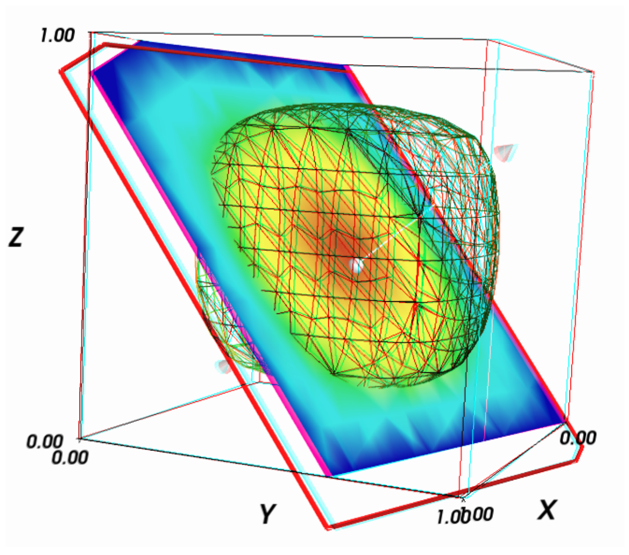
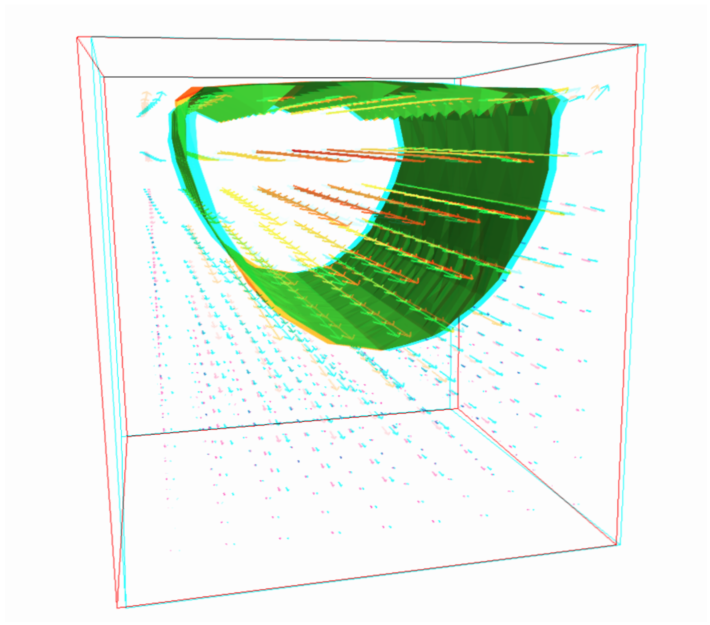
Efficient C++ finite element computing with Rheolef

P. Saramito and J. Etienne

version 5.93 update 22 March 2011





`Rheolef` is a computer environment that serves as a convenient *laboratory* for computations involving finite element methods. It provides a set of unix commands and C++ algorithms and containers. In particular, this environment allows the user to express a partial derivative problem in terms of finite element **spaces**, discrete **fields**, bilinear **forms**, geometries and meshes. `Rheolef` is the only one environment to our knowledge that bases on such powerful variational concepts as basic data structures.

This set of data structure is completed by the most up-to-date algorithms: preconditioned solvers for incompressible elasticity, Stokes and Navier-Stokes flows, characteristic method for convection dominated heat problems, . . .

All examples presented along the present manual are available in the `example/` directory of the `rheolef` source distribution: the directory where examples are available is given by the following unix command:

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

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 `Pierre.Saramito@imag.fr`.

Contents

Part I

Getting started with simple problems

Chapter 1

Introduction to rheolef

`rheolef` provides build-in classes for solving some linear and non-linear partial differential equations, based on the finite element method (see e.g. [1]). All example files presented along the present manual are available in the `examples/` directory of the `rheolef` distribution, e.g. `/usr/local/share/doc/rheolef/examples/`. Please, replace the prefix `/usr/local` by the suitable prefix used for your installation, as returned by the command `rheolef-config --prefix`. Before to run examples, please check your Rheolef installation with:

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

1.1 The model problem

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

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

The variational formulation of this problem expresses:
(VF): find $u \in H^1_0(\Omega)$ such that:

$$a(u, v) = m(1, v),\ \forall v \in H^1_0(\Omega)$$

where the bilinear forms $a(.,.)$ and $m(.,.)$ are defined by

$$\begin{aligned} a(u, v) &= \int_\Omega \nabla u.\nabla v\, dx, \quad \forall u, v \in H^1_0(\Omega) \\ m(u, v) &= \int_\Omega uv\, dx, \quad \forall u, v \in L^2(\Omega) \end{aligned}$$

The bilinear form $a(.,.)$ defines a scalar product in $H^1_0(\Omega)$ and is related to the *energy* form. This form is associated to the $-\Delta$ operator. The $m(.,.)$ is here the classical scalar product on $L^2(\Omega)$, and is related to the *mass* form.

1.2 Approximation

Let us introduce a mesh $\mathcal{T}_h$ of Ω and the finite dimensional space X_h of continuous picewise 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:*

$$a(u_h, v_h) = m(1, v_h)\ \forall v_h \in V_h$$

By developping 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.

1.3 File ‘`dirichlet.cc`’

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
int main(int argc, char**argv) {
  geo omega (argv[1]);
  space Xh (omega, argv[2]);
  Xh.block ("boundary");
  form a (Xh, Xh, "grad_grad");
  form m (Xh, Xh, "mass");
  field fh (Xh, 1);
  field uh (Xh);
  uh ["boundary"] = 0;
  if (omega.dimension() < 3) {
    ssk<Float> fact = ldlt(a.uu);
    uh.u = fact.solve (m.uu*fh.u + m.ub*fh.b - a.ub*uh.b);
  } else {
    size_t max_iter = 10000;
    Float tol = 1e-15;
    uh.u = 0;
    int status = pcg (a.uu, uh.u, m.uu*fh.u + m.ub*fh.b - a.ub*uh.b,
                 ic0(a.uu), max_iter, tol, &cerr);
  }
  cout << uh;
  return 0;
}
```

1.4 Comments

This code applies for both one, two or three dimensional meshes and for both picewise 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.

```
    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` or `P2`.

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

The homgeneous Dirichlet conditions are declared on the boundary.

```
form a (Xh, Xh, "grad_grad");
```

The form $a(.,.)$ is the energy form.

```
form m (Xh, Xh, "mass");
```

The form $m(.,.)$ is the mass form.

```
field fh (Xh, 1);
```

The field `fh` is initialized with constant value 1.

```
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 $(\varphi_i)_{0\leq i<\dim(X_h)}$ be the basis of X_h associated to the Lagrange nodes, i.e. 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 picewise 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 *bloked* 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 restriction to $X_h \times X_h$ of the mass form:

$$m(f_h, v_h) = \begin{pmatrix} \texttt{vh.u} & \texttt{vh.b} \end{pmatrix} \begin{pmatrix} \texttt{m.uu} & \texttt{m.ub} \\ \texttt{m.bu} & \texttt{m.bb} \end{pmatrix} \begin{pmatrix} \texttt{fh.u} \\ \texttt{fh.b} \end{pmatrix}$$

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

$$\begin{pmatrix} \mathtt{vh.u} & \mathtt{vh.b} \end{pmatrix} \begin{pmatrix} \mathtt{a.uu} & \mathtt{a.ub} \\ \mathtt{a.bu} & \mathtt{a.bb} \end{pmatrix} \begin{pmatrix} \mathtt{uh.u} \\ \mathtt{uh.b} \end{pmatrix} = \begin{pmatrix} \mathtt{vh.u} & \mathtt{vh.b} \end{pmatrix} \begin{pmatrix} \mathtt{m.uu} & \mathtt{m.ub} \\ \mathtt{m.bu} & \mathtt{m.bb} \end{pmatrix} \begin{pmatrix} \mathtt{fh.u} \\ \mathtt{fh.b} \end{pmatrix}$$

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

$$\mathtt{a.uu} * \mathtt{uh.u} = \mathtt{m.uu} * \mathtt{fh.u} + \mathtt{m.ub} * \mathtt{fh.b} - \mathtt{a.ub} * \mathtt{uh.b}$$

The LDL^t factorization of the `a.uu` matrix is then performed:

```
ssk<Float> fact = ldlt(a.uu);
```

Then, the `unkown part` is solved:

```
uh.u = fact.solve(m.uu*fh.u + m.ub*fh.b - a.ub*uh.b);
```

When $d > 3$, an iterative strategy is prefered for solving the linear system:

```
uh.u = 0;
int status = pcg (a.uu, uh.u, m.uu*fh.u + m.ub*fh.b - a.ub*uh.b,
                  ic0(a.uu), max_iter, tol, &cerr);
```

The iterative algorithm is the conjugate gradient `pcg`, preconditionned by the incomplete Cfholeski factorisation `ic0`. Finaly, the field is printed to standard output:

```
cout << uh;
```

1.5 How to compile the code

First, create a `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.

1.6 How to run the program

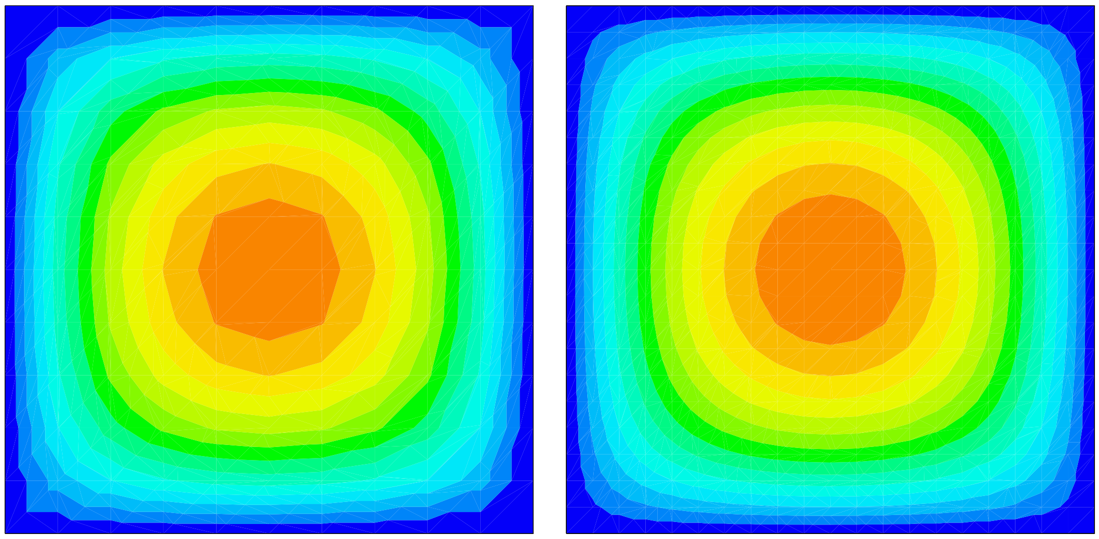


Figure 1.1: Solution of the model problem for $N = 2$: (left) P_1 element; (right) P_2 element.

Enter the comands:

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

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

```
./dirichlet square.geo P1 | field -
```

The result is piped to the post-treatment, performed by the `field` command. The `field` command reads this result on its standard input and presents it in a graphical form. A temporary file could also be used to store the result, using the '`.field`' file format:

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

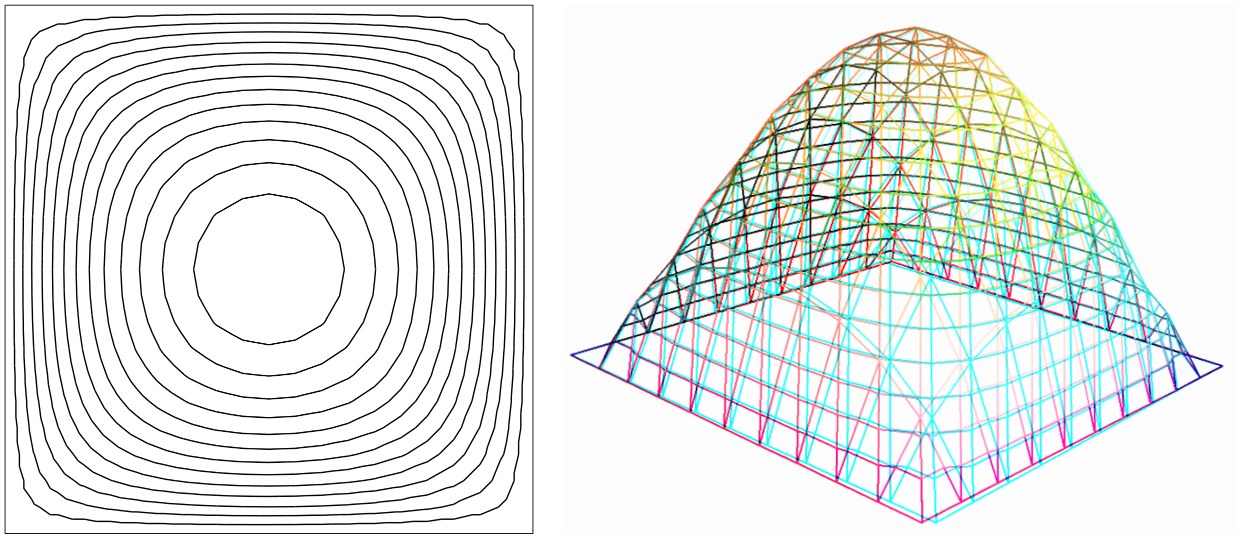


Figure 1.2: Alternative representations of the solution of the model problem ($N = 2$ and the P_1 element): (left) in black-and-white; (right) in elevation and stereoscopic anagyph mode.

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

```
field square.field -black-and-white
field square.field -gray
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.

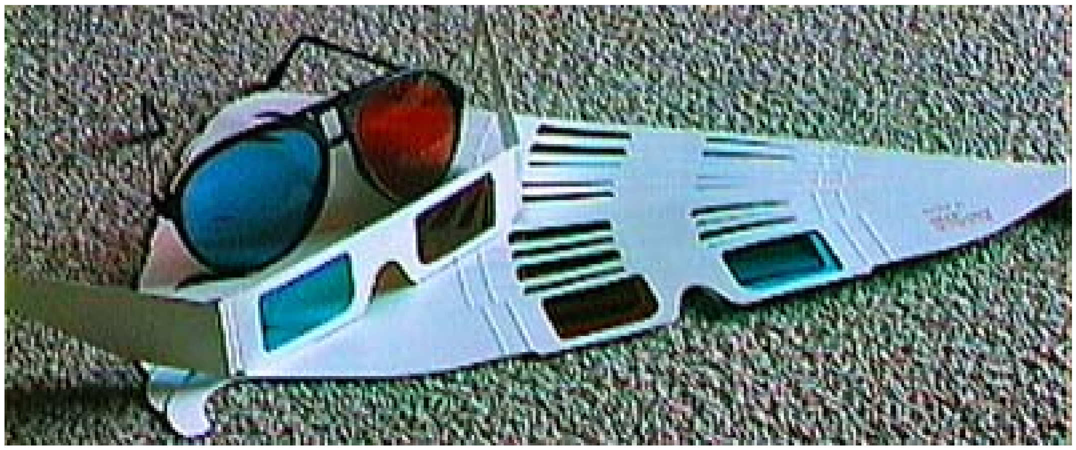


Figure 1.3: Red-cyan anaglyph glasses for the stereoscopic visualization.

See `http://en.wikipedia.org/wiki/Anaglyph_image` for more and `http://www.alpes-stereo.com/lunettes.html` for how to find anaglyphe 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
```

Turning to the P2 approximation simply writes:

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

Fig. 1.1.right shows the result. Now, let us condider a mono-dimensional problem $\Omega =]0,1[$:

```
mkgeo_grid -e 10 -boundary > 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 -
```

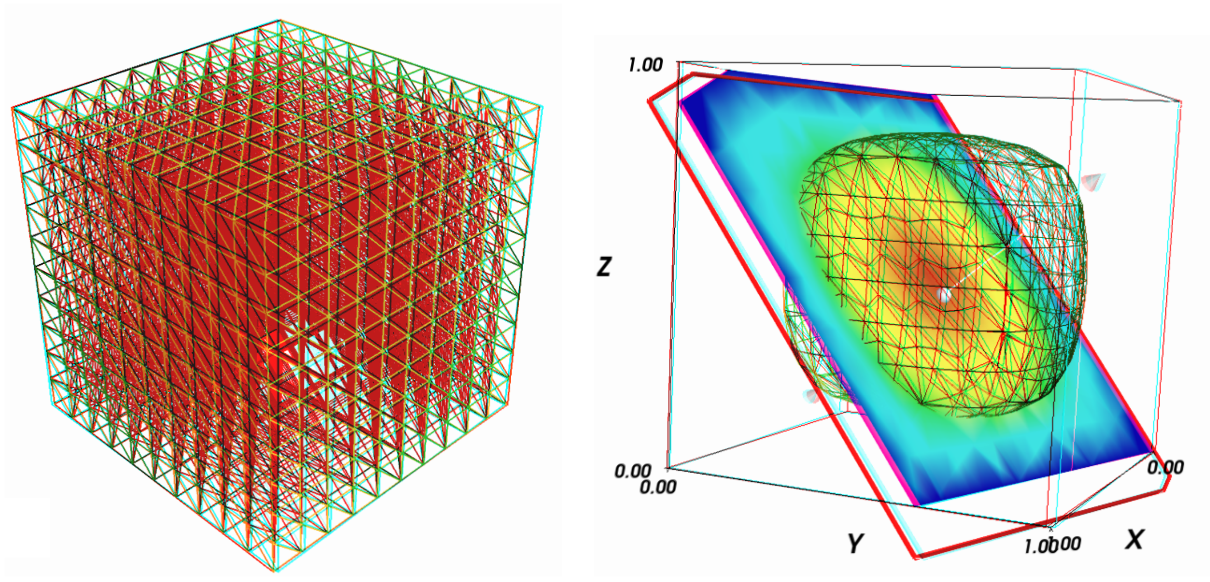


Figure 1.4: Solution of the model problem for $N = 3$ and the P_1 element : (left) mesh; (right) isovalue, cut planes and stereo anaglyph renderings.

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

```
mkgeo_grid -T 10 -boundary > box.geo
geo box.geo
geo box.geo -stereo -full
geo box.geo -stereo -cut
```

The previous commands draw the mesh with all internal edges, stereoscopic anaglyph and then with a 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.

Then, we perform the computation and the visualization:

```
./dirichlet box.geo P1 > box.field
field box.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 box.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 visualisation capacities of our environment, let us go back to the Dirichlet problem and perform the `P2` approximation:

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

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. [2], annex).

1.7 How to extend the program

A good unix command may check the validity of its arguments, and may have some default values. These points are left to the reader as an exercice.

1.8 Limitations

When using the P_2 interpolation and some non-polygonal domains (i.e. curved boundaries), the method works not optimaly: it converges in H_1 norm as $h^{3/2}$ instead of the optimal h^2 in the case of polygonal domains. The optimal convergence requires the use of isoparametric P2 elements (i.e. curved triangles): this will be a development in the future.

Chapter 2

Getting Started

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 example presents a special problem where the solution is defined up to a constant: the Neumann problem for the Laplace operator.

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.

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:
$(P_2)_h$ *find u, defined in Ω such that:*

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

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

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

where

$$\begin{array}{rcl} a(u,v) & = & \displaystyle\int_\Omega \nabla u.\nabla v \, dx \\ m(u,v) & = & \displaystyle\int_\Omega uv \, dx \\ V & = & \{v \in H^1(\Omega); \ v_{|\partial\Omega} = g\} \\ V_0 & = & H^1_0(\Omega) \end{array}$$

The computation of the right-hand side may be considered with attention since f is not *a priori* piecewise polynomial.

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:

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

The approximate problem writes:
$(VF_2)_h$: *find* $u_h \in V_h$ *such that:*

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

Notices that the evaluation of the right-hand side is performed by remplacing f by its Lagrange interpolation $\Pi_h(h)$ on Ω, while the boundary condition g is remplaced by its Lagrange interpolation $\pi_h(h)$ on $\partial\Omega$.
Let us choose $\Omega \subset R^N$, $N = 1, 2, 3$ and

$$\begin{array}{rcl} f(x) & = & N \sin(\sum_{i=1}^{N} x_i) \\ g(x) & = & \sin(\sum_{i=1}^{N} x_i) \end{array}$$

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

$$u(x) = \sin(\sum_{i=1}^{N} x_i)$$

. The following C++ code implement this problem in the `rheolef` environment.

