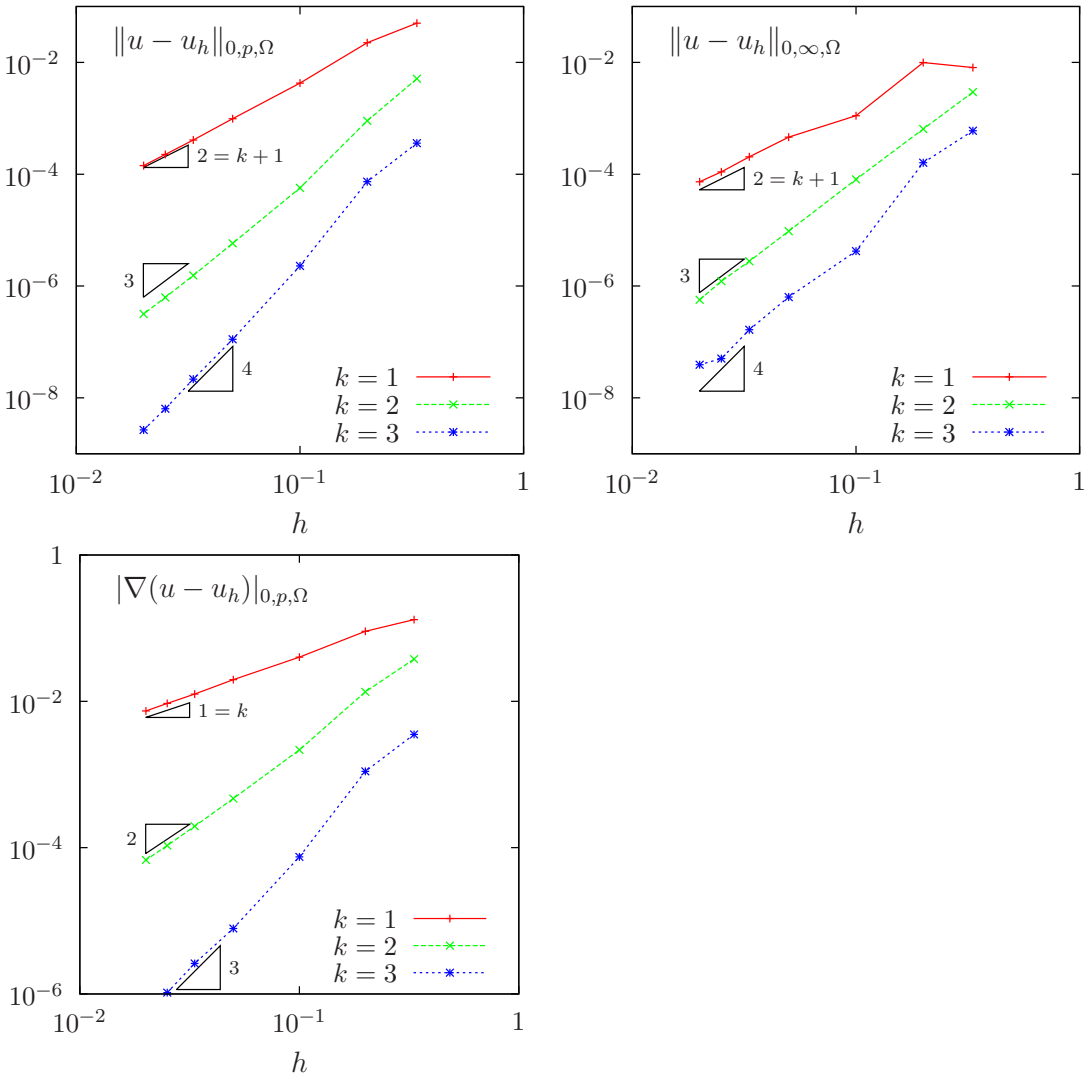


Figure 8.9: The p-Laplacian for $d=2$: error analysis.

Part IV

Technical appendices

Appendix A

How to write a variational formulation ?

The major keypoint for using **Rheolef** is to put the problem in variational form. Then this variational form can be efficiently translated into `C++` language. This appendix is dedicated to readers who are not fluent with variational formulations and some related functionnal analysis tools.

A.1 The Green formula

Let us come back to the model problem presented in section 1.1, page 15, equations (1.1)-(1.2) and details how this problem is transformed into (1.3).

Let $H_0^1(\Omega)$ the space of functions whose gradient square has a finite sum over Ω and that vanishes on $\partial\Omega$:

$$H_0^1(\Omega) = \{v \in L^2(\Omega);\ \nabla v \in L^2(\Omega)^d \text{ and } v = 0 \text{ on } \partial\Omega\}$$

We start by multiplying (1.1) by an arbitrarily test-function $v \in H_0^1(\Omega)$ and then integrate over Ω :

$$-\int_\Omega \Delta u\, v \,\mathrm{d}x = \int_\Omega f\, v \,\mathrm{d}x, \quad \forall v \in H_0^1(\Omega)$$

The next step is to invoque an integration by part, the so-called Green formula:

$$\int_\Omega \Delta u\, v \,\mathrm{d}x + \int_\Omega \nabla u.\nabla v \,\mathrm{d}x = \int_{\partial\Omega} \frac{\partial u}{\partial n}\, v \,\mathrm{d}s, \quad \forall u, v \in H^1(\Omega)$$

Since our test-function v vanishes on the boundary, the integral over $\partial\Omega$ is zero and the problem becomes:

$$\int_\Omega \nabla u.\nabla v \,\mathrm{d}x = \int_\Omega f\, v \,\mathrm{d}x, \quad \forall v \in H_0^1(\Omega)$$

This is exactly the variational formulation (1.3), page 15.

A.2 The vectorial Green formula

In this section, we come back to the linear elasticity problem presented in section 4.1, page 47, equations (4.1)-(4.2) and details how this problem is transformed into (4.3).

Let Γ_d (resp. Γ_n) denotes the parts of the boundary $\partial\Omega$ related to the homogeneous Dirichlet boundary condition $\mathbf{u} = 0$ (resp. the homogeneous Neumann boundary condition $\sigma(\mathbf{u})\,\mathbf{n} = 0$).