File ‘`dirichlet-nh.cc`’

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

size_t N;
Float f (const point& x) { return 1.0*N*sin(x[0]+x[1]+x[2]); }
Float g (const point& x) { return sin(x[0]+x[1]+x[2]); }

int main(int argc, char**argv) {
  geo omega (argv[1]);
  N = omega.dimension();
  space Vh (omega, argv[2]);
  Vh.block ("boundary");
  form a(Vh, Vh, "grad_grad");
  form m(Vh, Vh, "mass");
  field uh (Vh);
  field fh = interpolate(Vh, f);
  const domain& boundary = omega["boundary"];
  space Wh (omega, boundary, argv[2]);
  uh["boundary"] = interpolate(Wh, g);
  if (omega.dimension() < 3) {
    ssk<Float> fact = ldlt(a.uu);
    uh.u = fact.solve (m.uu*fh.u + m.ub*fh.b - a.ub*uh.b);
  } else {
    size_t max_iter = 10000;
    Float tol = 1e-15;
    uh.u = 0;
    int status = pcg (a.uu, uh.u, m.uu*fh.u + m.ub*fh.b - a.ub*uh.b,
                 ic0(a.uu), max_iter, tol, &cerr);
  }
  cout << uh;
  return 0;
}
```

Comments

The code looks like the previous one, related to homogeneous boundary conditions. Let us comments the changes. The class `point` describes the coordinates of a point $(x_1, \ldots, x_N) \in R^N$ as a N-uplet of `Float`. The `Float` type is usually a `double`. This type depends upon the `rheolef` configuration (see also Installing,,Configure options), and could represent some high precision floating point class such as `doubledouble` (quadruple) or `bigfloat` (arbitrary).

```
field fh = interpolate(Vh,f);
```

This implements the Lagrange interpolation operator $\Pi_h(f)$.

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

The space of picewise P_k functions is builded. This space is suitable for the interpolation of g on the boundary:

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

The values of the degrees of freedom related to the boundary are stored into the field `u`, 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 remplacing `dirichlet` by `dirichlet-nh`:

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

for the bidimensional case:

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

and for the tridimensional case:

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

Here, the `P1` approximation can be remplaced by the `P2` one, by modifying the command-line argument.

2.2 Non-homogeneous Neumann boundary conditions for $I - \Delta$ 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:
$(P_3)_h$: *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 variationnal formulation of this problem expresses:
(VF_3) *find $u \in H^1(\Omega)$ such that:*

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

where

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

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:*

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

where

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

The treatment of the right-hand side is similar to the case of the non-homogeneous Dirichlet condition: the evaluation of the right-hand side is performed by remplacing f by its Lagrange interpolation $\Pi_h f$ on Ω, while the boundary condition g is remplaced by its Lagrange interpolation $\pi_h g$ on $\partial\Omega$.

Let us choose $\Omega \subset R^N$, $N = 1, 2, 3$ and

$$\begin{aligned} f(x) &= 1 + \frac{1}{2N}\sum_{i=1}^{N} x_i(1 - x_i) \\ g(x) &= -\frac{1}{2N} \end{aligned}$$

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

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

File ‘neumann.cc’

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

size_t N;
Float f (const point& x) { return 1+(0.5/N)*(x[0]*(1-x[0])+x[1]*(1-x[1])+x[2]*(1-x[2])); }
Float g (const point& x) { return -0.5/N; }

int main(int argc, char**argv) {
  geo omega (argv[1]);
  N = omega.dimension();
  space Vh (omega, argv[2]);
  form m (Vh, Vh, "mass");
  form a (Vh, Vh, "grad_grad");
  a = a+m;
  field fh = interpolate(Vh, f);
  space Wh (omega, omega["boundary"], argv[2]);
  field gh = interpolate(Wh, g);
  form mb (Wh, Vh, "mass");
  field uh(Vh);
  if (omega.dimension() < 3) {
    ssk<Float> fact = ldlt(a.uu);
    uh.u = fact.solve(m.uu*fh.u + m.ub*fh.b + mb.uu*gh.u + mb.ub*gh.b - a.ub*uh.b);
  } else {
    size_t max_iter = 10000;
    Float tol = 1e-15;
    uh.u = 0;
    int status = pcg (a.uu, uh.u, m.uu*fh.u + m.ub*fh.b + mb.uu*gh.u + mb.ub*gh.b - a.ub*uh.b,
                      ic0(a.uu), max_iter, tol, &cerr);
  }
  cout << uh;
  return 0;
}
```

Comments

The code looks like the previous one. Let us comments the changes.

```
space Wh (omega, omega["boundary"], argv[2]);
field gh = interpolate(Wh, g);
```

The space W_h of piecewise P_k continuous functions defined on the boundary of Ω is defined here. Then, the Lagrange interpolation is performed on g.

```
form mb (Wh, Vh, "mass");
```

The form `mb` implements the restriction of the L^2 scalar product on W_h.

```
uh.u = fact.solve(m.uu*fh.u + m.ub*fh.b + mb.uu*gh.u + mb.ub*gh.b - a.ub*uh.b);
```

The direct solver related to `uh.u` is called on the system:

$$\begin{pmatrix} \mathtt{a.uu} & \mathtt{a.ub} \\ \mathtt{a.bu} & \mathtt{a.bb} \end{pmatrix} \begin{pmatrix} \mathtt{uh.u} \\ \mathtt{uh.b} \end{pmatrix} = \begin{pmatrix} \mathtt{m.uu} & \mathtt{m.ub} \\ \mathtt{m.bu} & \mathtt{m.bb} \end{pmatrix} \begin{pmatrix} \mathtt{fh.u} \\ \mathtt{fh.b} \end{pmatrix} + \begin{pmatrix} \mathtt{mb.uu} & \mathtt{mb.ub} \\ \mathtt{mb.bu} & \mathtt{mb.bb} \end{pmatrix} \begin{pmatrix} \mathtt{gh.u} \\ \mathtt{gh.b} \end{pmatrix}$$

Reduced to the `uh.u` component, this problem writes also:

```
a.uu*uh.u = m.uu*fh.u + m.ub*fh.b + mb.uu*gh.u + mb.ub*gh.b - a.ub*uh.b
```

Notices that `uh.b` has here a null size.

2.2.1 How to run the program

First, compile the program:

```
make neumann
```

Running the program is obtained from the homogeneous Dirichlet case, by remplacing `dirichlet` by `neumann`.

2.3 The Robin boundary conditions

Formulation

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

$$\begin{aligned} -\Delta u &= 1 \text{ in } \Omega \\ \frac{\partial u}{\partial n} + u &= g \text{ on } \partial\Omega \end{aligned}$$

The variationnal 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

$$\begin{array}{rcl} a(u,v) &=& \displaystyle\int_\Omega \nabla u.\nabla v\,dx + \int_{\partial\Omega} uv\,ds \\ l(v) &=& m(1,v) + m_b(g,v) \\ m(u,v) &=& \displaystyle\int_\Omega uv\,dx \\ m_b(g,v) &=& \displaystyle\int_{\partial\Omega} gv\,ds \end{array}$$

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:*

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

where

$$l_h(v_h) = m(1, v_h) + m_b(\pi_h g, v_h)$$

Notices that the evaluation of the right-hand side is performed by remplacing the boundary condition g by its Lagrange interpolation $\pi_h(h)$ on $\partial\Omega$.
Let us choose $\Omega =]0,1[^N$, $N = 1,2,3$ and

$$g(x) \quad = \quad \frac{1}{2N}\left(-1 + \sum_{i=1}^{N} x_i(1-x_i)\right)$$

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

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

The following C++ code implement this problem in the `rheolef` environment.

File ‘robin.cc’

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

size_t N;
Float u_ex (const point& x) { return 0.5*(x[0]*(1-x[0]) + x[1]*(1-x[1]))/Float(N); }
Float g    (const point& x) { return u_ex(x) - 0.5/N; }

int main(int argc, char**argv) {
  geo omega (argv[1]);
  N = omega.dimension();
  domain boundary = omega["boundary"];
  space Xh (omega, argv[2]);
  space Wh (omega, boundary, argv[2]);
  form a  (Xh, Xh, "grad_grad");
  form ab (Xh, Xh, "mass", boundary);
  a = a + ab;
  form m  (Xh, Xh, "mass");
  form mb (Wh, Xh, "mass") ;
  field fh (Xh, 1);
  field gh = interpolate(Wh, g);
  field uh (Xh);
  if (omega.dimension() < 3) {
    ssk<Float> fact = ldlt(a.uu);
    uh.u = fact.solve(m.uu*fh.u + m.ub*fh.b + mb.uu*gh.u + mb.ub*gh.b - a.ub*uh.b);
  } else {
    size_t max_iter = 10000;
    Float tol = 1e-15;
    uh.u = 0;
    int status = pcg (a.uu, uh.u, m.uu*fh.u + m.ub*fh.b + mb.uu*gh.u + mb.ub*gh.b - a.ub*uh.b,
                  ic0(a.uu), max_iter, tol, &cerr);
  }
  cout << uh;
  return 0;
}
```

Comments

The code looks like the previous one. Let us comments the changes.

```
    form ab (Xh, Xh, "mass", boundary);
    a = a + ab;
```

The boundary contribution to the $a(.,.)$ form on $X_h \times X_h$ is introduced.
Notices that the exact solution is a second-order polynomial. Thus, the P_2 approximation furnishes the exact solution, up to the machine precision.

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 remplacing `dirichlet` by `robin`.

2.4 Non-homogeneous Neumann boundary conditions for the Laplace operator

Formulation

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:*

$$\begin{array}{rcl} -\Delta u & = & f \text{ in } \Omega \\ \dfrac{\partial u}{\partial n} & = & g \text{ on } \partial\Omega \end{array}$$

Since this problem only involves the derivatives of u, it is clear that its solution is never unique [3, p. 11]. A discrete version of this problem could be solved iterative by the conjugate gradient or the MINRES algorithm [4]. In order to solve it by a direct method, we turn the difficuty 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 variationnal formulation of this problem expresses:
(VF_5): *find* $u \in V$ *such that:*

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

where

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

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:

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

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 satifies 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:*

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

where

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

File ‘neumann-laplace.cc’

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
#include "neumann-laplace-assembly.h"
size_t N;
Float f (const point& x) { return 1; }
Float g (const point& x) { return -0.5/N; }
int main(int argc, char**argv) {
  geo omega (argv[1]);
  N = omega.dimension();
  space Xh (omega, argv[2]);
  form m (Xh, Xh, "mass");
  form a (Xh, Xh, "grad_grad");
  field b = m*field(Xh,1.0);
  csr<Float> A = neumann_laplace_assembly (a.uu, b.u);
  field fh = interpolate(Xh, f);
  space Wh (omega, omega["boundary"], argv[2]);
  field gh = interpolate(Wh, g);
  form mb (Wh, Xh, "mass");
  field lh = m*fh + mb*gh;
  vec<Float> L(lh.u.size()+1, 0.0);
  for (size_t i = 0; i < L.size()-1; i++) L.at(i) = lh.u.at(i);
  L.at(L.size()-1) = 0;
  vec<Float> U (L.size());
  ssk<Float> fact_A = ldlt(A);
  U = fact_A.solve(L);
  field uh(Xh);
  for (size_t i = 0; i < U.size()-1; i++) uh.u.at(i) = U.at(i);
  Float lambda = U.at(U.size()-1);
  cout << setprecision(numeric_limits<Float>::digits10)
       << catchmark("u") << uh
       << catchmark("lambda") << lambda << endl;
}
```

Comments

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

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

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

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

The problem admits the following matrix form:

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

The matrix $\mathcal{A}$ is symetric and non-singular, but indefinite : it admits eigenvalues that are eitehr strictly positive or strictly negative. It can be factored in LDL^T form:

```
    csr<Float> A = neumann_laplace_assembly (a.uu, b.u);
    ssk<Float> fact_A = ldlt(A);
```

The rest of the code is mostly standard. The statement

```
    cout << setprecision(numeric_limits<Float>::digits10)
         << catchmark("u") << uh
         << catchmark("lambda") << lambda << endl;
```

writes the solution (u_h, λ) with full digit precision and labeled format, suitable for post-traitement, visualization and error analysis. It remain to look at the $\mathcal{A}$ matrix assembly function.

File ‘`neumann-laplace-assembly.h`’

```
template <class T>
csr<T> neumann_laplace_assembly (const csr<T>& a, const vec<T>& b) {
  size_t n = a.nrow();
  asr<T> A (n+1, n+1);
  Array<size_t>::const_iterator ia = a.ia().begin();
  Array<size_t>::const_iterator ja = a.ja().begin();
  typename Array<T>::const_iterator va = a.a().begin();
  typename Array<T>::const_iterator vb = b.begin();
  for (size_t i = 0; i < n; i++) {
    for (size_t p = ia[i]; p < ia[i+1]; p++) {
      A.entry(i,ja[p]) = va[p];
    }
    A.entry(i,n) = A.entry(n,i) = vb[i];
  }
  return csr<T>(A);
}
```

Comments

The function takes as input the a matrix in compressed sparse row format `csr` (see e.g. [5]), and the b vector. It uses a temporary associative sparse row data structure `asr` for $\mathcal{A}$ that is converted to `csr` at the end of the function.

2.4.1 How to run the program

As usual, enter:

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

Part II

Fluids and solids computations

Chapter 3

The linear elasticity and the Stokes problems

3.1 The linear elasticity problem

Formulation

The total Cauchy stress tensor expresses:

$$\sigma(\mathbf{u}) = \lambda\,\mathrm{div}(\mathbf{u}).I + 2\mu D(\mathbf{u})$$

where λ and μ are the Lamé coefficients. Let us consider the elasticity problem for the *embankment*, in $\Omega =]0,1[^N$, $N = 2, 3$. The problem writes:

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

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

where $\mathbf{f} = (0, -1)$ when $N = 2$ and $\mathbf{f} = (0, 0, -1)$ when $N = 3$. 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.

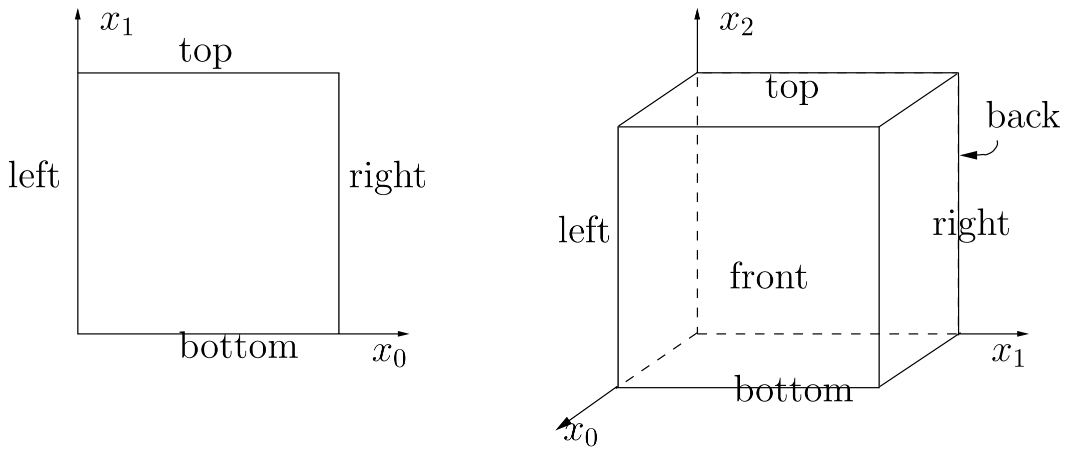


Figure 3.1: The boundary domains for the square and the cube.

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

For $N = 2$ we define the boundaries:

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

and for $N = 3$:

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

These boundaries are represented on Fig. 3.1.

The variational formulation of this problem expresses:

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

$$a(\mathbf{u}, \mathbf{v}) = m(\mathbf{f}, \mathbf{v}),\ \forall \mathbf{v} \in \mathbf{V},$$

where

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

Approximation

We introduce a mesh $\mathcal{T}_h$ of Ω and for $k = 1, 2$, the following finite dimensional spaces:

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

The approximate problem writes:

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

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

File ‘`embankment.cc`’

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
#include "embankment.h"
int main(int argc, char**argv) {
  const Float lambda = 1;
  geo  omega (argv[1]);
  space Vh = embankment_space (omega, argv[2]);
  field uh (Vh, 0.0), fh (Vh, 0.0);
  fh [omega.dimension()-1] = -1.0;
  form m  (Vh, Vh, "mass");
  form a1 (Vh, Vh, "div_div");
  form a2 (Vh, Vh, "2D_D");
  form a = lambda*a1 + a2;
  if (omega.dimension() < 3) {
    ssk<Float> fact = ldlt(a.uu);
    uh.u = fact.solve (m.uu*fh.u + m.ub*fh.b - a.ub*uh.b);
  } else {
    size_t max_iter = 10000;
    Float tol = 1e-15;
    uh.u = 0;
    int status = pcg (a.uu, uh.u, m.uu*fh.u + m.ub*fh.b - a.ub*uh.b,
                 ic0(a.uu), max_iter, tol, &cerr);
  }
  cout << catchmark("lambda")  << lambda << endl
       << catchmark("u")       << uh;
  return 0;
}
```

File ‘`embankment.h`’

```
space embankment_space (const geo& omega_h, std::string approx) {
  space Vh (omega_h, approx, "vector");
  Vh.block("bottom"); Vh.block("left");
  if (omega_h.dimension() == 3) {
    Vh.block("right");  Vh.block("back");
  }
  return Vh;
}
```

Comments

The space is defined in a separate file ‘`embankment.h`’, for subsequent reusage:

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

Note here the multi-components features for the velocity field **u**. The boundary condition contain a special case for tridimensionnal case. 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 [omega_h.dimension()-1] = -1.0;
```

Finally, the λ 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

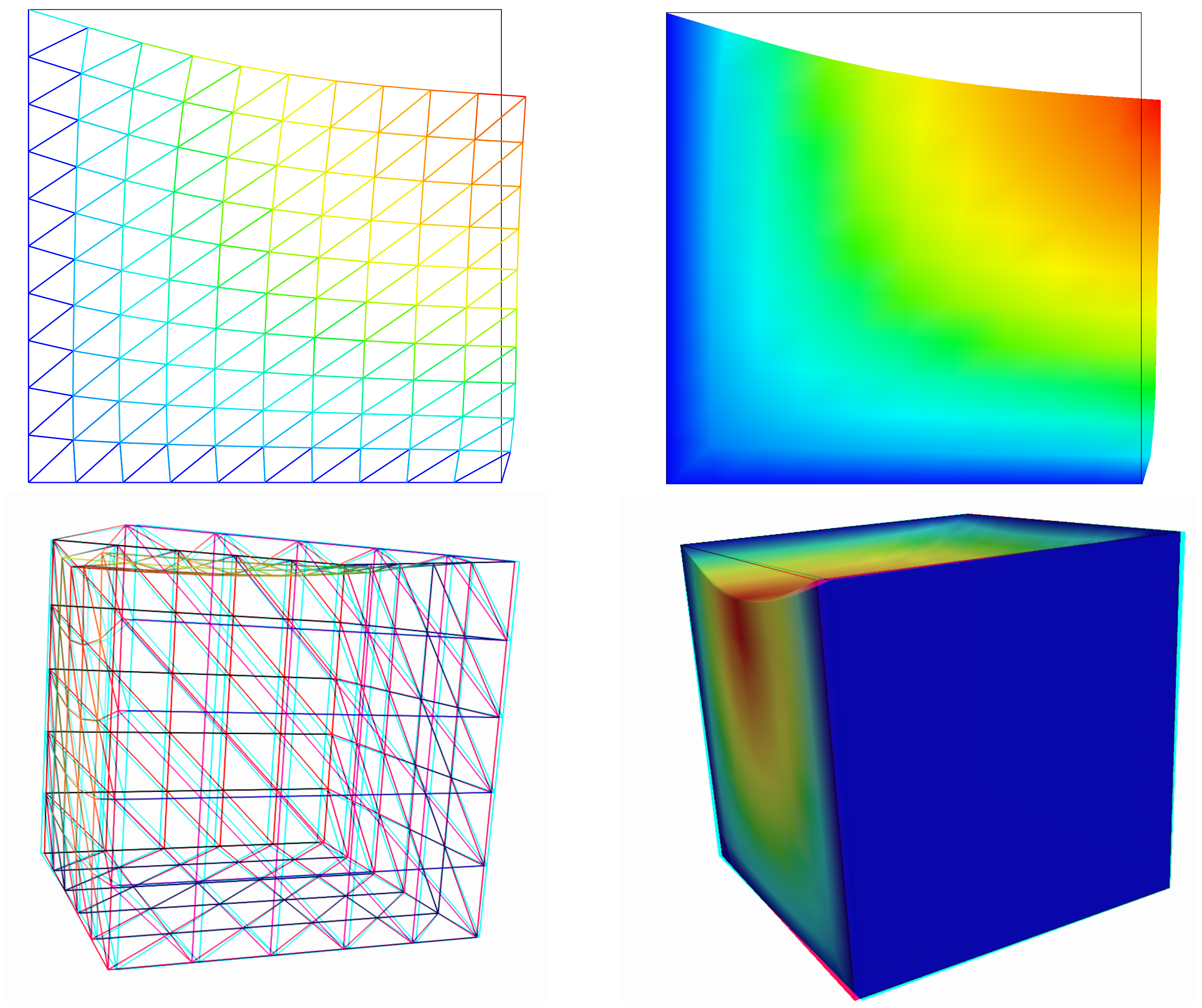


Figure 3.2: The linear elasticity for $\lambda = 1$ and $N = 2$ and $N = 3$: both wireframe and filled surfaces ; stereoscopic anaglyph mode for 3D solutions.

We assume that the previous code is contained in the file ‘`embankment.cc`’. Compile the program as usual (see page 10):

```
make embankment
```

and enter the comands:

```
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.mfield
```

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

```
mfield square-P1.mfield -u -deformation
mfield square-P1.mfield -u -deformation -fill
```

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

```
man mfield
```

The view is shown on Fig. 3.2. A specific field can be also selected for a scalar visualization:

```
mfield -u0 square-P1.mfield | field -
mfield -u1 square-P1.mfield | field -
```

Next, perform a P_2 approximation of the solution:

```
./embankment square.geo P2 > square-P2.mfield
mfield -u square-P2.mfield -deformation
```

Finally, let us consider the three dimensional case

```
mkgeo_grid -T 10 > cube.geo
./embankment cube.geo P1 > cube-P1.mfield
mfield -u -deformation cube-P1.mfield -stereo
mfield -u -deformation cube-P1.mfield -stereo -fill
```

The two last commands show the solution in 3D stereoscopic anaglyph mode. The graphic is represented on Fig. 3.2. The P_2 approximation writes:

```
./embankment cube.geo P2 > cube-P2.mfield
mfield -u -deformation cube-P2.mfield
```

3.2 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 `.mfield` file. Let us introduce:

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

This computation expresses:

find σ_h *such that:*

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

where

$$\begin{aligned} m(\sigma, \tau) &= \int_\Omega \sigma : \tau \, dx, \\ b(\tau, \mathbf{u}) &= \lambda \int_\Omega \mathrm{div}(\mathbf{u}) \, \mathrm{tr}\, \tau \, dx + \int_\Omega 2D(\mathbf{u}) : \tau \, dx, \\ \mathrm{tr}\, \tau &= \sum_{i=1}^{N} \tau_{ii}. \end{aligned}$$

File ‘stress.cc’

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

int main(int argc, char** argv) {
  Float lambda;
  field uh;
  cin >> catchmark("lambda") >> lambda
      >> catchmark("u")      >> uh;
  string approx = (uh.get_approx() == "P2") ? "P1d" : "P0";
  space Th  (uh.get_geo(), approx, "tensor");
  space Xh  (uh.get_geo(), approx);
  form two_D  (uh.get_space(), Th, "2D");
  form div    (uh.get_space(), Xh, "div");
  form inv_mt (Th, Th, "inv_mass");
  form inv_m  (Xh, Xh, "inv_mass");
  field q  = inv_m*(div*uh);
  field sh = inv_mt*(two_D*uh);
  for (size_t i = 0; i < uh.dimension(); i++)
      sh(i,i) += lambda*q;
  cout << catchmark("s") << sh;
  return 0;
}
```

Comments

First notice that this code applies for any deformation field, and is not rectricted to our *embankment* problem.

The `P0` and `P1d` stands for the piecewise constant and picewise linear discontinuous approximations, respectively. Since elements of T_h are discontinuous accross interelement boundaries, the mass operator is block-diagonal and can be inverted one time for all: this operation results in the `inv_mass` operator.

How to run the program

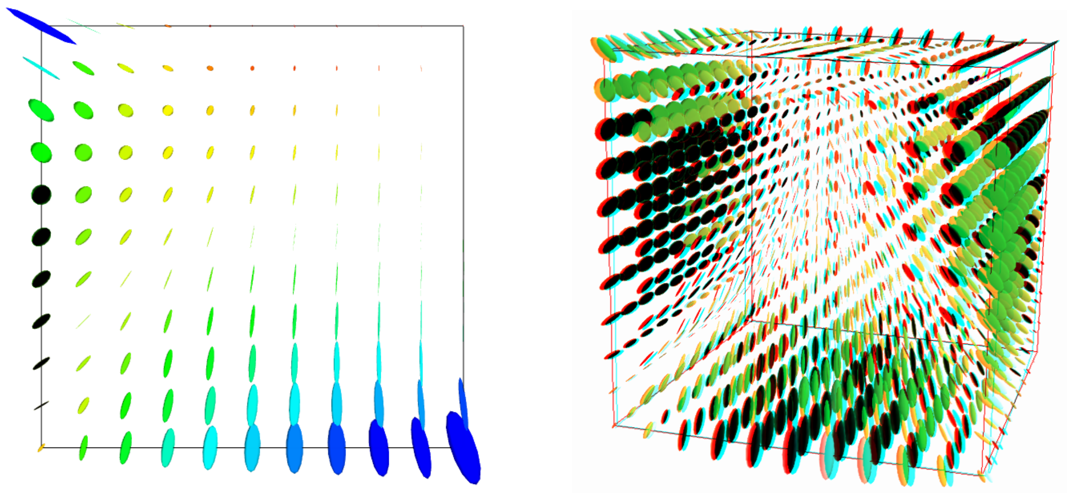


Figure 3.3: The stress tensor visualization (linear elasticity $\lambda = 1$).

First, compile the program:

```
make stress
```

The visualization for the stress tensor as ellipes writes:

```
./stress < square-P1.mfield | mfield -s -tensor -proj -
```

Conversely, the 3D visualization bases on ellipsoides:

```
./stress < cube-P1.mfield | mfield -s -tensor -proj -
```

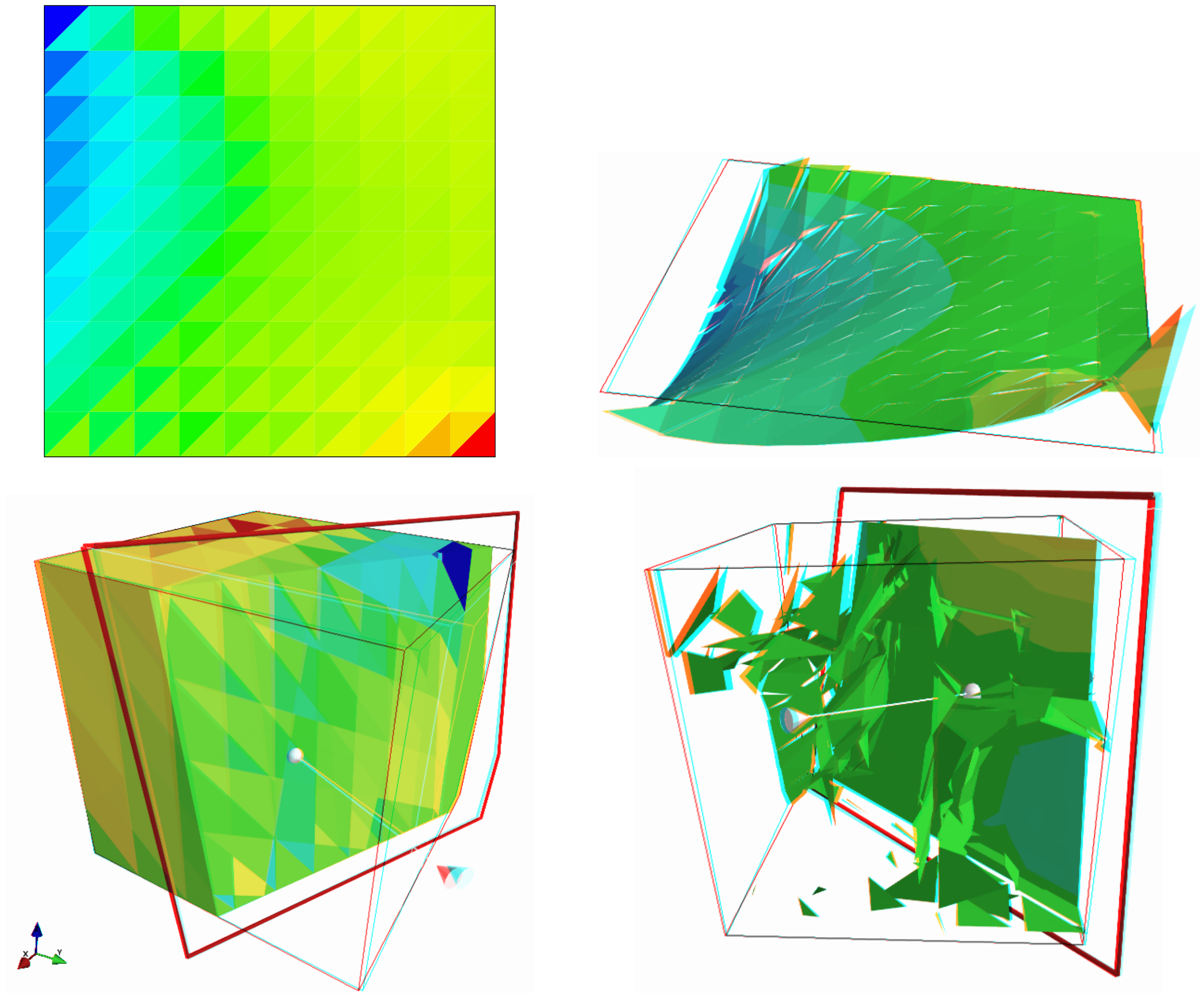


Figure 3.4: The σ_{01} stress component (linear elasticity $\lambda = 1$): $d = 2$ (top) and $d = 3$ (bottom) ; P_0 (left) and P_1 discontinuous approximation (right).

You can observe a discontinuous constant or piecewise linear representation of the approximate stress component σ_{00} (see Fig. 3.4):

```
./stress < square-P1.mfield | mfield -s01 - | field -
./stress < square-P2.mfield | mfield -s01 - | field -elevation -stereo -
```

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 field can be also projected on a continuous piecewise linear space, using the `-proj` option:

```
./stress < square-P1.mfield | mfield -s01 - | field -proj -elevation -stereo -
./stress < square-P2.mfield | mfield -s01 - | field -proj -elevation -stereo -
```

The tridimensionnal case writes simply (see Fig. 3.4):

```
./stress < cube-P1.mfield   | mfield -s01 - | field -stereo -
```

```
./stress < cube-P2.mfield   | mfield -s01 - | field -stereo -fill -
```

and also the P1-projected versions write:

```
./stress < cube-P1.mfield   | mfield -s01 - | field -proj -stereo -
./stress < cube-P2.mfield   | mfield -s01 - | field -proj -stereo -
```

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

3.3 Mesh adaptation

The main principle of the auto-adaptive mesh writes [6–11]:

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

The initial mesh is used to compute a first solution. The adaptive loop compute an *adaptive criteria*, denoted by `ch`, that depends upon the problem under consideration and the polynomial approximation used. Then, a new mesh is generated, based on this critera. A second solution on an adapted mesh can be constructed. The adaptation loop converges generaly in roughly 10 to 20 iterations.

Let us apply this principle to the elasticity problem.

File ‘`embankment-adapt-2d.cc`’

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
#include "embankment.h"
field criteria(Float lambda, const field& uh) {
  string approx = (uh.get_approx() == "P2") ? "P1d" : "P0";
  if (approx == "P0") return sqrt(sqr(uh[0])+sqr(uh[1]));
  space Th  (uh.get_geo(), approx, "tensor");
  space Xh  (uh.get_geo(), approx);
  form two_D  (uh.get_space(), Th, "2D");
  form div    (uh.get_space(), Xh, "div");
  form mt (Th, Th, "mass");
  form m  (Xh, Xh, "mass");
  form inv_mt (Th, Th, "inv_mass");
  form inv_m  (Xh, Xh, "inv_mass");
  field qh      = inv_m*(div*uh);
  field two_Duh = inv_mt*(two_D*uh);
  return sqrt(lambda*sqr(qh) +   sqr(field(two_Duh(0,0)))
         + sqr(field(two_Duh(1,1))) + 2*sqr(field(two_Duh(0,1))));
}
```

```
field solve(const geo& omega, const string& approx, Float lambda) {
  space Vh = embankment_space(omega, approx);
  field uh (Vh, 0.0), fh (Vh, 0.0);
  fh[1] = -1.0;
  form m  (Vh, Vh, "mass");
  form a1 (Vh, Vh, "div_div");
  form a2 (Vh, Vh, "2D_D");
  form a = lambda*a1 + a2;
  ssk<Float> fact = ldlt(a.uu);
  uh.u = fact.solve (m.uu*fh.u + m.ub*fh.b - a.ub*uh.b);
  return uh;
}
int main(int argc, char**argv) {
  const Float lambda = 1;
  geo  omega_h (argv[1]);
  string approx  = (argc > 2) ? argv[2] : "P1";
  size_t n_adapt = (argc > 3) ? atoi(argv[3]) : 5;
  adapt_option_type options;
  options.hcoef = 1.1;
  options.hmin  = 0.004;
  for (size_t i = 0; true; i++) {
      field uh = solve(omega_h, approx, lambda);
      orheostream o (omega_h.name(), "mfield");
      o << catchmark("lambda")  << lambda << endl
        << catchmark("u")       << uh;
      if (i == n_adapt) break;
      field ch = criteria(lambda,uh);
      omega_h  = geo_adapt(ch, options);
      omega_h.save();
  }
}
```

Comments

The code works for both linear or quadratic approximations. Since there is not yet any available anisotropic adaptive mesh generator in three dimension, we have specialized here the code for the dimension two.

The criteria is here:

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

The `adapt_option_type` declaration is used by `rheolef` to send options to the mesh generator. The `hcoef` parameter controls 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

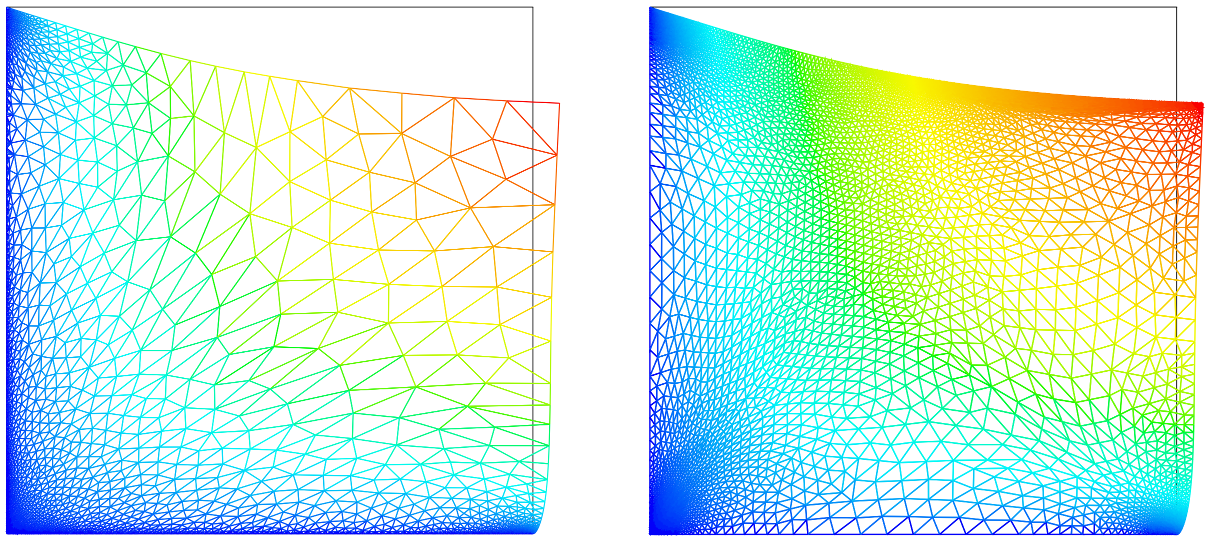


Figure 3.5: Adapted meshes: the deformation visualization for P_1 and P_2 approximations.

The compilation command writes:

```
make embankment-adapt-2d
```

For a piecewise linear approximation:

```
mkgeo_grid -t 10 > square-P1.geo
./embankment-adapt-2d square-P1  P1 5
```

The code performs a loop of five mesh adaptations: the corresponding meshes are stored in files, from `square-P1-1.geo.gz` to `square-P1-5.geo.gz`, and the associated solutions in files, from `square-P1-1.mfield.gz` to `square-P1-5.mfield.gz`. The additional '`.gz`' suffix expresses that the files are compressed using `gzip`.

```
geo square-P1-5.geo
mfield -u -deformation square-P1-5.mfield
```

Note that the '`.gz`' suffix is assumed by the `geo` and the `mfield` commands. Enter the $\boxed{\text{w}}$ key in the graphic window: you will obtain the graphic shown on Fig.3.5.
For a piecewise quadratic approximation:

```
mkgeo_grid -t 10 > square-P2.geo
./embankment-adapt-2d square-P2  P2
```

Then, the visualization writes:

```
geo square-P2-5.geo
mfield -u -deformation square-P2-5.mfield
```

3.4 The Stokes problem

Formulation

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

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

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

where $D(\mathbf{u}) = (\boldsymbol{\nabla}\mathbf{u} + \boldsymbol{\nabla}\mathbf{u}^T)/2$. The boundaries are represented on Fig. 3.1, page 29.
The variational formulation of this problem expresses:

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

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

where

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

Approximation

The Talor-Hood [12] finite element approximation of the Stokes problem is considered. We introduce a mesh $\mathcal{T}_h$ of Ω and the following finite dimensional spaces:

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

The approximate problem writes:

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

$$\begin{array}{lcccl}
a(\mathbf{u}_h,\mathbf{v}) & + & b(\mathbf{v},p_h) & = & 0, \ \forall \mathbf{v} \in \mathbf{V}_h(0), \\
b(\mathbf{u}_h,q) & & & = & 0, \ \forall q \in Q_h.
\end{array} \tag{3.1}$$

File ‘cavity.h’

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

File ‘stokes-cavity.cc’

```
#include "rheolef.h"
#include "rheolef/mixed_solver.h"
using namespace rheolef;
using namespace std;
#include "cavity.h"
#include "pcg_solver.h"
int main(int argc, char**argv) {
  geo  omega (argv[1]);
  space Vh = cavity_space (omega, "P2");
  space Qh (omega, "P1");
  field uh = cavity_field (Vh, 1);
  field ph (Qh, 0.);
  form mp (Qh, Qh, "mass");
  form a  (Vh, Vh, "2D_D");
  form b (Vh, Qh, "div"); b = -b;
  int   max_iter = 5000;
  Float tol      = 1e-15;
  if (omega.dimension() < 3) {
    int status = pcg_abtb (a.uu, b.uu, uh.u, ph.u, -(a.ub*uh.b), -(b.ub*uh.b),
      ldlt(mp.uu), ldlt(a.uu), max_iter, tol, &cerr);
  } else {
    int status = pcg_abtb (a.uu, b.uu, uh.u, ph.u, -(a.ub*uh.b), -(b.ub*uh.b),
      pcg_solver(mp.uu), pcg_solver(a.uu), max_iter, tol, &cerr);
  }
  cout << catchmark("u")  << uh
       << catchmark("p")  << ph;
  return 0;
}
```

Comments

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

The problem admits the following matrix form:

$$\left(\begin{array}{cc} \mathtt{a.uu} & \mathtt{trans(b.uu)} \\ \mathtt{b.uu} & 0 \end{array}\right)\left(\begin{array}{c} \mathtt{uh.u} \\ \mathtt{ph.u} \end{array}\right) = \left(\begin{array}{c} -\mathtt{a.ub} * \mathtt{uh.b} \\ -\mathtt{b.ub} * \mathtt{uh.b} \end{array}\right)$$

An initial value for the pressure field is provided:

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

This system is solved by the preconditioned conjugate gradient algorithm :

```
int status = pcg_abtb (a.uu, b.uu, uh.u, ph.u, -(a.ub*uh.b), -(b.ub*uh.b),
  ldlt(mp.uu), ldlt(a.uu), max_iter, tol, &cerr);
```