We suppose that $\partial\Omega = \overline{\Gamma}_d \cap \overline{\Gamma}_n$. Let us introduce the following functional space:

$$\mathbf{V} = \{\mathbf{v} \in H^1(\Omega)^d;\ \mathbf{v} = 0 \text{ on } \Gamma_d\}$$

Then, multiplying the first equation of (4.2) by an arbitrarily test-function $\mathbf{v} \in \mathbf{V}$ and then integrate over Ω :

$$-\int_\Omega \mathbf{div}(\sigma(\mathbf{u})).\mathbf{v}\,\mathrm{d}x = \int_\Omega \mathbf{f}.\mathbf{v}\,\mathrm{d}x, \quad \forall \mathbf{v} \in \mathbf{V}$$

The next step is to invoque an integration by part:

$$\int_\Omega \mathbf{div}\,\tau.\mathbf{v}\,\mathrm{d}x + \int_\Omega \tau : D(\mathbf{v})\,\mathrm{d}x = \int_{\partial\Omega} \tau : (\mathbf{v}\otimes\mathbf{n})\,\mathrm{d}s, \quad \forall \tau \in L^2(\Omega)^{d\times d},\ \forall \mathbf{v} \in \mathbf{V}$$

Recall that $\mathbf{div}\,\tau$ denotes $\left(\sum_{j=0}^{d-1} \partial_j \tau_{i,j}\right)_{0\leqslant i<d}$, i.e. the vector whose component are the divergence of each row of τ. Also, $\sigma : \tau$ denote the double contracted product $\sum_{i,j=0}^{d-1} \sigma_{i,j}\tau_{i,j}$ for any tensors σ and τ, and that $\mathbf{u}\otimes\mathbf{v}$ dotes the $\tau_{i,j} = u_i\, v_j$ tensor, vectors $\mathbf{u}$ and $\mathbf{v}$. Remark that $\tau : (\mathbf{u}\otimes\mathbf{v}) = (\tau\,\mathbf{v}).\mathbf{u} = \sum_{i,j=0}^{d-1} \tau_{i,j}\, u_i\, v_j$. Choosing $\tau = \sigma(\mathbf{u})$ in the previous equation leads to:

$$\int_\Omega \sigma(\mathbf{u}) : D(\mathbf{v})\,\mathrm{d}x = \int_{\partial\Omega} (\sigma(\mathbf{u})\,\mathbf{n}).\mathbf{v}\,\mathrm{d}s + \int_\Omega \mathbf{f}.\mathbf{v}\,\mathrm{d}x, \quad \forall \mathbf{v} \in \mathbf{V}$$

Since our test-function v vanishes on Γ_d and the solution satisfie the homogeneous Neumann boundary condition $\sigma(\mathbf{u})\,\mathbf{n} = 0$ on Γ_n, the integral over $\partial\Omega$ is zero and the problem becomes:

$$\int_\Omega \sigma(\mathbf{u}) : D(\mathbf{v})\,\mathrm{d}x = \int_\Omega \mathbf{f}.\mathbf{v}\,\mathrm{d}x, \quad \forall \mathbf{v} \in \mathbf{V}$$

From the definition of $\sigma(\mathbf{u})$ in (4.1) page 47 we have:

$$\begin{aligned}\sigma(\mathbf{u}) : D(\mathbf{v}) &= \lambda\,\mathrm{div}(\mathbf{u})\,(I : D(\mathbf{v})) + 2\mu D(\mathbf{u}) : D(\mathbf{v}) \\ &= \lambda\,\mathrm{div}(\mathbf{u})\,\mathrm{div}(\mathbf{v}) + 2\mu D(\mathbf{u}) : D(\mathbf{v})\end{aligned}$$

and the previous relation becomes:

$$\int_\Omega \lambda \mathrm{div}(\mathbf{u})\,\mathrm{div}(\mathbf{v})\,\mathrm{d}x + \int_\Omega 2\mu D(\mathbf{u}) : D(\mathbf{v})\,\mathrm{d}x = \int_\Omega \mathbf{f}.\mathbf{v}\,\mathrm{d}x, \quad \forall \mathbf{v} \in \mathbf{V}$$

This is exactly the variational formulation (4.3), page 48.

A.3 The Green formula on a surface

Let Γ a closed and orientable surface of $\mathbb{R}^d$, $d = 2, 3$ and $\mathbf{n}$ its unit normal. From [25], appendix C we have the following integration by part:

$$\int_\Gamma \mathrm{div}_s\mathbf{v}\,\xi\,\mathrm{d}s + \int_\Gamma \mathbf{v}.\nabla_s\xi\,\mathrm{d}s = \int_\Gamma \mathbf{v}.\mathbf{n}\,\xi\,\mathrm{div}\,\mathbf{n}\,\mathrm{d}s$$

for all $\xi \in H^1(\Gamma)$ and $\mathbf{v} \in H^1(\Gamma)^d$. Notice that $\mathrm{div}\,\mathbf{n}$ represent the surface curvature. Next, we choose $\mathbf{v} = \nabla_s\varphi$, for any $\varphi \in H^2(\Gamma)$. Remaking that $\mathbf{v}.\mathbf{n} = 0$ and that $\mathrm{div}_s\mathbf{v} = \Delta_s\varphi$. Then:

$$\int_\Gamma \Delta_s\,\xi\,\mathrm{d}s + \int_\Gamma \nabla_s\varphi.\nabla_s\xi\,\mathrm{d}s = 0$$

This formula is the starting point for all variational formulations of problems defined on a surface (see chapter 7).

Appendix B

How to prepare a mesh ?

Since there is many good mesh generators, **Rheolef**does not provide a built-in mesh generator. There are several ways to prepare a mesh for **Rheolef**.

We present here several procedures: by using the `bamg` bidimensional anisotropic mesh generator, written by Fréderic Hecht [21], and the `gmsh` mesh generator, suitable when $d = 1, 2$ and 3, and written by Christophe Geuzaine and Jean-François Remacle [17].