The preconditioner is here the mass matrix `mp.uu` for the pressure: as showed in [13], 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 mixed_solver
```

When $d = 2$, it is interessant to factorize both `a.uu` and the preconditionner `mp.uu` one time for all the iterations: the arguments `ldlt(mp.uu)` and `ldlt(a.uu)` are passed to the `pcg_abtb` algorithm. When $d = 3$, it is preferable, for both computing time and memory occupation point of view, to switch to an interative solver for subproblems related to `a.uu` and `mp.uu`:

```
int status = pcg_abtb (a.uu, b.uu, uh.u, ph.u, -(a.ub*uh.b), -(b.ub*uh.b),
  pcg_solver(mp.uu), pcg_solver(a.uu), max_iter, tol, &cerr);
```

Finaly, the `pcg_solver` class is a convenient wrapper class for the call to the `pcg` algorithm, used for 3D problems. It uses the incomplete Choeski factorization preconditionner `ic0`.

File '`pcg_solver.h`'

```
struct pcg_solver {
  pcg_solver (const csr<Float>& a1) : a(a1), m(ic0(a1)) {}
  vec<Float> solve (const vec<Float>& b) const {
    size_t max_iter = 10000;
    Float tol = 1e-15;
    vec<Float> x (b.size(), 0.0);
    int status = pcg (a, x, b, m, max_iter, tol);
    return x;
  }
  csr<Float> a;
  basic_ic0<Float> m;
};
```

How to run the program

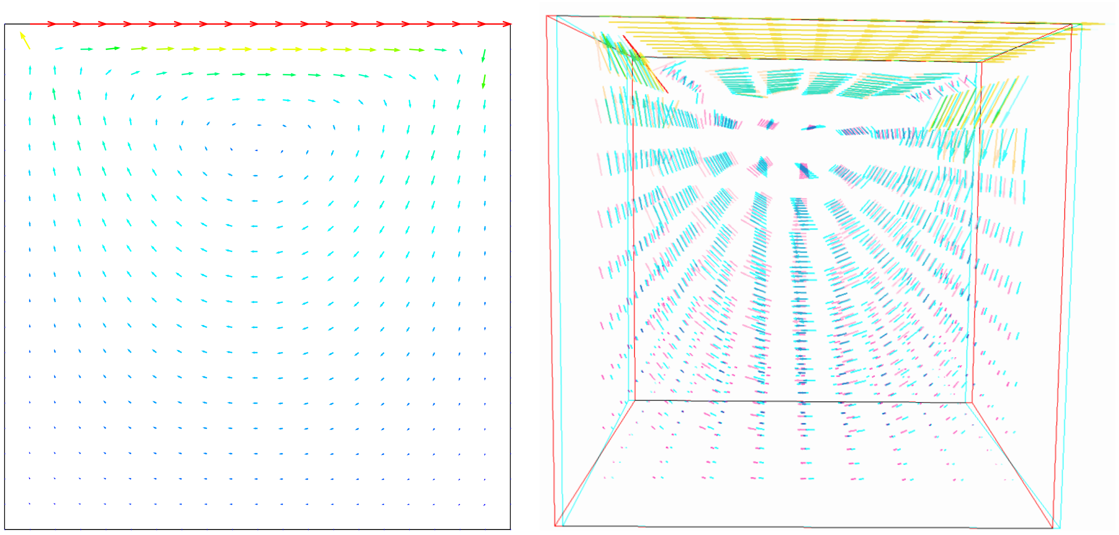


Figure 3.6: The velocity visualization for $N = 2$ and $N = 3$ with stereo anaglyph.

We assume that the previous code is contained in the file '`stokes-cavity.cc`'. Then, compile the program as usual (see page 10):

```
make stokes-cavity
```

and enter the comands:

```
mkgeo_grid -t 10 > square.geo
./stokes-cavity square > cavity2d.mfield
```

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

```
mfield cavity2d.mfield -u -velocity
```

Run also some scalar visualizations:

```
mfield cavity2d.mfield -u0 | field -
mfield cavity2d.mfield -u1 | field -
mfield cavity2d.mfield -p  | field -
```

Next, perform another computation on a finer mesh:

```
mkgeo_grid -t 20 > square-fine.geo
./stokes-cavity square-fine.geo > cavity2d-fine.mfield
```

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

```
      mkgeo_grid -T 5 > cube.geo
      ./stokes-cavity cube.geo > cavity3d.mfield
```

and the corresponding visualization:

```
      mfield cavity3d.mfield -u -velocity -stereo
      mfield cavity3d.mfield -u0 | field -
      mfield cavity3d.mfield -u1 | field -
      mfield cavity3d.mfield -u2 | field -
      mfield cavity3d.mfield -p  | field -
```

3.5 Computing the vorticity

Formulation and approximation

When $N = 2$, we define [3, page 30] for any distributions ϕ and $\mathbf{v}$:

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

and when $N = 3$:

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

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

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

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

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

The approximate variational formulation writes:

$$\begin{aligned}\omega_h \in Y_h,\ \int_\Omega \omega_h\, \xi\, dx &= \int_\Omega \mathrm{curl}\,\mathbf{u}_h\, \xi\, dx,\ \forall \xi \in Y_h \quad \text{when } N = 2, \\ \boldsymbol{\omega} \in \mathbf{Y}_h,\ \int_\Omega \boldsymbol{\omega}_h.\boldsymbol{\xi}\, dx &= \int_\Omega \mathbf{curl}\,\mathbf{u}_h.\boldsymbol{\xi}\, dx,\ \forall \boldsymbol{\xi} \in \mathbf{Y}_h \quad \text{when } N = 3.\end{aligned}$$

File ‘vorticity.cc’

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

int main(int argc, char** argv) {
  field uh;
  cin >> uh;
  string option =  (uh.n_component() == 3) ? "vector" : "scalar";
  space Lh  (uh.get_geo(), "P1d", option);
  form curl  (uh.get_space(), Lh, "curl");
  form inv_m (Lh, Lh, "inv_mass");
  cout << catchmark("w") << inv_m*(curl*uh);
  return 0;
}
```

How to run the program

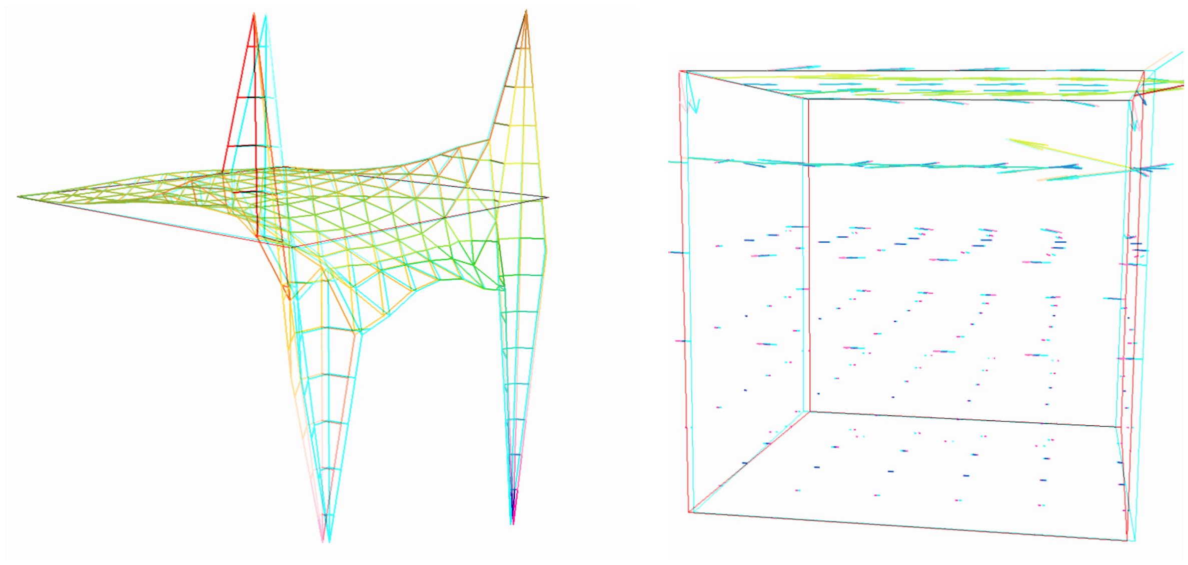


Figure 3.7: The vorticity: elevation view for $N = 2$ and vector representation for $N = 3$ (with anaglyph).

For $N = 2$, just enter:

```
make vorticity
./vorticity < cavity2d.mfield | field -evelavtion -stereo -
```

and you observe a discontinuous piecewise linear representation of the approximate vorticity. The approximate vorticity field can also be projected on a continuous piecewise linear space, using the `-proj` option (See Fig. 3.7 left):

```
./vorticity < cavity2d.mfield | field -proj -
./vorticity < cavity2d.mfield | \
          field -proj -elevation -scale 2 -stereo -nofill -
```

For $N = 3$, the whole vorticity vector can also be visualized (See Fig. 3.7 right):

```
./vorticity < cavity3d.mfield | \
          mfield -proj -w -velocity -vscale 2 -stereo -
```

In the previous command, the `-proj` option has been used: since the 3D render has no support for discontinuous picewise 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, roughtly similar to the bidimensionnal case.

```
./vorticity < cavity3d.mfield | mfield -w1 - | field -
./vorticity < cavity3d.mfield | mfield -w1 - | field -proj -
```

3.6 Computing the stream function

Formulation and approximation

The stream function satisfies $\mathbf{curl}\,\psi = \mathbf{u}$. When $N = 2$, the stream function ψ is the solution of the following problem [3, page 88]:

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

and when $N = 3$, the stream function is a vector-valued field $\boldsymbol{\psi}$ that satisfies [3, page 90]:

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

File ‘`streamf-cavity.cc`’

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

int main (int argc, char** argv) {
  field wh;
  cin >> wh;
  string option =  (wh.n_component() == 3) ? "vector" : "scalar";
  space Ph (wh.get_geo(), "P2", option);
  Ph.block("top");  Ph.block("bottom");
  if (wh.dimension() == 3) {
    Ph.block("back"); Ph.block("front");
  } else {
    Ph.block("left"); Ph.block("right");
  }
  form m (wh.get_space(), Ph, "mass");
  form c (Ph,  Ph, "grad_grad");
  field psih (Ph, 0.);
  ssk<Float> fact_c = ldlt(c.uu);
  psih.u = fact_c.solve (m.uu*wh.u + m.ub*wh.b - c.ub*psih.b);
  cout << catchmark("psi") << psih;
  return 0;
}
```

How to run the program

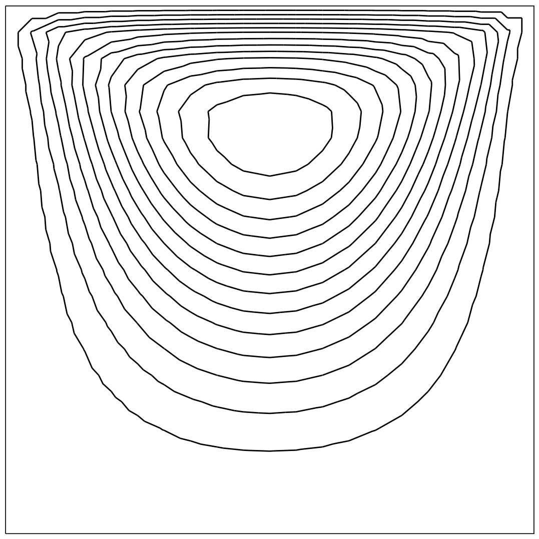
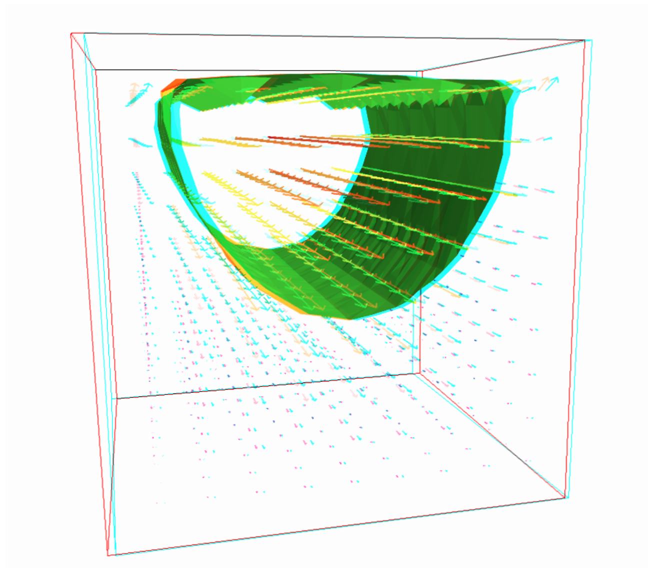



Figure 3.8: The stream function visualization: isolines for $N = 2$, and combined vectors and isonorm surface for $N = 3$.

For $N = 2$, just enter (see Fig. 3.8 left):

```
make streamf-cavity
./vorticity < cavity2d.mfield | ./streamf-cavity | field -black-and-white -
```

For $N = 3$, the second component of the stream function is showed by:

```
./vorticity < cavity3d.mfield | ./streamf-cavity | mfield -psi1 - | \
          field -normal 0 1 0 -
```

The whole stream function vector can be visualized:

```
./vorticity < cavity3d.mfield | ./streamf-cavity | \
          mfield -psi -velocity
```

The combined representation of Fig. 3.8.right has been obtained in three steps. First, enter:

```
./vorticity < cavity3d.mfield | ./streamf-cavity | mfield -psi1 - | \
          field -normal 0 1 0 -noclean -noexecute -name psi1 -
```

This command creates a file '`psi1.vtk`'. Next, in the previous `mayavi` window associated to the whole stream function, select the `File/Load data/VTK file` menu and load '`psi1.vtk`'. Finaly, select the `Vizualize/Module/IsoSurface` menu. Observe that the 3D stream function is mainly represented by its second component.

Chapter 4

Nearly incompressible elasticity and the stabilized Stokes problems

4.1 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

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

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 [12] 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:*

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

File ‘incompresible-elasticity.cc’

```
#include "rheolef.h"
#include "rheolef/mixed_solver.h"
using namespace rheolef;
using namespace std;
#include "embankment.h"
#include "pcg_solver.h"
int main(int argc, char**argv) {
  geo omega (argv[1]);
  Float inv_lambda = (argc > 2 ? atof(argv[2]) : 0);
  space Vh = embankment_space(omega, "P2");
  space Qh (omega, "P1");
  field uh (Vh,0.0), fh (Vh, 0.0);
  fh [omega.dimension()-1] = -1.0;
  field ph (Qh, 0.);
  form mu (Vh, Vh, "mass");
  form mp (Qh, Qh, "mass");
  form a (Vh, Vh, "2D_D");
  form b (Vh, Qh, "div"); b = -b;
  form c = inv_lambda*mp;
  int   max_iter  = 5000;
  Float tol       = 1e-15;
  if (omega.dimension() < 3) {
    int status = pcg_abtbc (a.uu, b.uu, c.uu, uh.u, ph.u, -(a.ub*uh.b) + (mu*fh).u,
        -(b.ub*uh.b), ldlt(mp.uu), ldlt(a.uu), max_iter, tol, &cerr);
  } else {
    int status = pcg_abtbc (a.uu, b.uu, c.uu, uh.u, ph.u, -(a.ub*uh.b) + (mu*fh).u,
        -(b.ub*uh.b), pcg_solver(mp.uu), pcg_solver(a.uu), max_iter, tol, &cerr);
  }
  cout << setprecision(numeric_limits<Float>::digits10)
       << catchmark("inv_lambda")  << inv_lambda << endl
       << catchmark("u")  << uh
       << catchmark("p")  << ph;
  return 0;
}
```

Comments

The problem admits the following matrix form:

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

The problem is similar to the Stokes one (see page 42): this system is solved by the preconditioned conjugate gradient algorithm :

```
    int status = pcg_abtbc (a.uu, b.uu, c.uu, uh.u, ph.u, -(a.ub*uh.b) + (mu*fh).u,
        -(b.ub*uh.b), ldlt(mp.uu), ldlt(a.uu), max_iter, tol, &cerr);
```

The preconditioner is here the mass matrix `mp.uu` for the pressure. As showed in [13], 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

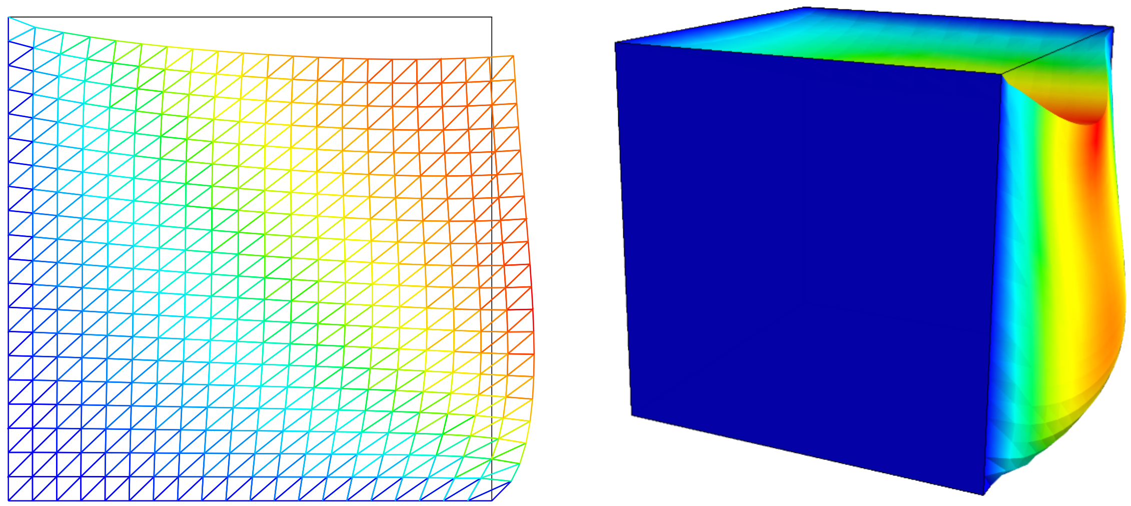


Figure 4.1: The incompressible linear elasticity ($\lambda = +\infty$) for $N = 2$ and $N = 3$.

We assume that the previous code is contained in the file '`incompressible-elasticity.cc`'. Compile the program as usual (see page 10):

```
      make incompressible-elasticity
```

and enter the comands:

```
      mkgeo_grid -t 10 > square.geo
      ./incompressible-elasticity square.geo 0 > square.mfield
      mfield -u -deformation square.mfield

      mkgeo_grid -T 10 > cube.geo
```

```
./incompressible-elasticity cube.geo 0 > cube.mfield
mfield -u -deformation cube.mfield -fill
```

The visualization is performed as usual: see section 3.1, page 32. Compare the results on FIG. 4.1, obtained for $\lambda = +\infty$ with those of FIG. 3.2, page 32, obtained for $\lambda = 1$.

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

4.2 The $P_1 - bubble$ element for the Stokes problem

Formulation and approximation

Let us go back to the Stokes problem. In section 3.4, page 40, the Taylor-Hood finite element was considered. Here, we turn to the mini-element [14] proposed by Arnold, Brezzi and Fortin, also well-known as the *P1-bubble* element. This element is generaly 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 technics and will presents some new `rheolef` features.

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

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

where $B(K) = \text{vect}(\lambda_1 \times \ldots \times \lambda_{N+1})$ 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 (3.1), page 40, 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.

File '`contraction-bubble.cc`'