B.1 Bidimensional mesh with `bamg`

We first create a '`square.bamgcad`' file:

```
MeshVersionFormatted
  0
Dimension
  2
Vertices
  4
  0  0     1
  1  0     2
  1  1     3
  0  1     4
Edges
  4
  1  2     101
  2  3     102
  3  4     103
  4  1     104
hVertices
  0.1 0.1 0.1 0.1
```

This is an uniform mesh with element size $h = 0.1$. We refer to the `bamg` documentation [21] for the complete file format description. Next, enter the mesh generator commands:

```
bamg -g square.bamgcad -o square.bamg
```

Then, create the file '`square.dmn`' that associate names to the four boundary domains of the mesh. Here, there is four boundary domains:

```
EdgeDomainNames
```

```
4
bottom
right
top
left
```

and enter the translation command:

```
bamg2geo square.bamg square.dmn > square.geo
```

This command creates a '`square.geo`' file. Look at the mesh via the command:

```
geo square
```

This presents the mesh it in a graphical form, usually with `gnuplot`. You can switch to the `mayavi` renders:

```
geo square -mayavi
```

A finer mesh could be generated by:

```
bamg -coef 0.5 -g square.bamgcad -o square-0.5.bamg
```

B.2 Unidimensional mesh with `gmsh`

The simplest unidimensional mesh is a line:

```
h_local = 0.1;
Point(1) = {0, 0, 0, h_local};
Point(2) = {1, 0, 0, h_local};
Line(3) = {1,2};
Physical Point("left")  = {1};
Physical Point("right") = {2};
Physical Point("boundary") = {1,2};
Physical Line("interior") = {3};
```

The mesh generation command writes:

```
gmsh -1 line.mshcad -format msh -o line.msh
```

Then, the conversion to '`.geo`' format and the visualization:

```
msh2geo line.msh > line.geo
geo line
```

B.3 Bidimensional mesh with `gmsh`

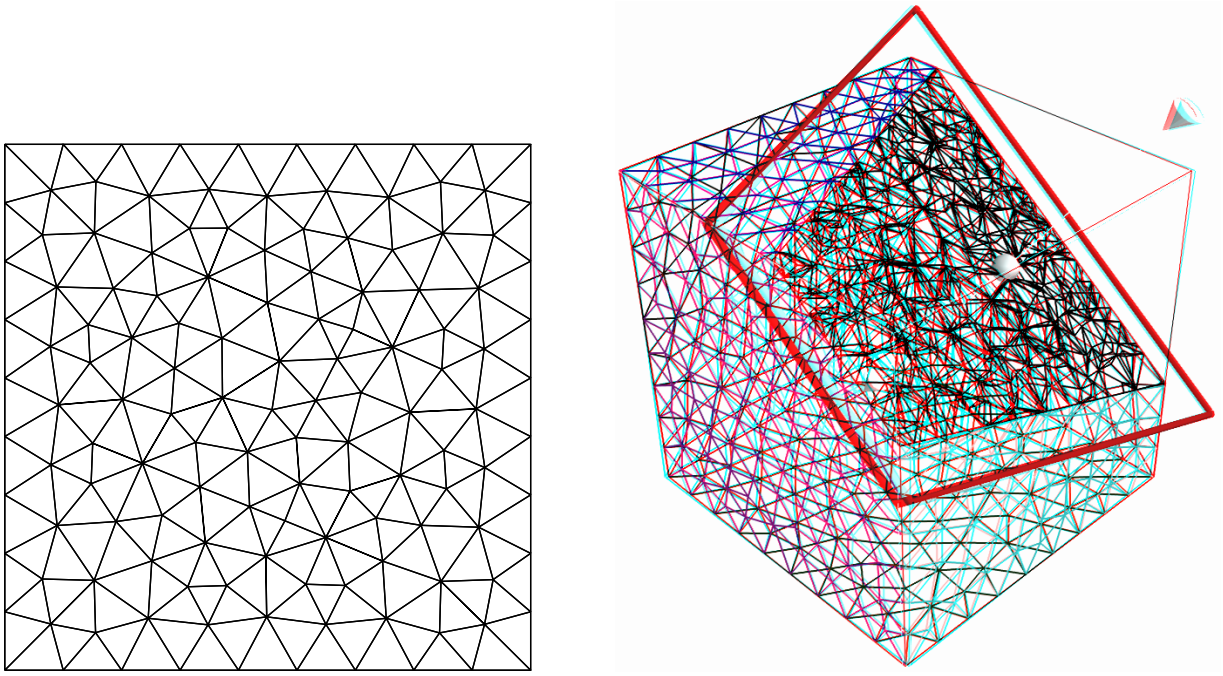


Figure B.1: Visualization of the `gmsh` meshes '`square.geo`' and '`cube.geo`'.

We first create a '`square.mshcad`' file:

```
n = 10.0;
hloc = 1.0/n;
Point(1) = {0, 0, 0, hloc};
Point(2) = {1, 0, 0, hloc};
Point(3) = {1, 1, 0, hloc};
Point(4) = {0, 1, 0, hloc};
Line(1) = {1,2};
Line(2) = {2,3};
Line(3) = {3,4};
Line(4) = {4,1};
Line Loop(5) = {1,2,3,4};
Plane Surface(6) = {5} ;
Physical Point("left_bottom")  = {1};
Physical Point("right_bottom") = {2};
Physical Point("right_top")    = {3};
Physical Point("left_top")     = {4};
Physical Line("boundary") = {1,2,3,4};
Physical Line("bottom") = {1};
Physical Line("right")  = {2};
Physical Line("top")    = {3};
Physical Line("left")   = {4};
Physical Surface("interior") = {6};
```

This is an uniform mesh with element size $h = 0.1$. We refer to the `gmsh` documentation [17] for the complete file format description. Next, enter the mesh generator commands:

```
gmsh -2 square.mshcad -format msh -o square.msh
```

Then, enter the translation command:

```
msh2geo square.msh > square.geo
```

This command creates a '`square.geo`' file. Look at the mesh via the command:

```
geo square
```

Remark that the domain names, defined in the .mshcad file, are included in the gmsh .msh input file and are propagated in the .geo by the format conversion.