```
#include "rheolef.h"
#include "rheolef/mixed_solver.h"
using namespace rheolef;
using namespace std;
Float u_upstream (const point& x) { return sqr(8.-x[0])/512.; }
int main(int argc, char**argv) {
  geo  omega (argv[1]);
  domain upstream = omega["upstream"];
  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[0].block ("axis");
  X1h[0].block ("downstream");
  X1h[0].block ("axis");
  space Xh = X1h * Bh;
  field uh (Xh);
  field ph (Qh, 0.);
  uh[0]["wall"] = 0;
  uh[1]["wall"] = 0;
  uh[0]["upstream"] = 0;
  uh[1]["upstream"] = interpolate(Wh,u_upstream);
  uh[0]["downstream"] = 0;
  uh[0]["axis"] = 0;
  form a1 (X1h, X1h, "2D_D");
  form ab (Bh,  Bh,  "2D_D");
  form a0 = form_nul (Bh, X1h);
  form_manip a_manip;
  a_manip << size(2,2)
          << a1        << a0
          << trans(a0) << ab;
  form a (Xh, Xh);
  a_manip >> a;
  ssk<Float> fact_a = ldlt(a.uu);
  form b1 = - form(X1h, Qh, "div");
  form bb = - form(Bh,  Qh, "div");
  form_manip b_manip;
  b_manip << size(1,2)
          << b1 << bb;
  form b (Xh, Qh);
  b_manip >> b;
  form mp (Qh, Qh, "mass");
  ssk<Float> fact_mp = ldlt(mp.uu);
  int   max_iter  = 500;
  Float tol       = 1e-14;
  int status = pcg_abtb (a.uu, b.uu, uh.u, ph.u, -(a.ub*uh.b), -(b.ub*uh.b),
        fact_mp, fact_a, max_iter, tol, &cerr);
  cout << catchmark("u")  << uh
       << catchmark("p")  << ph;
  return status;
}
```

Comments

The boundary conditions are suitable for a flow in an abrupt contraction geometry. The matrix structure is similar to those of the Taylor-Hood element, and thus the same efficient augmented Lagrangian algorithms applies.

The global form $a(.,.)$ over $\mathbf{X}_h$ is obtained by concatenation of the forms $a_1(.,.)$ and $a_b(.,.)$ over $\mathbf{X}_h^{(1)}$ and $\mathbf{B}_h$ repectively, thanks to the `form_manip` class:

$$\texttt{a} = \begin{pmatrix} \texttt{a1} & 0 \\ 0 & \texttt{ab} \end{pmatrix}$$

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 files '`contraction.bamgcad`' and '`contraction.dmn`' provided with the the `rheolef` distribution. The directory where examples are available is given by the following unix command:

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

The building mesh procedure is presented in appendix A, page A. Running this example is left as an exercice to the reader.

4.3 The stabilized Stokes problem

An alternative and popular implementation of this element eliminates the unknowns related to the bubble components (see e.g. [15], page 24). This elimination can be easily performed since the form $a_b(.,.)$ over $\mathbf{B}_h$ is diagonal, due to the fact that the bubble functions vanishes on the boundary of elements. The system reduces to:

$$\begin{pmatrix} A1 & B1^T \\ B1 & -C \end{pmatrix} \begin{pmatrix} U1 \\ P \end{pmatrix} = \begin{pmatrix} F \\ G \end{pmatrix}$$

Remarks that the matrix structure is similar to those of the nealy incompressible elasticity (see 4.1, page 4.1). A direct finite element formulation for this problem is known as the $P_1 - P_1$ stabilized element, proposed by Brezzi and Pitkäranta [16].

4.4 Axisymetric geometries

The coordinate system is associated to the geometry description, stored together with the mesh in the '`.geo`':

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

Note the additional line in the header:

```
        ...
        coordinate_system rz
        ...
```

Here `"rz"` means that the coordinate system $(x_0, x_1) = (r, z)$. Notes that the coordinate system (z, r) is also supported but not fully tested yet: it may also be implemented by a suitable coordinate swap. The `"cartesian"` argument string is also supported and means that the usual coordinate system may be used. In the `"rz"` case, the L^2 functional space is equipped with the following weighted scalar product

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

and all usual bilinear forms are now implemented by using this weight.

By this way, a program source code can handle both cartesian and axisymetric systems: only the input geometry makes the difference.

```
    myprog square.geo
    myprog square-rz.geo
```

Thus, the coordinate system can be chosen at run time and we can expect an efficient source code reduction.

Two complete examples `stokes-poiseuille.cc` and `stokes-poiseuille-bubble.cc` are provided in the `example` directory. These examples are related to the Poiseuille flow of a fluid either between parallel planes (cartesian) or in a pipe (axisymetric). The first example implements the Taylor-Hood approximation while the second one uses the bubble-stabilized P1-P1 approximation.

Chapter 5

Time-dependent problems

5.1 The heat equation

Formulation

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

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

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

Approximation

Let $\Delta t > 0$ and $t_n = n\Delta t$, $n \geq 0$. 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 $u^{(0)} = 0$. We reuse the bilinear forms a and m defined in section 1.1, page 7 for the Dirichlet problem and introduce the bilinear form $c = m + \Delta t\, a$. The variationnal formulation of the time-discretized problem writes:
$(VF)_n$: Let u^n being known, find $u^{n+1} \in H^1_0(\Omega)$ such that

$$c\,(u^{n+1}, v) = m(u^n + \Delta t\, f(t_{n+1}), v), \ \forall v \in H^1_0(\Omega).$$

This is a Poisson-like problem. The discretization with respect to space of this problem is similar to those presened in section 1.1, page 7.

File ‘heat.cc’

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
int main (int argc, char **argv) {
  geo omega (argv[1]);
  size_t n_max = (argc > 2) ? atoi(argv[2]) : 10;
  Float delta_t = 0.5/n_max;
  space Vh (omega, "P1");
  Vh.block ("boundary");
  form m (Vh, Vh, "mass");
  form a (Vh, Vh, "grad_grad");
  form c = m + delta_t*a;
  ssk<Float> c_fact = ldlt (c.uu);
  field fh (Vh, 1.);
  field uh (Vh, 0.);
  branch event ("t","u");
  cout << event (0, uh);
  for (size_t n = 1; n <= n_max; n++) {
    field mb = m*(uh + delta_t*fh);
    uh.u = c_fact.solve (mb.u - c.ub*uh.b);
    cout << event (Float(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 $(t_n, u^n)_{n\geq 0}$, on the standard output in the '`.branch`' file format. An instruction as:

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

is equivalent to the formated output

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

How to run the program

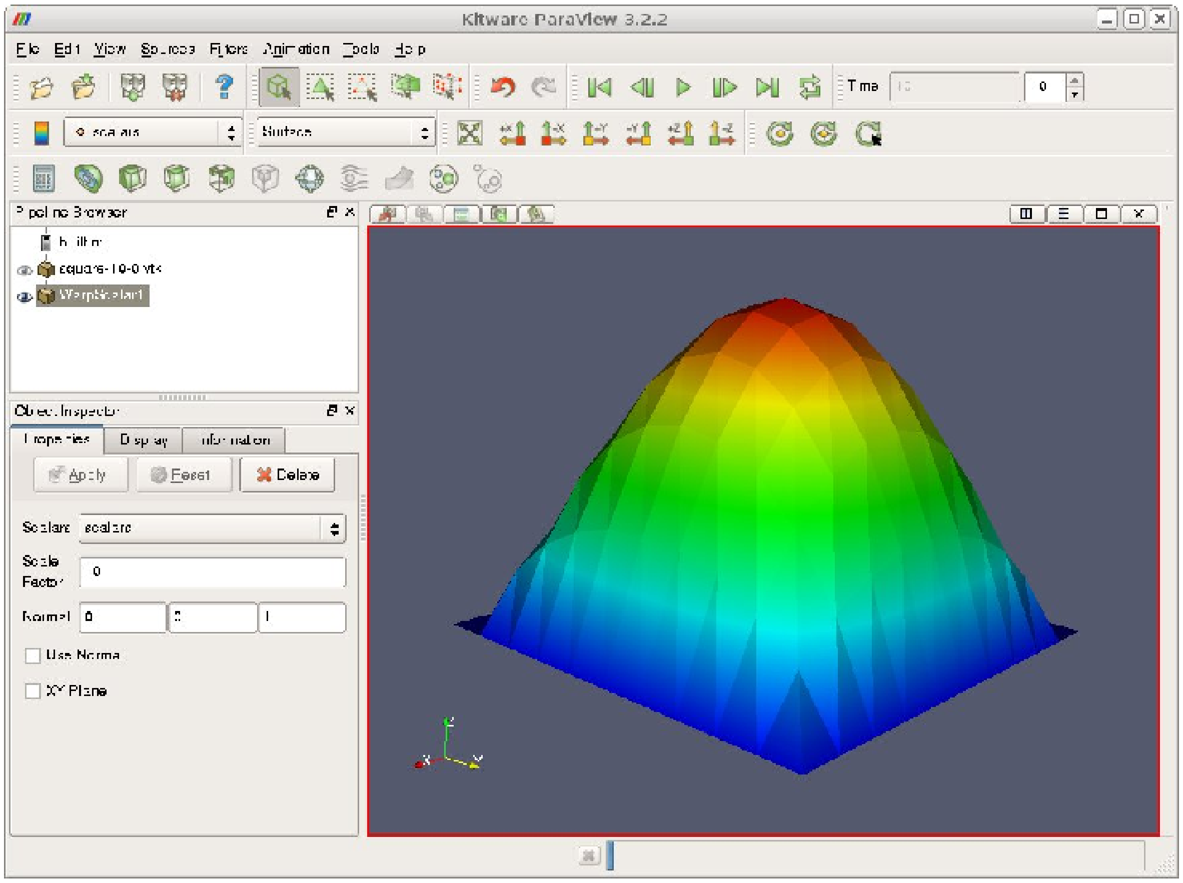


Figure 5.1: Animation of the solution of the heat problem.

We assume that the previous code is contained in the file '`heat.cc`'. Then, compile the program as usual (see page 10):

```
make heat
```

For a one dimensional problem, enter the comands:

```
mkgeo_grid -e 10 -boundary > line-10.geo
./heat line-10.geo > line-10.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 -boundary > square-10.geo
./heat square-10.geo > square-10.branch
```

For a tridimensional one:

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

How to run the animation

```
branch line-10.branch   -gnuplot
```

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-10.branch -paraview
paraview &
```

A window appears, that looks like a video player. Then, open the `File->open` menu and load `square-10.vtk`. Then, press the $\boxed{\text{apply}}$ green button and, in the `object inspector window`, select `display` and clic on the $\boxed{\text{rescale to data range}}$ button. Then clic on the video $\boxed{\text{play}}$ 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, clic on the graphic window, rotate the view and finally play the animation
To generate an animation file[1], go to the `File->save animation` menu and enter as file name `square-10` and as file type `jpeg`. A collection of `jpeg` files are generated by `paraview`. Then, run the unix command:

```
ffmpeg -r 2 -i 'square-10.%04d.jpg' square-10.mov
```

The animation file `square-10.mov` can now be started from any video player, such as `vlc`:

```
vlc --loop square-10.mov
```

For the tridimensional case, the animation feature is similar.

5.2 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*

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

where $\mathbf{u}$, ϕ_0 and ϕ_Γ being known.

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

Time approximation

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

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

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

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

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

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. [17]), and is known as the method of characteristic. The efficient localization of $X^n(x)$ in an unstructured mesh involves a quadtree data structure[2].

The following code implements the classical rotating Gaussian hill (see e.g. [18]).

File ‘convect.cc’

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
Float nu = 1e-3;
point u (const point & x) { return point(-x[1], x[0]); }
struct phi : unary_function<point,Float> {
  Float operator() (const point& x) {
    point xc (0.25, 0, 0);
    Float sigma = 0.01;
    return sigma/(sigma + 4*nu*t)
        * exp(-(sqr( x[0]*cos(t) + x[1]*sin(t) - xc[0])
              + sqr(-x[0]*sin(t) + x[1]*cos(t) - xc[1]))/(sigma + 4*nu*t));
  }
  phi (Float tau) : t(tau) {}
  protected: Float t;
};
int main (int argc, char **argv) {
  geo omega (argv[1]);
  size_t n_max  = (argc > 2) ? atoi(argv[2]) : 24;
  Float delta_t = 2*acos(-1.)/n_max;
  space Vh (omega, "P1");
  Vh.block ("boundary");
  form m (Vh, Vh, "mass");
  form a (Vh, Vh, "grad_grad");
  form c = m + delta_t*nu*a;
  ssk<Float> c_fact = ldlt (c.uu);
  field uh = interpolate (Vh*Vh, u);
```

[2]The quadtree is only implemented for two-dimensional problems in `rheolef`. It is planed to be extended to tridimensional problems.

```
  field phih = interpolate (Vh, phi(0));
  cerr << "# # t\terror" << endl;
  geomap X (Vh, -delta_t*uh);
  branch event ("t","phi");
  cout << event (0, phih);
  for (size_t n = 1; n <= n_max; n++) {
    Float t = Float(n)*delta_t;
    field prec_phih = compose (phih, X);
    phih.u = c_fact.solve (m.uu*prec_phih.u + m.ub*prec_phih.b - c.ub*phih.b);
    field e = phih - interpolate (Vh, phi(t));
    cerr << "# " << t << "\t" << sqrt(m(e,e)) << endl;
    cout << event (t, phih);
  }
}
```

Comments

We take $\Omega =]-\frac{1}{2},\frac{1}{2}[^2$ and $T = \pi$. This problem provides an example for a convection-diffusion equation with a variable velocity field $\mathbf{u}(x_0, x_1) = (-x_1, x_0)$ and a known analytical solution:

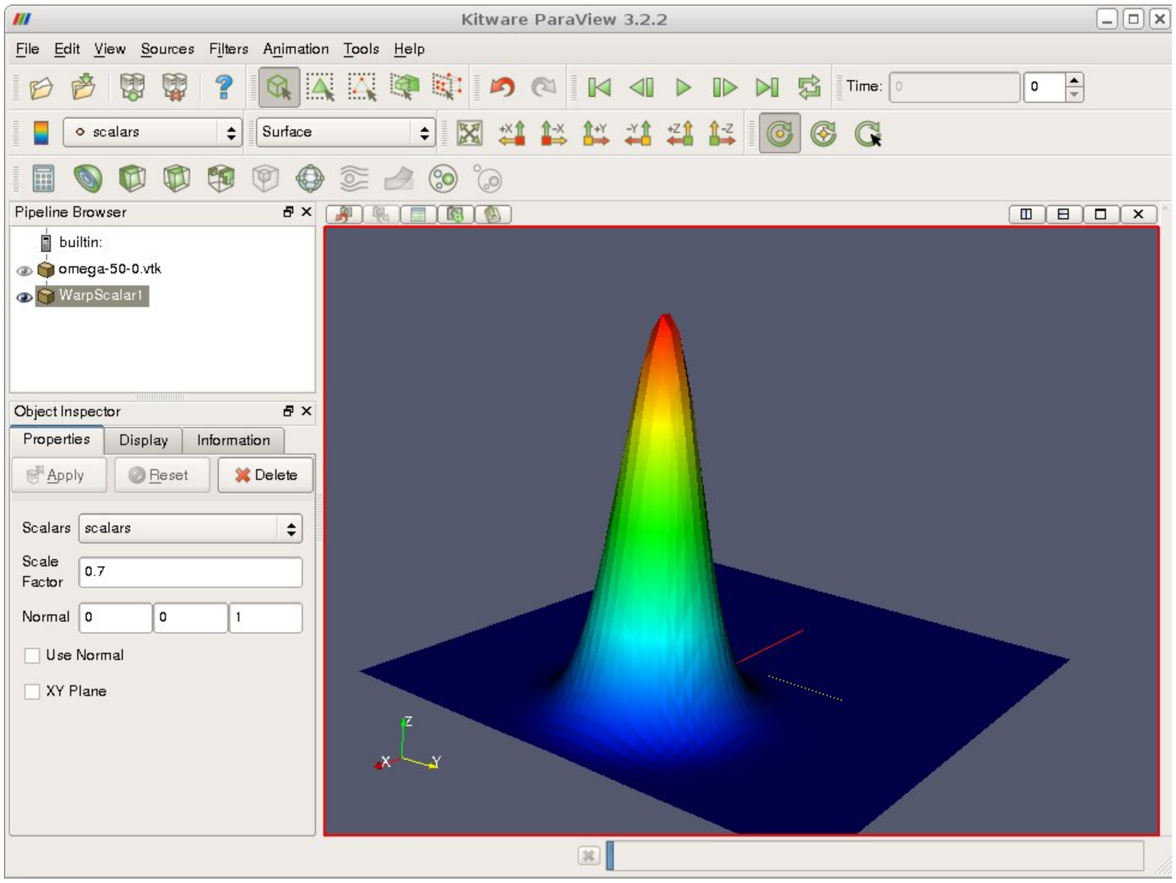
$$\phi(x,t) = \frac{\sigma}{\sigma + 4\nu t} \exp\left(-\frac{(x_0\cos(t) + x_1\sin(t) - x_{c,0})^2 + (-x_0\sin(t) + x_1\cos(t) - x_{c,1})^2}{\sigma + 4\nu t} \right)$$

where $\sigma > 0$ is the slope and x_c is the center of the hill.

Notice the use of a class-function implementation `phi` for the implementation of $\phi(t)$ as a function of x. Such programation style has been introduced in the Standard Template Library [19], 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.

The `geomap` variable `X` implements the localizer $X^n(x)$ and the `compose` function is used to perform the composition $\phi_h \circ X^n$.

How to run the program



builtin:
omega-50-0.vtk
WarpScalar1

Scalars scalars
Scale Factor 0.7
Normal 0 0 1
Use Normal
XY Plane

Figure 5.2: Animation of the solution of the rotating hill problem.

We assume that the previous code is contained in the file ‘`convect.cc`’. Then, compile the program as usual (see page 10):

```
make convect
```

and enter the comands:

```
mkgeo_grid -t 50 -boundary -a -0.5 -b 0.5 -c -0.5 -d 0.5 > omega-50.geo
./convect omega-50.geo > omega-50.branch
```

The visualization and animation are similar to those of the head problem previously presented in paragraph 5.1:

```
branch -paraview omega-50
paraview&
```

The result is shown on Fig. 5.2.

5.3 The Navier-Stokes problem

Formulation

This longer example combines most fonctionalities presented in the previous examples.

Let us consider the Navier-Stokes problem for the driven cavity in $\Omega =]0,1[^N$, $N = 2,3$. Let $Re > 0$ be the Reynolds number, and $T > 0$ a final time. The problem writes:

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

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

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 5.3, page 64. The boundaries are represented on Fig. 3.1, page 29.

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 Euler scheme:

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

where, following [20, 21]:

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

It is a second order extension of the method previously introduced in paragraph 5.2 page 61. The scheme defines a second order recurence for the sequence $(\mathbf{u}^n)_{n\geq -1}$, that starts with $\mathbf{u}^{-1} = \mathbf{u}^0 = 0$.

Variationnal formulation

The variationnal 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:*

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

where

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

where

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

and $b(.,.)$ and $\mathbf{V}(\alpha)$ was already introduced in paragraph 3.4, page 40, while studying the Stokes problem.

Space approximation

The Taylor-Hood [12] finite element approximation of this generalised Stokes problem was also considered in paragraph 3.4, page 40. 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:*

$$\begin{array}{lclcll} a(\mathbf{u}_h^{n+1},\mathbf{v}) & + & b(\mathbf{v},p_h^{n+1}) & = & m(\mathbf{f}_h^n,\mathbf{v}), & \forall \mathbf{v} \in \mathbf{V}_h(0), \\ b(\mathbf{u}_h^{n+1},q) & & & = & 0, & \forall q \in Q_h. \end{array} \tag{5.1}$$

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.

File ‘navier-stokes-solve.h’

```
// second order scheme (experimental)
#include "rheolef/mixed_solver.h"
#include "cahouet-chabart.h"
using namespace std;
int navier_stokes_solve (Float Re, Float delta_t, field f0h, field& uh, field& ph,
    size_t& max_iter, Float& tol, std::ostream *p_cerr=0, std::string label = "navier-stokes") {
  const space& Vh = uh.get_space();
  const space& Qh = ph.get_space();
  form m (Vh, Vh, "mass");
  form a (Vh, Vh, "2D_D");
  a = a + 1.5*(Re/delta_t)*m;
  ssk<Float> fact_a = ldlt(a.uu);
  form b (Vh, Qh, "div"); b = -b;
  cahouet_chabart preconditioner (Qh, 1.5*(Re/delta_t));
  if (p_cerr != 0) *p_cerr << "[" << label << "] #n |du/dt|" << endl;
  field uh1 = uh;
  geomap_option_type opts;
  opts.n_track_step = 10;
  for (size_t n = 0; true; n++) {
    field uh2 = uh1;
    uh1  = uh;
    field uh_star = 2.0*uh1 - uh2;
    geomap X1 (Vh,     -delta_t*uh_star, opts);
    geomap X2 (Vh, -2.0*delta_t*uh_star, opts);
    field  fh = f0h + 0.5*(Re/delta_t)*(4.0*compose(uh1,X1) - compose(uh2,X2));
    size_t gs_max_iter = 500;
    Float  gs_tol      = 1e-15;
    int status = pcg_abtb (a.uu, b.uu, uh.u, ph.u, (m*fh).u - a.ub*uh.b, -(b.ub*uh.b),
      preconditioner, fact_a, gs_max_iter, gs_tol);
    if (status != 0) warning_macro ("solve generalized stokes: precision not reached: tol = " << tol
    field duh_dt = (3*uh - 4*uh1 + uh2)/(2*delta_t);
    Float residual = sqrt(m(duh_dt,duh_dt));
    if (p_cerr != 0) *p_cerr << "[" << 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 generalised Stokes problem and manages a right-hand side $\mathbf{f}_h$:

```
    field uh_star = 2.0*uh1 - uh2;
    geomap X1 (Vh,     -delta_t*uh_star);
    geomap X2 (Vh, -2.0*delta_t*uh_star);
    field  fh = f0h + 0.5*(Re/delta_t)*(4.0*compose(uh1,X1) - compose(uh2,X2));
```

This last computation is similar to those done in the '`convect.cc`' example. The generalized Stokes problem is solved by the conjugate gradient algorithm :

```
    int status = pcg_abtb (a.uu, b.uu, uh.u, ph.u, (m*fh).u - a.ub*uh.b, -(b.ub*uh.b),
      preconditioner, fact_a, gs_max_iter, gs_tol, &cerr);
```

The preconditioner is here the Cahouet and Chabart one [22]. As showed in [23], the number of iterations need by the conjugate gradient algorithm to reach a given precision is then independent of the mesh size. This preconditioner leads to the resolution of the follwoing subproblem:

$$(M_h^{-1} + \lambda A_h^{-1})q_h = r_h$$

where $\lambda = Re/\Delta t$, $r_h \in Q_h$ is given and M and A are respectively the the mass matrix and the discrete Poisson operator with Neumann boundary condition. The resolution of this subproblem hqs been previously developed in section 2.4, page 22.