B.4 Tridimensional mesh with `gmsh`

First, create a '`cube.mshcad`' file:

```
Mesh.Algorithm   = 7; // bamg
Mesh.Algorithm3D = 7; // mmg3d
a = 0; c = 0; f = 0;
b = 1; d = 1; g = 1;
n = 10;
hloc = 1.0/n;
Point(1) = {a, c, f, hloc};
Point(2) = {b, c, f, hloc};
Point(3) = {b, d, f, hloc};
Point(4) = {a, d, f, hloc};
Point(5) = {a, c, g, hloc};
Point(6) = {b, c, g, hloc};
Point(7) = {b, d, g, hloc};
Point(8) = {a, d, g, hloc};
Line(1) = {1,2};
Line(2) = {2,3};
Line(3) = {3,4};
Line(4) = {4,1};
Line(5) = {5,6};
Line(6) = {6,7};
Line(7) = {7,8};
Line(8) = {8,5};
Line(9)  = {1,5};
Line(10) = {2,6};
Line(11) = {3,7};
Line(12) = {4,8};
Line Loop(21) = {-1,-4,-3,-2};
Plane Surface(31) = {21} ;
Line Loop(22) = {5,6,7,8};
Plane Surface(32) = {22} ;
Line Loop(23) = {1,10,-5,-9};
Plane Surface(33) = {23} ;
Line Loop(24) = {12,-7,-11,3};
Plane Surface(34) = {24} ;
Line Loop(25) = {2,11,-6,-10};
Plane Surface(35) = {25} ;
Line Loop(26) = {9,-8,-12,4};
```

```
Plane Surface(36) = {26} ;
Surface Loop(41) = {31,32,33,34,35,36};
Volume(51) = {41};
Physical Surface("bottom") = {31};
Physical Surface("top") = {32};
Physical Surface("left") = {33};
Physical Surface("front") = {35};
Physical Surface("right") = {34};
Physical Surface("back") = {36};
Physical Volume("internal") = {51};
```

Next, enter the mesh generator commands:

```
gmsh -3 cube.mshcad -format msh -o cube.msh
```

Then, enter the translation command:

```
msh2geo cube.msh > cube.geo
```

This command creates a '`cube.geo`' file. Look at the mesh via the command:

```
geo cube
geo cube.geo -mayavi -cut
```

The second command allows to see inside the mesh.

Appendix C

Migrating to Rheolef version 6.0

Due to its new distributed memory and computation support, **Rheolef** version 6.0 presents some backward incompatibilities with previous versions: codes using previous versions of the library should be slightly modified. This appendix presents some indications for migrating existing code.

C.1 What is new in Rheolef 6.0 ?

The major main features are:

- support **distributed achitectures**: the code looks sequential, is easy to read and write but can be run massively parallel and distributed, based on the MPI library.
- **high order polynomial** approximation: P_k basis are introduced in this version, for $k \geqslant 0$. This feature will be improved in the future developments.
- **mesh adaptation** and the **charateristic method** are now available for **three-dimensional** problems.

In order to evoluate in these directions, internal data structures inside the library are completely rewritten in a different way, and thus this version is a completely new library.

Conversely, the library and unix command interfaces was as less as possible modified.

Nevertheless, the user will find some few backward incompatibilities: 5.93 based codes will not directly compile with the 6.0 library version. Let us review how to move a code from 5.93 to 6.0 version.

C.2 What should I have to change in my 5.x code ?

1. Namespace

The `namespace rheolef` was already introduced in last 5.93 version. Recall that a code usually starts with:

```
#include "rheolef.h"
using namespace rheolef;
```

2. Environment

The MPI library requires initialisation and the two command line arguments. This initialisation is performed via the `boost::mpi` class `environment`: The code entry point writes:

```
int main (int argc, char** argv) {
environment rheolef (argc,argv);
...
```

3. Fields and forms data accessors

The accesses to unknown and blocked data was of a `field uh` was direct, as `uh.u` and `uh.b`. This access is no more possible in a distributed environment, as non-local value requests may be optimized and thus, read and write access may be controled thought accessors. These accessors are named `uh.u()` and `uh.b()` for read access, and `uh.set_u()` and `uh.set_b()` for write access. Similarly, a `form a` has accessors as `a.uu()`.

A typical 5.93 code writes:

```
ssk<Float> sa = ldlt(a.uu);
uh.u = sa.solve (lh.u - a.ub*uh.b);
```

and the corresponding 6.0 code is:

```
solver sa (a.uu());
uh.set_u() = sa.solve (lh.u() - a.ub()*uh.b());
```

This major change in the library interface induces the most important work when porting to the 6.0 version.

Notice also that the old `ssk<Float>` class has been supersetted by the `solver` class, that manages both direct and iterative solvers in a more effective way. For three-dimensional problems, the iterative solver is the default while direct solvers are used otherwise. In the same spirit, a `solver_abtb` has been introduced, for Stokes-like mixed problem. These features facilitate the dimension-independent coding style provided by the **Rheolef** library.

4. Distributed input and output streams

Input and output *sequential* standard streams `cin`, `cout` and `cerr` may now replaced by *distributed* **Rheolef** streams `din`, `dout` and `derr` as:

```
din >> omega;
dout << uh;
```

These new streams are available togeher with the `idiststream` and `odiststream` classes of the **Rheolef** library.

5. File formats '.geo' and '.field' have changed

The '`.geo`' and '`.field`' file formats have changed. The '`.mfield`' is now obsolete: it has been merged into the '`.field`' format that supports now multi-component fields. Also, the corresponding `mfield` unix command is obsolete, as these features are integrated in the `field` unix command.

At this early stage of the 6.0 version, it is not yet possible to read the old '`.geo`' format, but this backward compatibility will be assured soon.

6. Space on a domain

A space defined on a domain `"boundary"` of a mesh `omega` was defined in the 5.93 version as:

```
space Wh (omega["boundary"], omega, "P1");
```

It writes now:

```
space Wh (omega["boundary"], "P1");
```

as the repetition of `omega` is no more required.

7. Nonlinear expressions involving fields

Non-linear operations, such as `sqrt(uh)` or `1/uh` was directly supported in **Rheolef**-5.x.

```
space Xh (omega, "P1");
field uh (Xh, 2.);
field vh = 1/uh;
```

Notice that non-linear operations as `1/uh` do not returns in general picewise polynomials while `uh*uh` is piecewise quadratic. In **Rheolef**-5.x, the returned value was implicitly the Lagrange interpolant of the nonlinear expression in space `Xh`. For more clarity, **Rheolef**-6.x requires an explicit call to the `interpolate` function and the code should write:

```
field vh = interpolate (Xh, 1/uh);
```

Notice that when the expression is linear, there is no need to call `interpolate`.

C.3 New features in Rheolef 6.4

The **Rheolef**-6.x code is in active developments. While backward compatibility s maintained since 6.0, some styles and idioms evoluates in order to increase the expressivity and the flexibility of the interface library. Here is the summary of these evolutions.

1. Nonlinear expressions

Nonlinear expressions has been extended since **Rheolef**-6.4 to expression mixing `field` and functions or `field_functor`. For instance, when `u_exact` is a `field_functor`, an L^2 error could be computed using a nonlinear expression submitted to the `integrate` function:

```
Float err_l2  = sqrt (integrate (omega, sqr (uh - u_exact()), qopt));
```

2. Right-hand-side specification

For specifiying a right-hand-side involving `f`, previous code style, from **Rheolef**-6.0 to 6.3 was using:

```
field lh = riesz (Xh, f());
```

Rheolef-6.4 introduces:

```
test v (Xh);
field lh = integrate (f()*v);
```

This feature opens new possibilities of right-hand-side specifications, e.g. expressions involving some derivatives of the test-function `v`. The `riesz` function is no more needed: it is maintained for backward compatibility purpose.

3. Form specification

For specifiying a bilinear form, previous code style, from **Rheolef**-6.0 to 6.3 was using a specification based on a name:

```
form a (Xh, Xh, "grad_grad");
```

Rheolef-6.4 introduces:

```
trial u (Xh); test v (Xh);
form a = integrate (dot(grad(u),grad(v)));
```

This feature opens new possibilities for form specifications and more flexibility. The `form` specification based on a name is no more needed: it is maintained for backward compatibility purpose.

Appendix D

GNU Free Documentation License

Version 1.1, March 2000

Copyright © 2000 Free Software Foundation, Inc.
59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

*

Preamble

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

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

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

Applicability and Definitions

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

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

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary

Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

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

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

Verbatim Copying

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

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

Copying in Quantity

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

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

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

Modifications

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
- State on the Title page the name of the publisher of the Modified Version, as the publisher.
- Preserve all the copyright notices of the Document.
- Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- Include an unaltered copy of this License.
- Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

- In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.
- Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties – for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

Combining Documents

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

Collections of Documents

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single

copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

Aggregation With Independent Works

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

Translation

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

Termination

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

Future Revisions of This License

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

*

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being LIST"; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Bibliography

[1] E. M. Abdalass. *Résolution performante du problème de Stokes par mini-éléments, maillages auto-adaptatifs et méthodes multigrilles – applications.* PhD thesis, Thèse de l'école centrale de Lyon, 1987. 68

[2] L. Abouorm. *Méthodes mathématiques pour les écoulements sur des surfaces.* PhD thesis, M2R Université J. Fourier, Grenoble, 2010. 109, 110

[3] D. N. Arnold, F. Brezzi, and M. Fortin. A stable finite element for the Stokes equations. *Calcolo*, 21:337–344, 1984. 67

[4] F. Auteri, N. Parolini, and L. Quartapelle. Numerical investigation on the stability of singular driven cavity flow. *J. Comput. Phys.*, 183(1):1–25, 2002. 93

[5] G. K. Batchelor. *An introduction to fluid dynamics.* Cambridge university press, UK, sixth edition, 1967. 73

[6] H. Borouchaki, P. L. George, F. Hecht, P. Laug, B. Mohammadi, and E. Saltel. Mailleur bidimensionnel de Delaunay gouverné par une carte de métriques. Partie II: applications. Technical Report RR-2760, INRIA, 1995. 54

[7] K. Boukir, Y. Maday, B. Metivet, and E. Razafindrakoto. A high-order characteristic/finite element method for the incompressible Navier-Stokes equations. *Int. J. Numer. Meth. Fluids*, 25:1421–1454, 1997. 86

[8] H. Brezis. *Analyse fonctionnelle. Théorie et application.* Masson, Paris, 1983. 117

[9] F. Brezzi and J. Pitkäranta. On the stabilization of finite element approximation of the Stokes equations. In *Efficient solutions of elliptic systems, Kiel, Notes on numerical fluid mechanics*, volume 10, pages 11–19, 1984. 69

[10] M. J. Castro-Diaz, F. Hecht, B. Mohammadi, and O. Pironneau. Anisotropic unstructured mesh adaption for flow simulations. *Int. J. Numer. Meth. Fluids*, 25(4):475–491, 1997. 54

[11] K. Deckelnick, G. Dziuk, C.M. Elliott, and C.-J. Heine. An h-narrow band finite element method for elliptic equations on implicit surfaces. *IMA Journal of Numerical Analysis*, to appear:0, 2009. 98

[12] M. Dicko. *Méthodes mathématiques pour les écoulements sur des surfaces.* PhD thesis, M2P Université J. Fourier, Grenoble, 2011. 109

[13] J. Donea and A. Huerta. *Finite element methods for flow problems.* Wiley, New-York, 2003. 92

[14] E. Erturk, T. C. Corke, and C. Gökçol. Numerical solutions of 2-D steady incompressible driven cavity flow at high Reynolds numbers. *Int. J. Numer. Meth. Fluids*, 48:747–774, 2005. 93

[15] G. Fourestey and S. Piperno. A second-order time-accurate ALE Lagrange-Galerkin method applied to wind engineering and control of bridge profiles. *Comput. Methods Appl. Mech. Engrg.*, 193:4117–4137, 2004. 86

[16] T. Gelhard, G. Lube, M. A. Olshanskii, and J. H. Starcke. Stabilized finite element schemes with LBB-stable elements for incompressible flows. *J. Comput. Appl. Math.*, 177:243–267, 2005. 93

[17] C. Geuzaine and J.-F. Remacle. Gmsh: a three-dimensional finite element mesh generator with built-in pre- and post-processing facilities. *Int. J. Numer. Meths Engrg.*, 79(11):1309–1331, 2009. 143, 145