File '`cahouet-chabart.h`'

```
#include "neumann-laplace-assembly.h"
struct cahouet_chabart {
  cahouet_chabart (const space& Qh, Float lambda_1)
    : fact_m(), fact_c(), lambda(lambda_1) {
    form m (Qh, Qh, "mass");
    fact_m = ldlt(m.uu);
    form a (Qh, Qh, "grad_grad");
    form_diag dm (Qh, "mass");
    field mh (dm);
    csr<Float> c = neumann_laplace_assembly (a.uu, mh.u);
    fact_c = ldlt(c);
  }
  vec<Float> solve (const vec<Float>& Mp) const {
    vec<Float> q1 = fact_m.solve(Mp);
    vec<Float> Mp_e (Mp.size()+1);
    for (size_t i = 0; i < Mp.size(); i++) Mp_e.at(i) = Mp.at(i);
    Mp_e.at(Mp.size()) = 0;
    vec<Float> q2_e = fact_c.solve(Mp_e);
    vec<Float> q2 (Mp.size());
    for (size_t i = 0; i < q2.size(); i++) q2.at(i) = q2_e.at(i);
    vec<Float> q = q1 + lambda*q2;
    return q;
  }
  ssk<Float> fact_m;
  ssk<Float> fact_c;
  Float lambda;
};
```

File ‘navier-stokes-cavity.cc’

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
#include "navier-stokes-solve.h"
#include "cavity.h"
field criteria (Float Re, const field& uh) {
  space Xh  (uh.get_geo(), "P1d");
  form inv_m  (Xh, Xh, "inv_mass");
  form mpt   (Xh, uh.get_space()[0], "mass");
  form p = inv_m * trans(mpt);
  field c1 = sqr(p*uh[0]) + sqr(p*uh[1]);
  space Th  (uh.get_geo(), "P1d", "tensor");
  form two_D (uh.get_space(), Th, "2D");
  form inv_mt (Th, Th, "inv_mass");
  field two_Duh = inv_mt*(two_D*uh);
  field c2 = sqr(field(two_Duh(0,0))) + sqr(field(two_Duh(1,1))) + 2*sqr(field(two_Duh(0,1)));
  return sqrt(Re*c1 + c2);
}
int main (int argc, char**argv) {
  if (argc < 2) {
    cerr << "usage: " << argv[0] << " <geo> <Re> <Delta t> <n_adapt> <hcoef> <hmin>" << endl;
    exit (1);
  }
  geo    omega_h (argv[1]);
  Float  Re       = (argc > 2) ? atof(argv[2]) : 100;
  Float  delta_t  = (argc > 3) ? atof(argv[3]) : 0.05;
  size_t n_adapt  = (argc > 4) ? atoi(argv[4]) : 3;
  adapt_option_type options;
  options.hcoef   = (argc > 5) ? atof(argv[5]) : 2;
  options.hmin    = (argc > 6) ? atof(argv[6]) : 0.004; // 0.004
  options.hmax    = 0.1;
  space Vh = cavity_space (omega_h, "P2");
  space Qh (omega_h, "P1");
  field uh = cavity_field (Vh, 1.0);
  field ph (Qh, 0.0);
  field fh (Vh, 0.0);
  for (size_t i = 0; true; i++) {
    size_t max_iter = 500;
    Float tol = 1e-10;
    navier_stokes_solve (Re, delta_t, fh, uh, ph, max_iter, tol, &cerr);
    orheostream o (omega_h.name(), "mfield");
    o << setprecision(numeric_limits<Float>::digits10)
      << catchmark("Re") << Re << endl
      << catchmark("delta_t") << delta_t << endl
      << catchmark("u")  << uh
      << catchmark("p")  << ph;
    if (i >= n_adapt) break;
    field ch = criteria(Re,uh);
    omega_h  = geo_adapt(ch, options);
    omega_h.save();
    Vh = cavity_space (omega_h, "P2");
    Qh = space (omega_h, "P1");
    uh = cavity_field (Vh, 1.0);
    ph = field (Qh, 0.0);
    fh = field (Vh, 0);
  }
}
```

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.h`' introduced page 40. This file contains two functions that defines boundary conditions associated to the cavity driven problem.

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

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

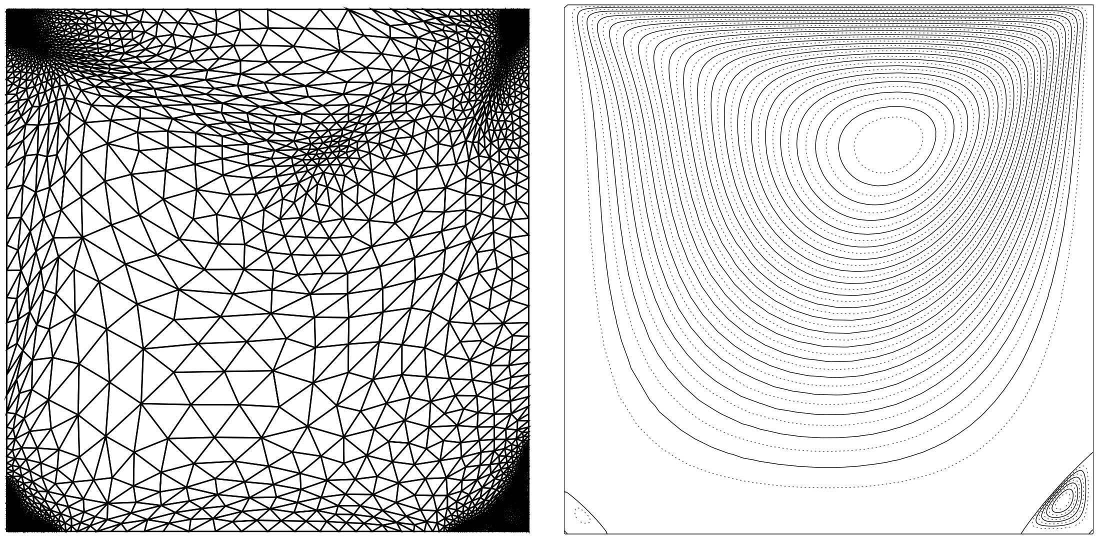
The `criteria` function is similar to those presented in the '`embankment-adapt-2d.cc`' example.

The `main` function enters in a loop, that solves at each iteration a generalised Stokes problem, then recompute the mesh and finally re-interpolates the solution $\mathbf{u}$ on the new mesh:

```
    uh = interpolate (Vh, uh);
```

How to run the program

$Re = 100$: 2481 vertices $\psi_{max} = 1.27 \times 10^{-4}, \psi_{min} = -0.10$



$Re = 400$: 3001 vertices $\psi_{max} = 6.4 \times 10^{-4}, \psi_{min} = -0.11$

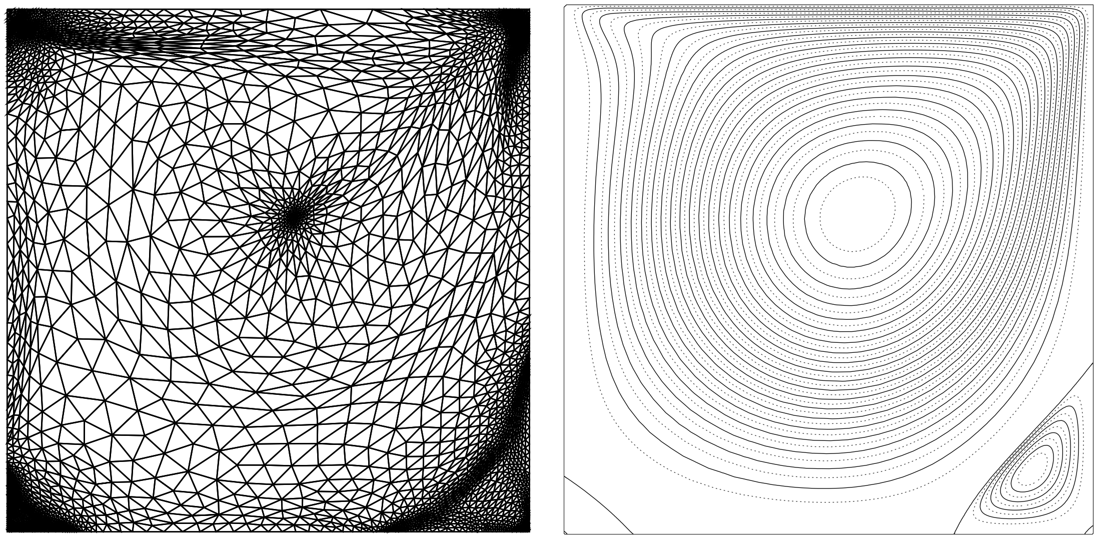


Figure 5.3: Meshes and stream functions associated to the solution of the Navier-Stokes equations for $Re = 100$ (top) and $Re = 400$ (bottom).

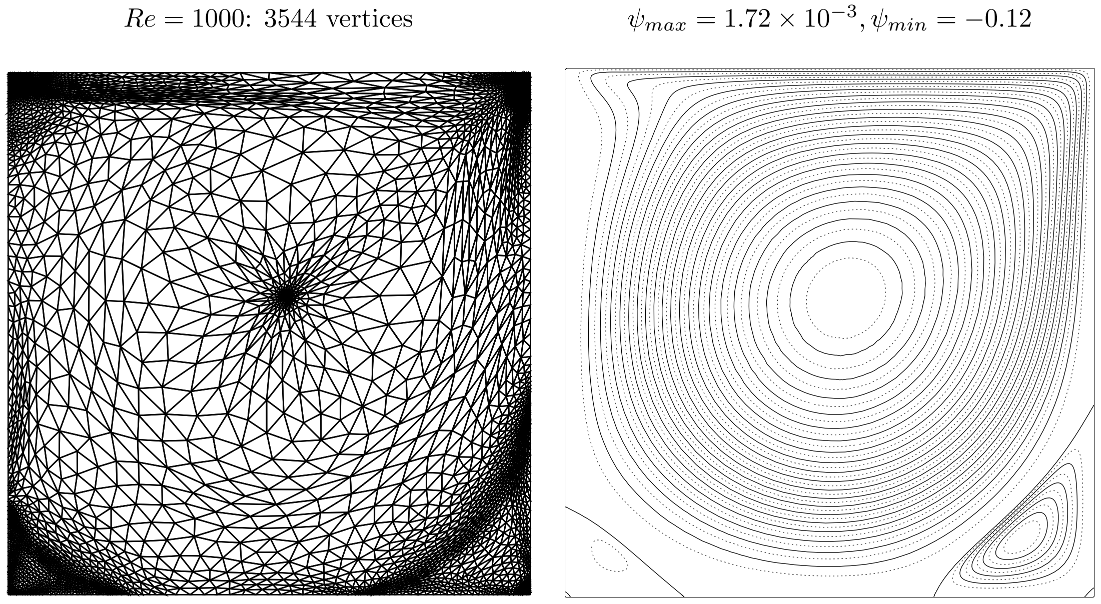

$Re = 1000$: 3544 vertices
$\psi_{max} = 1.72 \times 10^{-3}, \psi_{min} = -0.12$

Figure 5.4: Meshes and stream functions associated to the solution of the Navier-Stokes equations for $Re = 1000$.

First, creates an initial mesh:

```
    mkgeo_grid -t 10 > 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 three mesh adaptations. At each time step, the program prints an approximation of the time derivative, and stops when a stationary solution is reached. Then, extract and visualise the adapted mesh and the solution:

```
    geo square-100-3.geo
    mfield square-3.mfield.gz -u -velocity -vscale 10
```

The representation of the stream function writes:

```
    zcat square-3.mfield.gz | ./vorticity | ./streamf-cavity | \
           field -n-iso 15 n-iso-negative 10 -nofill -
```

The programs `vorticity` and `streamf-cavity` has been introduced pages 44 and 46, respectively. The last options of the `field` program draws isocontours of the stream function using lines, as shown on Fig. 5.3. 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
```

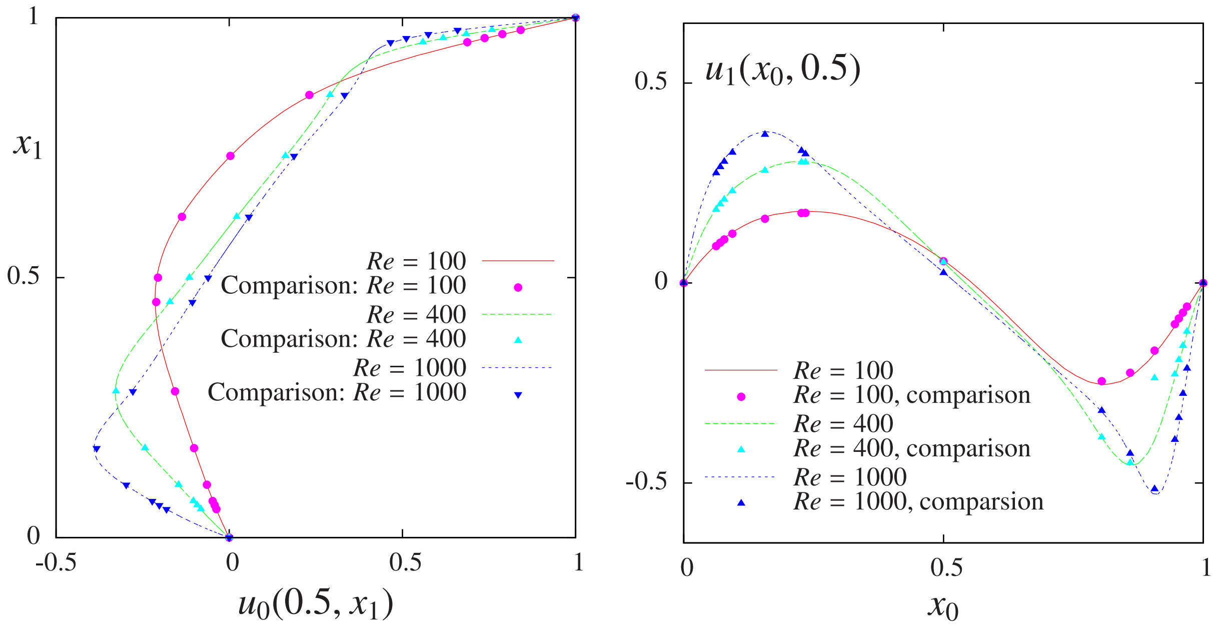


Figure 5.5: Navier-Stokes: velocity profiles along lines passing throught the center of the cavity, compared with data from [24]: (a) u_0 along the vertical line; (b) u_1 along the horizontal line line.

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

```
mfield square-3.mfield.gz -u0 | field -cut -normal -1 0 -origin 0.5 0 -gnuplot -
mfield square-3.mfield.gz -u1 | field -cut -normal  0 1 -origin 0 0.5 -gnuplot -
```

Fig. 5.5 compare the cuts with data from [24], table 1 and 2 (see also [25]). Observe that the solution is in good agreement with these previous computations.

Re		x_c	y_c	$-\psi_{\min}$
100	present	0.617	0.736	0.104
	Labeur and Wells [26]	0.608	0.737	0.104
	Donea and Huerta [27]	0.62	0.74	0.103
400	present	0.552	0.602	0.117
	Labeur and Wells [26]	0.557	0.611	0.115
	Donea and Huerta [27]	0.568	0.606	0.110
1000	present	0.532	0.569	0.121
	Labeur and Wells [26]	0.524	0.560	0.121
	Donea and Huerta [27]	0.540	0.573	0.110

Figure 5.6: Cavity flow: primary vortex position and stream function value.

Finaly, table 5.6 compares the primary vortex position and its associated stream function value. Notice also the good agreement with previous simulations. The small program that computes the primary vortex is showed below. Notice that degrees of freedom for the P_2 approximation of the stream function are listed with first the degrees of freedom associated to vertices and then thoses located at the middle of edges. When the extremum of the stream function is located on an edge, the program build first the edges, and then compute the middle-point associated to the edge degree of freedom.

File ‘vortex-position.cc’