[18] U. Ghia, K. N. Ghia, and C. T. Shin. High *Re* solutions for incompressible flow using the Navier-Stokes equations and a multigrid method. *J. Comput. Phys.*, 48:387–411, 1982. 92

[19] V. Girault and P. A. Raviart. *Finite element methods for the Navier-Stokes equations. Theory and algorithms.* Springer, 1986. 35, 60, 62

[20] M. M. Gupta and J. C. Kalita. A new paradigm for solving Navier-Stokes equations: streamfunction-velocity formulation. *J. Comput. Phys.*, 207:52–68, 2005. 92

[21] F. Hecht. *BAMG: bidimensional anisotropic mesh generator*, 2006. `http://www.ann.jussieu.fr/~hecht/ftp/bamg`. 54, 143

[22] P. Hood and C. Taylor. A numerical solution of the Navier-Stokes equations using the finite element technique. *Comp. and Fluids*, 1:73–100, 1973. 58, 66, 87

[23] Jr. J. E. Dennis and R. B. Schnablel. *Numerical methods for unconstraint optimization and nonlinear equations.* Prentice Hall, Englewood Cliff, N. J., 1983. 132

[24] A. Klawonn. An optimal preconditioner for a class of saddle point problems with a penalty term. *SIAM J. Sci. Comput*, 19(2):540–552, 1998. 59, 66

[25] A. Laadhari, C. Misbah, and P. Saramito. On the equilibrium equation for a generalized biological membrane energy by using a shape optimization approach. *Phys. D*, 239:1568–1572, 2010. 142

[26] R. J. Labeur and G. N. Wells. A Galerkin interface stabilisation method for the advection-diffusion and incompressible Navier-Stokes equations. *Comput. Meth. Appl. Mech. Engrg.*, 196(49–52):4985–5000, 2007. 92

[27] S. Melchior, V. Legat, P. Van Dooren, and A. J. Wathen. Analysis of preconditioned iterative solvers for incompressible flow problems. *Int. J. Numer. Meth. Fluids*, 2011. 93

[28] P. D. Minev and C. R. Ethier. A characteristic/finite element algorithm for the 3-D Navier-Stokes equations using unstructured grids. *Comput. Meth. in Appl. Mech. and Engrg.*, 178(1-2):39–50, 1998. 93

[29] D. R. Musser and A. Saini. *C++ STL tutorial and reference guide.* Addison Wesley, Reading, 1996. 24

[30] D. R. Musser and A. Saini. *STL tutorial and reference guide.* Addison-Wesley, 1996. 82

[31] M. A. Olshanskii and A. Reusken. A finite element method for surface PDEs: matrix properties. *Numer. Math.*, 114:491–520, 2010. 110

[32] M. A. Olshanskii, A. Reusken, and J. Grande. A finite element method for elliptic equations on surfaces. *SIAM J. Num. Anal.*, 47(5):3339–3358, 2009. 104, 109, 110

[33] M. L. Ould Salihi. *Couplage de méthodes numériques en simulation directe d'écoulements incompressibles*. PhD thesis, Université J. Fourier, Grenoble, 1998. 93

[34] C. C. Paige and M. A. Saunders. Solution of sparse indefinite systems of linear equations. *SIAM J. Numer. Anal.*, 12(4):617–629, 1975. 35

[35] O. Pironneau. *Méthode des éléments finis pour les fluides*. Masson, Paris, 1988. 81

[36] O. Pironneau and M. Tabata. Stability and convergence of a galerkin-characteristics finite element scheme of lumped mass type. *Int. J. Numer. Meth. Fluids*, 64:1240–1253, 2010. 82

[37] W. H. Press, S. A. Teulkolsky, W. T. Vetterling, and B. P. Flannery. *Numerical recepies in C. The art of scientific computing*. Cambridge University Press, UK, second edition, 1997. Version 2.08. 132

[38] P. A. Raviart and J. M. Thomas. *Introduction à l'analyse numerique des équations aux dérivées partielles*. Masson, Paris, 1983. 120

[39] N. Roquet, R. Michel, and P. Saramito. Errors estimate for a viscoplastic fluid by using Pk finite elements and adaptive meshes. *C. R. Acad. Sci. Paris, ser. I*, 331(7):563–568, 2000. 54

[40] H. Rui and M. Tabata. A second order characteristic finite element scheme for convection diffusion problems. *Numer. Math. (*to appear*)*, 2001. 81

[41] P. Saramito. *Simulation numérique d'écoulements de fluides viscoélastiques par éléments finis incompressibles et une méthode de directions alternées; applications*. PhD thesis, Institut National Polytechnique de Grenoble, 1990. 73

[42] P. Saramito. **Rheolef** home page. http://www-lmc.imag.fr/lmc-edp/Pierre.Saramito/rheolef, 2012. 25

[43] P. Saramito. *Language C++ et calcul scientifique*. CNRS-CCSD, 2012. http://cel.archives-ouvertes.fr/cel-00573975. 3

[44] P. Saramito. *Méthodes numériques en fluides complexes : théorie et algorithmes*. CNRS-CCSD, 2012. http://cel.archives-ouvertes.fr/cel-00673816. 117

[45] P. Saramito and N. Roquet. An adaptive finite element method for viscoplastic fluid flows in pipes. *Comput. Meth. Applied Mech. Engng*, 190(40-41):5391–5412, 2001. 22

[46] Pierre Saramito. Are curved and high order gmsh meshes really high order ?, 2012. http://www.geuz.org/pipermail/gmsh/2012/006967.html. 105

[47] J. Shen. Hopf bifurcation of the unsteady regularized driven cavity flow. *J. Comp. Phys.*, 95:228–245, 1991. http://www.math.purdue.edu/~shen/pub/Cavity.pdf. 93

[48] B. Stroustrup. *The C++ programming language*. Addison Wesley, Reading, special edition, 1997. 3

[49] B. Stroustrup. C++ programming styles and libraries. *InformIt.com*, 0:0, 2002. 2

[50] M.-G. Vallet. Génération de maillages anisotropes adaptés, application à la capture de couches limites. Technical Report RR-1360, INRIA, 1990. 54

[51] Wikipedia. The Stokes stream function. *Wikipedia*, 2012. http://en.wikipedia.org/wiki/Stokes_stream_function. 73

[52] N. Wirth. *Algorithm + data structure = programs*. Prentice Hall, NJ, USA, 1985. 2

List of example files

Makefile, 16
cavity.icc, 56
convect.cc, 79
convect_error.cc, 81
cosinusprod.icc, 24
cosinusprod_error.cc, 24
cosinusprod_laplace.icc, 22
cosinusrad.icc, 26
cosinusrad_laplace.icc, 26
dirichlet-nh.cc, 22
dirichlet.cc, 14
dirichlet.icc, 119
elasticity_criterion.icc, 53
elasticity_solve.icc, 53
embankment.cc, 47
embankment.icc, 47
embankment_adapt.cc, 52
eta.icc, 118
heat.cc, 75
helmholtz_band.cc, 109
helmholtz_band_iterative.cc, 107
helmholtz_s.cc, 96
incompressible-elasticity.cc, 64
laplace_band.cc, 110
laplace_s.cc, 101
level_set_sphere.cc, 104
navier_stokes_cavity.cc, 86
navier_stokes_criterion.icc, 87
navier_stokes_solve.icc, 85
neumann-laplace.cc, 34
neumann-nh.cc, 30
nu.icc, 127
p_laplacian.h, 125
p_laplacian1.icc, 126
p_laplacian2.icc, 127
p_laplacian_circle.icc, 133
p_laplacian_damped_newton.cc, 131
p_laplacian_error.cc, 133
p_laplacian_fixed_point.cc, 116
p_laplacian_newton.cc, 125
poiseuille.h, 68
proj_band.cc, 108
robin.cc, 32
rotating-hill.h, 80
sinusprod_helmholtz.icc, 31
sphere.icc, 97
stokes_cavity.cc, 56
stokes_contraction_bubble.cc, 67
streamf_cavity.cc, 61
streamf_contraction.cc, 70
stress.cc, 49
torus.icc, 101
transmission.cc, 38
vortex_position.cc, 91
vorticity.cc, 59
cavity.icc, 87
contraction.mshcad, 68, 72
convect.cc, 86
cosinusrad_error.cc, 26
cube.mshcad, 144
dirichlet_nh_ball.cc, 26
helmholtz_s_error.cc, 99
line.mshcad, 55, 142
neumann-nh.cc, 96
square.bamgcad, 54, 89, 141
square.dmn, 54, 89, 141
square.mshcad, 143
stokes_contraction.cc, 70
streamf_cavity.cc, 89
streamf_contraction.cc, 72
stress.cc, 74
torus.mshcad, 103

List of commands

bamg2geo, 141
bamg, 54, 89, 141
    -splitpbedge, 120
branch, 80
    -gnuplot, 78
    -paraview, 78, 83
    -umax, 78
convect, 80
ffmpeg, 78
field, 17, 148
    -, 19
    -bw, 18, 70, 73, 89
    -catchmark, 58
    -comp, 49, 52, 73, 90
    -cut, 73, 90, 119
    -elevation, 18, 51, 108, 119
    -fill, 49
    -gray, 18
    -iso, 52
    -max, 71, 90
    -mayavi, 18, 49
    -min, 90
    -n-iso, 70
    -n-iso-negative, 70, 73, 89
    -noclean, 62
    -noexecute, 62
    -nofill, 18, 49
    -normal, 73, 90, 119
    -origin, 73, 90, 119
    -proj, 51
    -scale, 65, 89
    -stereo, 18, 20, 49, 51
    -velocity, 58, 89
geo, 17
    -cut, 19
    -full, 19
    -stereo, 19
    -subdivide, 98
gmsh, 55, 68, 72, 97, 103, 120, 142
gnuplot, 17, 19, 40, 49, 78, 81, 142
gzip, 54
library
    boost, 15, 35, 147
    CGAL, computational geometry, 79
    MPI, message passing interface, 15, 147
    STL, standard template library, 80
make, 16
man, 19
mayavi, 18, 20, 62, 142
mkgeo_ball, 97
    -e, 97
    -q, 26
    -s, 97
    -t, 26, 97
mkgeo_grid, 17, 80, 120
    -H, 20
    -T, 19, 83
    -a, 81
    -b, 81
    -c, 83
    -d, 83
    -e, 19
    -f, 83
    -g, 83
    -q, 20
    -region, 40
    -t, 17
    -zr, 71
mkgeo_ugrid, 26, 120
mpirun, 17, 35, 41
msh2geo, 68, 142, 144
    -zr, 72
paraview, 78, 83
rheolef-config, 13
    -check, 13
    -docdir, 13
    -exampledir, 13
sed, 26
visualization
    mesh, 17
      deformed, 49
vlc, 78
zcat, 89

Index

approximation, 14
  P0, 51
  P1b-P1, 65
  P1d, 51
  P1, 15, 47, 51, 64, 65, 84
  P2-P1, Taylor-Hood, 56, 65, 66, 70
  P2, 15, 19, 47, 51, 64, 84
  Pk, 15, 19, 147
  bubble, 65
  discontinuous, 39, 50, 51, 59
  high-order, 15, 19, 41, 147
  isoparametric, 98
  mixed, 56
argc, argv, command line arguments, 15, 147