```
#include "rheolef.h"
#include "rheolef/geo-connectivity.h"
using namespace rheolef;
using namespace std;
int main (int argc, char** argv) {
  field psi_h; cin >> psi_h; psi_h *= -1;
  Float psi_max = 0;
  size_t i_dof_max = 0;
  for (size_t i_dof = 0; i_dof < psi_h.size(); i_dof++) {
    if (psi_h.at(i_dof) <= psi_max) continue;
    psi_max = psi_h.at(i_dof);
    i_dof_max = i_dof;
  }
  point x_max;
  const geo& omega = psi_h.get_geo();
  if (i_dof_max < omega.n_vertex()) {
    x_max = omega.vertex(i_dof_max);
  } else {
    vector<pair<size_t,size_t> > edge;
    build_edge (omega, edge);
    size_t i_edge = i_dof_max - omega.n_vertex();
    point a = omega.vertex (edge[i_edge].first);
    point b = omega.vertex (edge[i_edge].second);
    x_max = (a+b)/2;
  }
  cout << "xc\t\tyc\t\tpsi" << endl
       << x_max[0] << "\t" << x_max[1] << "\t" << psi_max << endl;
}
```

For higher Reynolds number, Shen [28] showed in 1991 that the flow converges to a stationary state for Reynolds numbers up to 10000; 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 2002, Auteri et al. [29] estimated the critical value for the apparition of the first instability to $Re_1 \approx 8018$. In 2005, Erturk et al. [30] computed steady driven cavity solutions up to $Re \leq 21\,000$. Also in 2005, this result was infirmed by [31]: these authors estimated Re_1 close to 8000, in agreement with [29]. The 3D driven cavity has been investigated in [32] by the method of characteristic. 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 6

Multi-regions and non-constant coefficients problems

This chapter is related to the so-called transmission problem.

Formulation

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

$$\begin{aligned}
-\mathrm{div}(\eta \boldsymbol{\nabla} u) &= 1 \text{ in } \Omega \\
u &= 0 \text{ on } \Gamma_{\mathrm{left}} \cup \Gamma_{\mathrm{right}} \\
\frac{\partial u}{\partial n} &= 0 \text{ on } \Gamma_{\mathrm{top}} \cup \Gamma_{\mathrm{bottom}} \text{ when } N \geq 2 \\
\frac{\partial u}{\partial n} &= 0 \text{ on } \Gamma_{\mathrm{front}} \cup \Gamma_{\mathrm{back}} \text{ when } N = 3
\end{aligned}$$

We consider here very important special case when η is picewise constant. Let:

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

where $(\Omega_{\mathrm{west}}, \Omega_{east})$ is a partition of Ω. This problem is known as the **transmission** problem, since the solution and the flux are continuous on the interface:

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

where

$$\Gamma_0 = \partial\Omega_{\mathrm{west}} \cap \partial\Omega_{\mathrm{east}}$$

It expresses the transmission of the quantity u and its flux accross the interface Γ_0 between two regions that have different diffusion properties: We go back to the standard problem when $\varepsilon = 1$.
The variational formulation of this problem expresses:
(VF): find $u \in V$ such that:

$$a(u, v) = m(1, v), \ \forall v \in V$$

where the bilinear forms $a(.,.)$ and $m(.,.)$ are defined by

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

The bilinear form $a(.,.)$ defines a scalar product in V and is related to the *energy* form. This form is associated to the $-\text{div}\,\eta\boldsymbol{\nabla}$ operator. The $m(.,.)$ is here the classical scalar product on $L^2(\Omega)$, and is related to the *mass* form as usual.

The approximation of this problem could performed by a standard Lagrange P_k continuous approximation. We switch here to a mixed formulation that implements easily.

Mixed Formulation

Let us introduce the following vector-valued field:

$$\mathbf{p} = \eta\, \boldsymbol{\nabla} u$$

The problem can be rewritten as:

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

$$\begin{array}{rcrcll} \dfrac{1}{\eta}\mathbf{p} & - & \boldsymbol{\nabla} u & = & 0 & \text{in } \Omega \\ \text{div}\,\mathbf{p} & & & = & -1 & \text{in } \Omega \\ & & u & = & 0 & \text{on } \Gamma_{\text{left}} \cup \Gamma_{\text{right}} \\ & & \dfrac{\partial u}{\partial n} & = & 0 & \text{on } \Gamma_{\text{top}} \cup \Gamma_{\text{bottom}} \text{ when } N \geq 2 \\ & & \dfrac{\partial u}{\partial n} & = & 0 & \text{on } \Gamma_{\text{front}} \cup \Gamma_{\text{back}} \text{ when } N = 3 \end{array}$$

The variationnal formulation writes:

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

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

where

$$\begin{array}{rcl} \tilde{a}(\mathbf{p},\mathbf{q}) & = & \displaystyle\int_\Omega \frac{1}{\eta}\mathbf{p}.\mathbf{q}\, dx, \\ \tilde{b}(\mathbf{q},v) & = & -\displaystyle\int_\Omega \boldsymbol{\nabla} v.\mathbf{q}\, dx, \quad \forall u, v \in L^2(\Omega) \end{array}$$

This problem appears as more complex than the previous one, since there is two unknown instead of one. Nevertheless, the $\mathbf{p}$ unknown could be eliminated by inverting the operator associated to the $\tilde{a}$ form. This can be done easily since $\mathbf{p}$ is approximated by discontinuous piecewise polynomial functions.

Mixed approximation

Let $k \geq 1$ and

$$\begin{array}{rcl} \mathbf{Q}_h & = & \{\mathbf{q} \in (L^2(\Omega))^N;\ \mathbf{q}_{/K} \in (P_{k-1})^N\} \\ V_h & = & \{v \in V;\ v_{/K} \in P_k\} \end{array}$$

The mixed finite element approximation writes:
$(MVF)_h$: find $\mathbf{p}_h \in \mathbf{Q}_h$ and $u_h \in V_h$ such that:

$$\begin{array}{lclcll}
\tilde{a}(\mathbf{p}_h, \mathbf{q}) & + & \tilde{b}(\mathbf{q}, u_h) & = & 0, & \forall \mathbf{q} \in \mathbf{Q}_h \\
\tilde{b}(\mathbf{p}_h, v) & & & = & m(-1, v), & \forall v \in V_h
\end{array}$$

Here the operator associated to $\tilde{a}$ is block-diagonal and can be inverted at the element level. The problem reduced to a standard linear system, and solved by a direct method.

File ‘transmission.cc’

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

const Float epsilon = 0.01;

int main(int argc, char**argv) {
    geo omega (argv[1]);
    space Vh (omega, "P1");
    if (omega.dimension() <= 2) {
      Vh.block ("left"); Vh.block ("right");
    } else {
      Vh.block ("back"); Vh.block ("front");
    }
    field uh (Vh);
    if (omega.dimension() <= 2)
      uh["left"] = uh["right"] = 0;
    else
      uh["back"] = uh["front"] = 0;
    field fh (Vh, 1);

    space Qh (omega, "P0", "vector");
    field eta (Qh);
    eta ["east"] = 1;
    eta ["west"] = epsilon;
    form_diag d (eta);
    form grad (Vh, Qh, "grad");
    form m    (Vh, Vh, "mass");
    form inv_m (Qh, Qh, "inv_mass");
    form a = trans(grad)*(inv_m*d)*grad;
    ssk<Float> fact = ldlt(a.uu);
    uh.u = fact.solve (m.uu*fh.u + m.ub*fh.b - a.ub*uh.b);
    cout << setprecision(numeric_limits<Float>::digits10)
         << catchmark("epsilon") << epsilon << endl
         << catchmark("u")       << uh;
    return 0;
}
```

Comments

The code begins as usual. In the second part, the space $\mathbf{Q}_h$ is defined. For convenience, the diffusion coefficient η is introduced diagonal diffusion tensor operator d, i.e. for $N = 2$:

$$d = \begin{pmatrix} \eta & 0 \\ 0 & \eta \end{pmatrix}$$

It is first defined as an element of $\mathbf{Q}_h$ i.e. a vector-valued field:

```
field eta (Qh);
```

and then converted to a diagonal form representing the diffusion tensor operator:

```
        form_diag d (eta);
```

Notice that, by this way, an anisotropic diffusion operator could be considered. The statement

```
        cout << setprecision(numeric_limits<Float>::digits10)
             << catchmark("epsilon") << epsilon << endl
             << catchmark("u")       << uh;
```

writes ε and u_h in a labeled format, suitable for post-traitement, visualization and error analysis. The number of digits printed depends upon the `Float` type. When `Float` is a `double`, this is roughly 15 disgits; this number depends upon the machine precision. Here, we choose to print all the relevant digits: this choice is suitable for the following error analysis.

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-region.geo
        geo -gnuplot line-region.geo
```

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

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

In order to avoid mistakes with the `C++` style indexation, we denote by $(x_0, \ldots, x_{N-1})$ the cartesian coordinate system in $\mathbb{R}^N$.
Finaly, run the program and look at the solution:

```
        ./transmission line-region.geo > transmission1d.mfield
        mfield transmission1d.mfield -u | field -
```

The two dimensionnal case corresponds to the commands:

```
        mkgeo_grid -t 10 -region > square-region.geo
        geo square-region.geo
        ./transmission square-region.geo > transmission2d.mfield
        mfield transmission2d.mfield -u | field -
```

while the tridimensional to

```
mkgeo_grid -T 10 -region > cube-region.geo
./transmission cube-region.geo > transmission3d.mfield
mfield transmission3d.mfield -u | field -mayavi -
```

This problem is convenient, since the exact solution is known and is piecewise second order polynomial:

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

Since the change in the diffusion coefficient value fits the element boundaries, obtain the exact solution at the vertices of the mesh for the P_1 approximation, as shown on Fig. 6.1.

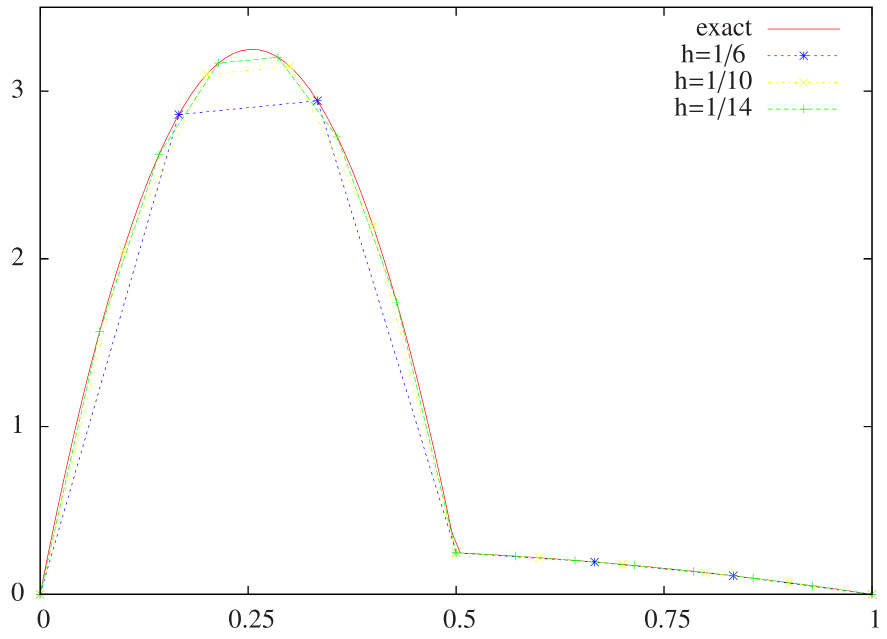


Figure 6.1: Transmission problem: $u_h = \pi_h(u)$ ($\varepsilon = 10^{-2}$, $N = 1$, P_1 approximation).

Chapter 7

The p-laplacian problem

`rheolef` provides build-in classes for solving some linear and non-linear partial differential equations, based on the finite element method (see e.g. [1]). All example files presented along the present manual are available in the `examples/` directory of the `rheolef` distribution, e.g. `/usr/local/share/doc/rheolef/examples/`.

7.1 Problem statement

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

$$\begin{aligned} -\mathrm{div}\left(|\boldsymbol{\nabla} u|^{p-2}\boldsymbol{\nabla} u\right) &= f \text{ in } \Omega \\ u &= 0 \text{ on } \partial\Omega \end{aligned}$$

where f is known and $f = 1$ in the computational examples. When $p = 2$, this problem reduces to the linear Poisson problem with homogeneous Dichlet boundary conditions. Otherwise, for any $p > 1$, the nonlinear problem is equivalent to the following minimisation problem:
(MP): find $u \in W_0^{1,p}(\Omega)$ such that:

$$u = \underset{v \in W_0^{1,p}(\Omega)}{\arg\min} \ \frac{1}{p}\int_\Omega |\nabla v|^p \,\mathrm{d}x - \int_\Omega f\, v \,\mathrm{d}x,$$

where $W_0^{1,p}(\Omega)$ denotes the usual Sobolev spaces of functions in $W^{1,p}(\Omega)$ that vanishes on the boundary [33, p. 118]. The variational formulation of this problem expresses:
(VF): find $u \in W_0^{1,p}(\Omega)$ such that:

$$a(u; u, v) = m(f, v), \ \forall v \in W_0^{1,p}(\Omega)$$

where $a(.,.)$ and $m(.,.)$ are defined for any $u_0, u, v \in W^{1,p}(\Omega)$ by

$$\begin{aligned} a(u_0; u, v) &= \int_\Omega |\nabla u_0|^{p-2}\nabla u.\nabla v \,\mathrm{d}x, \quad \forall u, v \in W_0^{1,p}(\Omega) \\ m(u, v) &= \int_\Omega u\, v \,\mathrm{d}x, \quad \forall u, v \in L^2(\Omega) \end{aligned}$$

The $m(.,.)$ is here the classical scalar product on $L^2(\Omega)$, and is related to the *mass* form. The quantity $a(u; u, u) = \|\boldsymbol{\nabla} u\|_{p,\Omega}$ 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.

7.2 The fixed-point algorithm

7.2.1 Principe of the algorithm

This nonlinear problem is then reduced to a sequence of linear subproblems by using the fixed-point algorithm. The sequence $\left(u^{(n)}\right)_{n\geq 0}$ is defined by recurrence as:

- $n = 0$: let $u^{(0)} \in W_0^{1,p}(\Omega)$ be known.
- $n \geq 0$: suppose that $u^{(n)} \in W_0^{1,p}(\Omega)$ is known and find $u^{(n+1)} \in W_0^{1,p}(\Omega)$ such that:

$$a\left(u^{(n)}; u^{(n+1)}, v\right) = m(f, v),\ \forall v \in W_0^{1,p}(\Omega)$$

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 picewise polynomial functions and V_h, the subspace of X_h containing elements that vanishes on the boundary of Ω:

$$\begin{array}{rcl} X_h & = & \left\{v_h \in C_0^0\left(\overline{\Omega}\right);\ v_{h/K} \in P_k,\ \forall K \in \mathcal{T}_h\right\} \\ V_h & = & \left\{v_h \in X_h;\ v_h = 0 \text{ on } \partial\Omega\right\} \end{array}$$

where $k = 1$ or 2. The approximate problem expresses: suppose that $u_h^{(n)} \in V_h$ is known and find $u_h^{(n+1)} \in V_h$ such that:

$$a\left(u_h^{(n)}; u_h^{(n+1)}, v_h\right) = m(f, v_h),\ \forall v_h \in V_h$$

By developping u_h on a basis of V_h, this problem reduces to a linear system. The implementation with `rheolef`, involving weighted forms, is quite standard: the weight field `wh` is inserted as the last argument to the form constructor. The following code implement this problem in the `rheolef` environment.

7.2.2 File ‘`p-laplacian-fixed-point.h`’

```
int p_laplacian_fixed_point (Float p, field fh, field& uh, Float& r, size_t& n) {
    Float tol = r;
    Float r0 = 0;
    size_t max_iter =  n;
    const geo& omega_h = uh.get_geo();
    const space& Vh = uh.get_space();
    string grad_approx = (Vh.get_approx() == "P2") ? "P1d" : "P0";
    space Th (omega_h, grad_approx, "vector");
    form m (Vh, Vh, "mass");
    form inv_mt (Th, Th, "inv_mass");
    form grad (Vh, Th, "grad");
    cerr << "# Fixed-point algorithm on p-laplancian: p = " << p << endl
         << "# n r v" << endl;
    n = 0;
    do {
      field grad_uh = inv_mt*(grad*uh);
      field wh = pow(sqr(grad_uh[0]) + sqr(grad_uh[1]), p/2.-1);
      form a (Vh, Vh, "grad_grad", wh);
      field rh = a*uh - m*fh;
      r = rh.u.max_abs();
      if (n == 0) r0 = r;
      Float v = (n == 0) ? 0 : log10(r0/r)/n;
      cerr << n << " " << r << " " << v << endl;
      if (r <= tol || n++ >= max_iter) break;
      ssk<Float> fact_a = ldlt(a.uu);
      uh.u = fact_a.solve (m.uu*fh.u + m.ub*fh.b - a.ub*uh.b);
    } while (true);
    return r <= tol;
}
```

7.2.3 File ‘p-laplacian-fixed-point.cc’

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
#include "p-laplacian-fixed-point.h"
#include "poisson-dirichlet.icc"
int main(int argc, char**argv) {
  geo omega_h (argv[1]);
  string approx = (argc > 2) ?      argv[2]  : "P1";
  Float  p      = (argc > 3) ? atof(argv[3]) : 2.5;
  cerr << "# P-Laplacian problem by fixed-point:" << endl
       << "# geo = " << omega_h.name() << endl
       << "# approx = " << approx << endl
       << "# p = " << p << endl;
  space Vh (omega_h, approx);
  Vh.block (omega_h["boundary"]);
  field uh (Vh);
  uh ["boundary"] = 0;
  field fh (Vh, 1);
  poisson_dirichlet (fh, uh);
  Float tol = 10*numeric_limits<Float>::epsilon();
  size_t max_iter = 500;
  int status = p_laplacian_fixed_point (p, fh, uh, tol, max_iter);
  cout << setprecision(numeric_limits<Float>::digits10)
       << catchmark("p") << p << endl
       << catchmark("u") << uh;
  return status;
}
```

7.2.4 File ‘poisson-dirichlet.icc’

```
void poisson_dirichlet (field fh, field& uh) {
    const space& Vh = uh.get_space();
    form a (Vh, Vh, "grad_grad");
    form m (Vh, Vh, "mass");
    ssk<Float> fact = ldlt(a.uu);
    uh.u = fact.solve (m.uu*fh.u + m.ub*fh.b - a.ub*uh.b);
}
```

7.2.5 Comments

The fixed-point algorithm is amorced with $u^{(0)}$ as the solution of the linear problem associated to $p = 2$, i.e. the standard Poisson problem with Dirichlet boundary conditions. The construction of the weighted form $a(.;.,.)$ writes:

```
field eta_h = pow(sqr(grad_uh[0]) + sqr(grad_uh[1]), p/2.-1);
form a (Vh, Vh, "grad_grad", eta_h);
```

7.2.6 Running the program

We assume that the previous code is contained in the file ‘`p-laplacian-fixed-point.cc`’. Compile the program, as usual:

```
make p-laplacian-fixed-point
```

and enter the comands:

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

The triangular mesh has a boundary domain named `boundary`.

```
./p-laplacian-fixed-point square.geo P1 1.1 > square-P1.field
```

The previous command solves the problem for the corresponding mesh and writes the solution in the file format ‘`.field`’.

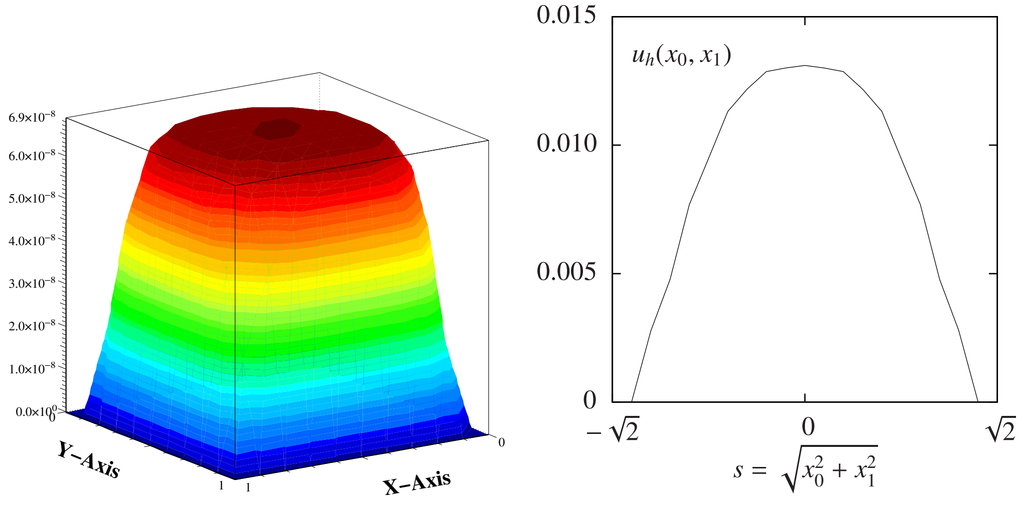


Figure 7.1: The p-laplacian for $N = 2$: (a) elevation view for $p = 1.1$; (b) cut along the first bissectice $x_0 - x_1 = 0$.

Run the field visualization:

```
field square-P1.field -elevation
field square-P1.field -cut -origin 0.5 0.5 -normal 1 1 -gnuplot
```

The first command shows an elevation view of the solution (see 7.1.a) while the second one shows a cut along the first bissectrice $x_0 = x_1$. (see 7.1.b).