BDF2 scheme, 84
benchmark
  driven cavity flow, 55, 84
  Dziuk-Elliott-Heine on a sphere, 96
  embankment, 45
  flow in an abrupt contraction, 66, 72
  Olshanskii-Reusken-Grande on a torus, 102
  rotating hill, 79
boundary condition
  Dirichlet, 13, 21, 39, 45, 55, 63, 84, 115, 124
  mixed, 63, 66, 72
  Neumann, 29, 33, 39, 45, 63
  Poiseuille flow, 68
  Robin, 31

class
  `Float`, 23
  `adapt_option_type`, 54
  `band`, 107
  `branch`, 76, 82
  `characteristic`, 79
  `communicator`, 35
  `csr<T>`, 35
  `doubledouble`, 23
  `environment`, 147
  `eye`, 108
  `field_functor`, 23, 149
  `field`, 14, 148
  `form_option_type`, 67, 72
  `form`, 14
  `geo`, 14
  `idiststream`, 148
  `level_set_option_type`, 104
  `odiststream`, 53, 148
  `point`, 23
  `quadrature_option_type`, 79, 118, 133
  `solver_abtb`, 86, 148
  `solver`, 16, 57, 148
  `space`, 14, 148
  `vec<T>`, 35
compilation, 16
convergence
  error
    versus mesh, 24, 99, 133
    versus polynomial degree, 24, 99, 133
  residue
    rate, 120, 121
    super-linear, 128
coordinate system
  axisymmetric, 68, 70, 71
  Cartesian, 22, 40
  spherical, 96
  torus, 102

directory of example files, 13, 68, 70
distributed computation, 15, 17, 35, 147

element shape, 41
error analysis, 24, 82, 99, 133

file format
  '`.bamgcad`' bamg geometry, 54, 141
  '`.bamg`' bamg mesh, 54, 141
  '`.branch`' family of fields, 77
  '`.dmn`' domain names, 141
  '`.field`' field, 17, 148
  '`.field`' multi-component field, 49, 148
  '`.geo`' mesh, 17, 68, 71, 141, 142, 144, 148
  '`.gz`' gzip compressed file, 54
  '`.jpg`' joint photographic experts group, 78

'`.mov`' quicktime video file, 78
'`.mshcad`' gmsh geometry, 55, 68, 72, 103, 142
'`.msh`' gmsh mesh, 55, 68, 72, 142
'`.vtk`' vtk file, 62, 78
form
$(\eta\nabla u).\nabla v$, 126
$2D(\mathbf{u}) : D(\mathbf{v})$, 56, 63
$2D(\mathbf{u}) : D(\mathbf{v}) + \lambda \mathrm{div}\,\mathbf{u}\,\mathrm{div}\,\mathbf{v}$, 47
$2D(\mathbf{u}) : D(\mathbf{v}) + \mathbf{u}.\mathbf{v}$, 85
$\eta\nabla u.\nabla v$, 39, 117
$\nabla_s u.\nabla_s v + uv$, 96
$\nabla u.\nabla v$, 14
$\nabla u.\nabla v + uv$, 31
uv, 24
$\mathrm{bcurl}(\mathbf{u}).\boldsymbol{\xi}$, 70
$\mathrm{curl}(\mathbf{u}).\boldsymbol{\xi}$, 58
$\mathrm{div}(\mathbf{u})\,q$, 56, 63
energy, 14, 38, 115
product, 67
weighted, 39, 117
quadrature formula, 118, 126
tensorial weight, 126
Fréchet derivative, 124
function
`adapt`, 52, 54
`catchmark`, 35, 47, 58, 76
`compose`, 79, 87, 117, 126
`damped_newton`, 131
`diag`, 108
`field_functor`, 133
`grad`, 117, 126
`integrate`, 15, 31, 67, 70, 79, 96, 117, 126, 133, 149
on a band, 108
`interpolate`, 23, 53, 68, 82, 149
`ldlt`, 35
`level_set`, 104
`newton`, 125
`norm2`, 53, 87, 117
`riesz`, 149
`sqr`, 53
class-function object, 22, 80, 127

geometry
axisymmetric, 68, 70
circle, 26, 97
contraction, 66, 72
cube, 19, 144
curved, 98
line, 19, 142
sphere, 96
square, 17, 143
surface, 95
curvature, 140
torus, 103
Green formula, 95, 139

Lagrange
interpolation, 22, 23, 30, 34
multiplier, 33, 57
node, 15
Lamé coefficients, 45

Makefile, 16
matrix
bloc-diagonal
inverse, 67
block structure, 16
concatenation, 35
diagonal, 108
factorization
Choleski, 16
identity, 108
indefinite, 35
singular, 35, 108
sparse, 35
mesh, 14, 141
adaptation, 147
anisotropic, 52, 87
connected components, 109
generation, 68, 72, 141
method
characteristic, 78, 84, 147
conjugate gradient algorithm, 16, 33, 56, 64
Euler implicit scheme, 75, 78
fixed-point, 116, 128
relaxation, 116, 122
level set, 95, 104
banded, 107
minres algorithm, 33, 108
Newton, 124
damped, 130

namespace
rheolef, 14, 147
std, 14
norm
in $W^{-1,p}$, 120
discrete version, 120
in $W^{1,p}$, 115
in $W_0^{1,p}$, 115

operator
adjoint, 130
curl, 58
divergence, 45

gradient, 45
symmetric part, 45
Helmholtz, 29
Helmholtz-Beltrami, 95
Laplace, 13
Laplace-Beltrami, 95

parallel computation, 15, 17, 35
polar coordinate system, 71
preconditioner, 56
Choleski incomplete factorization, 16
for nearly incompressible elasticity, 64
for Stokes problem, 56
problem
Helmholtz, 29
Navier-Stokes, 84
Poisson, 13, 21, 31, 33, 75, 115, 117
Stokes, 55, 65, 84
convection-diffusion, 78
elasticity, 45
heat, 75
linear tangent, 124
nonlinear, 84
p-Laplacian, 115
stabilized Stokes, 67
transmission, 38
projection, 51, 60

quadrature formula, 118
quadrature formulae
Gauss, 80
Gauss-Lobatto, 80

region, 38, 40
residual term, 120, 124

singular solution, 60
space
$W^{-1,p}$, 115
$W^{-1,p}$, dual of $W_0^{1,p}$, 120
$W^{1,p}$, 115
$W_0^{1,p}$, 115
dual, 120
duality bracket $\langle .,. \rangle$, 120
weighted (axisymmetric), 71
stabilization, 63
stream function, 60, 70, 89
axisymmetric, 71

tensor
Cauchy stress, 45, 74
field, 51
rate of deformation, 74
visualization as ellipsoid, 50

unknow and blocked components, 15

variable
`derr`, 148
`din`, 148
`dout`, 148
visualization
animation, 78
elevation view, 18, 119
stereoscopic anaglyph, 18, 49
vortex, 70, 73
vorticity, 58