7.2.7 Convergence properties of the fixed-point algorithm

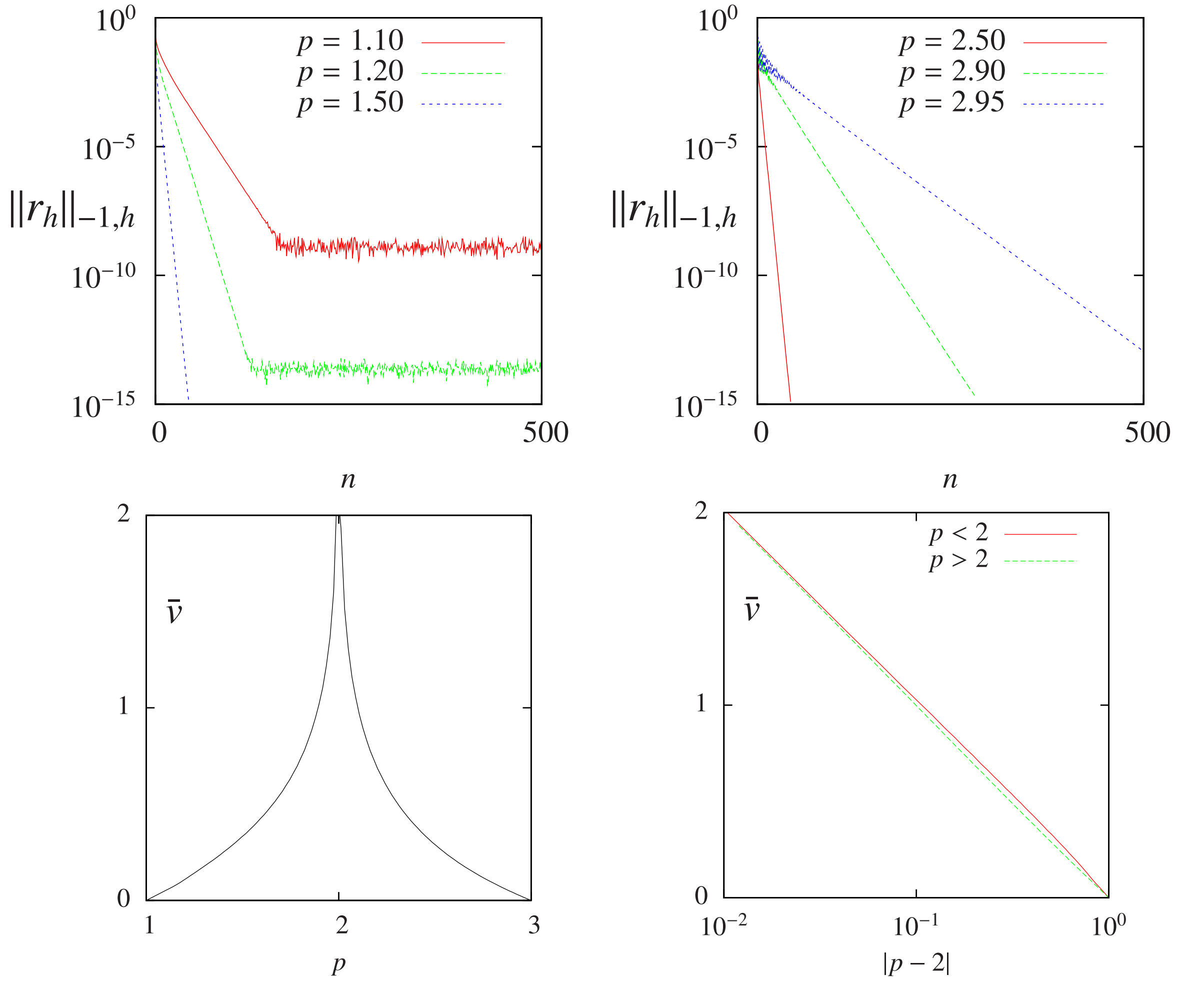


Figure 7.2: The fixed-point algorithm on the p-laplacian for $N = 2$: (a) convergence when $p < 2$; (b) when $p > 2$; (c) convergence rate versus p; (d) convergence rate versus p in semi-log scale.

The fixed-point algorithm prints also at each iteration n, the residual term r_n in discrete $H^{-1}(\Omega)$ and the convergence rate $v_n = log10(r_n/r_0)/n$. The residual term is defined by

$$r_h^{(n)} = A_h\left(u^{(n)}\right) - M_h f_h$$

where A_h and M_h are the discrete operators induced by the forms $a(.;.,.)$ and $m(.,.)$ on V_h, and defined for all $u_h, v_h \in Vh$ by:

$$\begin{array}{rcl} A_h(u_h)\, v_h^T & = & a(u_h; u_h, v_h) \\ (M_h u_h)\, v_h^T & = & m(u_h, v_h) \end{array}$$

where the elements of V_h are identified to elements of $\mathbb{R}^{\dim(V_h)}$. The $W^{-1,p}(\Omega)$ norm, defined for all $r \in W^{-1,p}(\Omega)$ by duality:

$$\|r\|_{-1,p,\Omega} = \sup_{\substack{v \in W^{1,p}(\Omega) \\ \|v\|_{1,p,\Omega}=1}} m(r,v)$$

By analogy, the discrete $W^{-1,p}(\Omega)$ norm, denoted as $\|.\|_{-1,h}$, is defined by duality for all $r_h \in V_h$ by:

$$\|r_h\|_{-1,h} = \sup_{\substack{v_h \in V_h \\ \|v_h\|_{1,p,\Omega}=1}} m(r_h,v_h) = \sup_{\mathbf{x} \in \mathrm{xdof}(V_h)} |r_h(\mathbf{x})|$$

where xdof(V_h) denotes the set of nodes associated to the V_h degrees of freedom. Since elements of V_h vanishes on the boundary, the xdof(V_h) contains all nodes associated to the degrees of freedoms of X_h except nodes located on the boudnary. Fig 7.2.a and 7.2.b show that the residual term decreases exponentially versus n, since the slope of the plot in semi-log scale tends to be strait. Thus, the convergence rate $v_n = log10(r_n/r_0)/n$ tends to a constant, denoted by $\bar{v}$. Fig 7.2.c shows the dependence of $\bar{v}$: $r_n \approx r_0 \times 10^{-\bar{v}\,n}$. Observe 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 in $p = 1$ and $p = 3$. The singularity in $p = 1$ is not surprising, since the problem is defined only when $p > 1$. Conversely, the singularity in $p = 3$ is not clear and requires more analysis. Fig 7.2.d shows the same plot in semi-log scale and shows that $\bar{v}$ behaves as: $\bar{v} \approx 2 \times \log_{10}(|p-2|)$. Finally, this study shows that the residual term behaves as:

$$r_n \approx 2\,|p-2|\,r_0\,10^{-n}$$

7.3 The Newton algorithm

7.3.1 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:

$$\begin{array}{rcll} F & : & W_0^{1,p}(\Omega) & \longrightarrow & W^{-1,p}(\Omega) \\ & & u & \longmapsto & F(u) = -\mathrm{div}\left(|\boldsymbol{\nabla} u|^{p-2}\boldsymbol{\nabla} u\right) - f \end{array}$$

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 $\left(u^{(n)}\right)_{n\geq 0}$ is classically defined by recurrence as:

- $n = 0$: let $u^{(0)} \in W_0^{1,p}(\Omega)$ be known.
- $n \geq 0$: 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 explicitely:

  $$u^{(n+1)} := u^{(n)} + \delta u^{(n)}$$

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

$$\begin{array}{rcll} -\mathrm{div}\left(|\boldsymbol{\nabla} u|^{p-2}\boldsymbol{\nabla}(\delta u) + (p-2)|\boldsymbol{\nabla} u|^{p-4}\left\{\boldsymbol{\nabla} u.\boldsymbol{\nabla}(\delta u)\right\}\boldsymbol{\nabla} u\right) & = & -r & \text{in } \Omega \\ \delta u & = & 0 & \text{on } \partial\Omega \end{array}$$

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

$$a_1(u;\delta u,\delta v) = m(r,v), \quad \forall \delta v \in W_0^{1,p}(\Omega)$$

where the $a_1(.;.,.)$ is defined for any $u, \delta u, \delta v \in W_0^{1,p}(\Omega)$ by:

$$a_1(u;\delta u,\delta v) = \int_\Omega \left(|\boldsymbol{\nabla} u|^{p-2}\boldsymbol{\nabla}(\delta u).\boldsymbol{\nabla}(\delta v) + (p-2)|\boldsymbol{\nabla} u|^{p-4}\left\{\boldsymbol{\nabla} u.\boldsymbol{\nabla}(\delta u)\right\}\left\{\boldsymbol{\nabla} u.\boldsymbol{\nabla}(\delta v)\right\}\right)\mathrm{d}x$$

For any $\boldsymbol{\xi} \in \mathbb{R}^N$ let us denote by $\eta(\boldsymbol{\xi})$ the following $N \times N$ matrix:

$$\eta(\boldsymbol{\xi}) = |\boldsymbol{\xi}|^{p-2}\, I + (p-2)|\boldsymbol{\xi}|^{p-4}\ \boldsymbol{\xi}\otimes\boldsymbol{\xi}$$

where I stands for the N-order identity matrix. Then the a_1 expresses in a more compact form:

$$a_1(u;\delta u,\delta v) = \int_\Omega \left(\eta(\boldsymbol{\nabla} u)\boldsymbol{\nabla}(\delta u)\right).\boldsymbol{\nabla}(\delta v)\,\mathrm{d}x$$

Clearly a_1 is linear and symmetric with respect to the two last variables.

7.3.2 File ‘`p-laplacian-newton.cc`’

```
#include "rheolef.h"
#include "rheolef/newton.h"
using namespace rheolef;
using namespace std;
#include "p-laplacian.h"
int main(int argc, char**argv) {
  geo omega_h (argv[1]);
  string approx = (argc > 2) ?      argv[2]  : "P1";
  Float  p      = (argc > 3) ? atof(argv[3]) : 2.5;
  cerr << "# P-Laplacian problem by Newton:" << endl
       << "# geo = " << omega_h.name() << endl
       << "# approx = " << approx << endl
       << "# p = " << p << endl;
  p_laplacian F (p, omega_h, approx);
  field uh = F.initial();
  Float tol = 1e6*numeric_limits<Float>::epsilon();
  size_t max_iter = 500;
  int status = newton (F, uh, tol, max_iter, &cerr);
  cout << setprecision(numeric_limits<Float>::digits10)
       << catchmark("p") << p << endl
       << catchmark("u") << uh;
  return status;
}
```

7.3.3 File ‘p-laplacian.h’

```
class p_laplacian {
public:
  typedef field value_type;
  typedef Float float_type;
  p_laplacian(Float p, const geo& omega_h, string approx = "P1");
  void reset(const geo& omega_h, string approx = "previous");
  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 norm (const field& uh) const;
  Float dual_norm (const field& Muh) const;
  Float dot (const field& uh, const field& vh) const;
  Float dual_dot (const field& Muh, const field& Mvh) const;
  field criteria(const field& uh) const;
  Float p;
  space Vh, Kh;
  field fh;
  form m, inv_mt, grad;
  ssk<Float> fact_m;
  mutable form a1;
  mutable ssk<Float> fact_a1;
};
#include "p-laplacian.icc"
```

7.3.4 File ‘p-laplacian.icc’

```
#include "poisson-dirichlet.icc"
p_laplacian::p_laplacian(Float p1, const geo& omega_h, string approx1)
 : p(p1), Vh(), Kh(), fh(),
  m(), inv_mt(), grad(), fact_m(), a1(), fact_a1() {
  reset(omega_h, approx1);
}
void p_laplacian::reset(const geo& omega_h1, string approx1) {
  if (approx1 == "previous") approx1 = Vh.get_approx();
  Vh = space(omega_h1, approx1);
  Vh.block (omega_h1["boundary"]);
  fh = field(Vh, 1);
  m = form (Vh, Vh, "mass");
  fact_m = ldlt(m.uu);
  string grad_approx = (Vh.get_approx() == "P2") ? "P1d" : "P0";
  space Th (fh.get_geo(), grad_approx, "vector");
  inv_mt = form (Th, Th, "inv_mass");
  grad = form (Vh, Th, "grad");
  Kh = space(fh.get_geo(), grad_approx, "tensor");
}
field p_laplacian::initial () const {
  field uh(Vh);
  uh [Vh.get_geo()["boundary"]] = 0;
  poisson_dirichlet (fh, uh);
  return uh;
```

```
}
void p_laplacian::update_derivative (const field& uh) const {
  field grad_uh = inv_mt*(grad*uh);
  field norm2_grad_uh = euclidian_norm2(grad_uh);
  field w0h = pow(norm2_grad_uh, p/2)/norm2_grad_uh;
  field w1h = pow(norm2_grad_uh, p/2)/sqr(norm2_grad_uh);
  field eta_h (Kh);
  eta_h(0,0) = w0h + (p-2)*w1h*sqr(grad_uh[0]);
  if (uh.dimension() >= 2) {
    eta_h(1,1) = w0h + (p-2)*w1h*sqr(grad_uh[1]);
    eta_h(0,1) =       (p-2)*w1h*grad_uh[0]*grad_uh[1];
  }
  if (uh.dimension() == 3) {
    eta_h(2,2) = w0h + (p-2)*w1h*sqr(grad_uh[2]);
    eta_h(1,2) =       (p-2)*w1h*grad_uh[1]*grad_uh[2];
    eta_h(0,2) =       (p-2)*w1h*grad_uh[0]*grad_uh[2];
  }
  a1 = form (Vh, Vh, "grad_grad", eta_h);
  fact_a1 = ldlt(a1.uu);
}
field p_laplacian::residue (const field& uh) const {
  field grad_uh = inv_mt*(grad*uh);
  field norm2_grad_uh = euclidian_norm2(grad_uh);
  field w0h = pow(norm2_grad_uh, p/2-1);
  form a (Vh, Vh, "grad_grad", w0h);
  field mrh = a*uh - m*fh;
  mrh.b = 0;
  return mrh;
}
field p_laplacian::derivative_solve (const field& mrh) const {
  field delta_uh (Vh,0);
  delta_uh.u = fact_a1.solve(mrh.u);
  delta_uh.b = 0;
  return delta_uh;
}
field p_laplacian::derivative_trans_mult (const field& mrh) const {
  field rh (Vh);
  rh.u = fact_m.solve(mrh.u);
  rh.b = 0;
  field mgh;
  mgh = a1*rh;
  mgh.b = 0;
  return mgh;
}
Float p_laplacian::dot (const field& uh, const field& vh) const {
  return m(uh,vh);
}
Float p_laplacian::norm (const field& uh) const {
  return sqrt(m(uh,uh));
}
Float p_laplacian::dual_dot (const field& mrh, const field& msh) const {
  field sh (Vh);
  sh.u = fact_m.solve(msh.u);
  sh.b = 0;
  return ::dot(mrh,sh);
}
Float p_laplacian::dual_norm (const field& mrh) const {
  return sqrt(dual_dot(mrh,mrh));
}
field p_laplacian::criteria(const field& uh) const {
  if (uh.get_approx() == "P1") return abs(uh);
  field grad_uh = inv_mt*(grad*uh);
  field norm2_grad_uh = euclidian_norm2(grad_uh);
  return pow(norm2_grad_uh, p/4);
}
```

7.3.5 Comments

The code implements a generic Newton algorithm in the file '`newton.h`'. The main program is '`p-laplacian-newton.cc`', that uses a class `p-laplacian`. This class interface is definied in the file '`p-laplacian.h`' and its implementation in '`p-laplacian.icc`' 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`. Notice that the $a_1(u; ., .)$ bilinear form is a tensorial weighted form, where $\eta(\boldsymbol{\nabla} u)$ is the weight tensor. In `rheolef`, the tensorial weight field `eta_h` is inserted as an usual scalar weight, by passing the weight parameter as the last argument to the form constructor. The introduction of the class '`p-laplacian`' allows an easiest implementation of several variants of the Newton algorithm.

7.3.6 Running the program

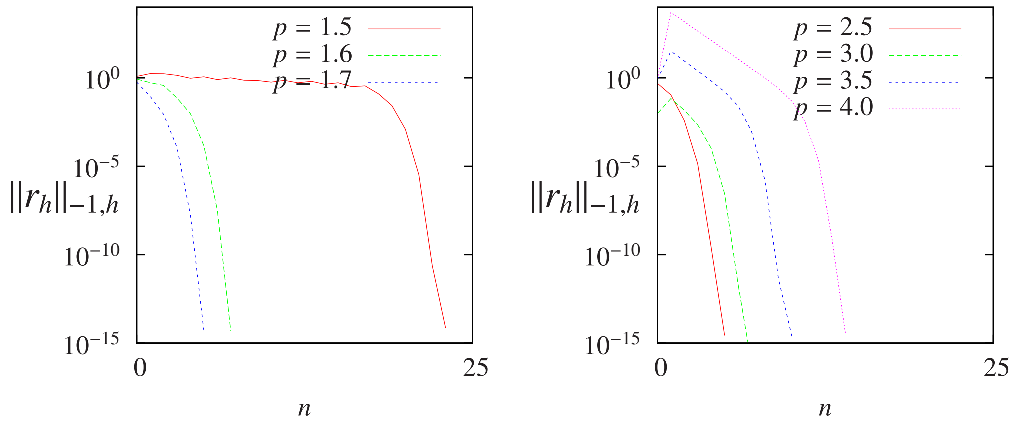


Figure 7.3: The Newton algorithm on the p-laplacian for $N = 2$: (a) convergence when $p < 2$; (b) when $p > 2$.

We assume that the previous code is contained in the file '`p-laplacian-newton.cc`'. As usual, enter:

```
make p-laplacian-newton
mkgeo_grid -t 10 -boundary > square.geo
./p-laplacian-newton square.geo P1 1.5 | field -
```

The program prints at each iteration n, the residual term r_n in discrete $L^2(\Omega)$ norm. Fig. 7.3.a and. 7.3.b shows that the residual terms tends very fast to zero. 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. Also the algorithm converge when $p \geq 3$, until $p = \approx 4$. It was not the case with the fixed point algorithm that diverges in that case. Finally, the Newton algorithm diverges for small values of p, e.g. $p < 1.5$ and the plot is not showed here. Conversely, when $p > 4$, the first iterations increases dramatically the residual terms, before to decrease. In that case, another strategy should be considered: the damped Newton algorithm. This is the subject of the next section.

7.4 The damped Newton algorithm

7.4.1 Principe of the algorithm

The Newton algorithm diverges when the initial $u^{(0)}$ is too far from a solution. Our aim is to modify the Newton algorithm and to obtain a *globaly convergent algorith*, i.e to converge to a solution for any initial $u^{(0)}$. 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 $n \geq 0$:

$$T\left(u^{(n+1)}\right) \leq T\left(u^{(n)}\right) + \alpha \left\langle T'\left(u^{(n)}\right), \; u^{(n+1)} - u^{(n)} \right\rangle_{V',V} \tag{7.1}$$

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 $n \geq 0$: 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 [34] and is also presented in [35, p. 385]. The step length λ that satify (7.1) is computed by using a finite sequence λ_k, $k = 0, 1 \ldots$ with a second order recurrence:

- $k = 0$: initialisation $\lambda_0 = 1$. If (7.1) is satified whith $u + \lambda_0\, d$ 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 initialisation. Also, we already have computed $g(1) = f(u+d)$ when verifiying whether (7.1) was satified. Thus, we consider the following approximation of $g(\lambda)$ by a second order polynom:

  $$\tilde{g}_1(\lambda) = \{g(1) - g(0) - g'(0)\}\lambda^2 + g'(0)\lambda + g(0)$$

  After a short computation, we find that the minimum of this polynom is:

  $$\tilde{\lambda}_1 = \frac{-g'(0)}{2\{g(1) - g(0) - g'(0)\}}$$

  Since the initialisation at $k = 0$ does not satisfy (7.1), it is possible to show that, when α is small enougth, we have $\tilde{\lambda}_1 \leq 1/2$ and $\tilde{\lambda}_1 \approx 1/2$. Let $\lambda_1 := \max(\lambda_{\min}, \tilde{\lambda}_1)$. If (7.1) is satisfied with $u + \lambda_1\, d$ then let $\lambda := \lambda_1$ and the sequence stop here.
- $k \geq 2$: second order recurrence. The quantities $g(0) = f(u)$ et $g'(0) = \rangle f'(u),\, d\langle$ are available, ytogether with λ_{k-1}, $g(\lambda_{k-1})$, λ_{k-2} and $g(\lambda_{k-2})$. Then, $g(\lambda)$ is approximed by the following third order polynom:

  $$\tilde{g}_k(\lambda) = a\lambda^3 + b\lambda^2 + g'(0)\lambda + g(0)$$

  where a et b are expressed by:

  $$\left(\begin{array}{c} a \\ b \end{array}\right) = \frac{1}{\lambda_{k-1} - \lambda_{k-2}} \left(\begin{array}{cc} \frac{1}{\lambda_{k-1}^2} & -\frac{1}{\lambda_{k-2}^2} \\ -\frac{\lambda_{k-2}}{\lambda_{k-1}^2} & \frac{\lambda_{k-1}}{\lambda_{k-2}^2} \end{array}\right) \left(\begin{array}{c} g(\lambda_{k-1}) - g'(0)\lambda_{k-1} - g(0) \\ g(\lambda_{k-2}) - g'(0)\lambda_{k-2} - g(0) \end{array}\right)$$

  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 (7.1) is satisfied with $u + \lambda_k\, d$ then let $\lambda := \lambda_k$ and the sequence stop here.

The sequence $(\lambda_k)_{k\geq 0}$ is strictly decreasing: when the stopping criteria is not satified until λ_k reaches the machine precision $\varepsilon_{\text{mach}}$ then the algorithm stops with an error.

7.4.2 File ‘p-laplacian-damped-newton.cc’

```
#include "rheolef.h"
#include "rheolef/damped-newton.h"
using namespace rheolef;
using namespace std;
#include "p-laplacian.h"
int main(int argc, char**argv) {
  geo omega_h (argv[1]);
  string approx = (argc > 2) ?      argv[2]  : "P1";
  Float  p      = (argc > 3) ? atof(argv[3]) : 2.5;
  cerr << "# P-Laplacian problem by Newton:" << endl
       << "# geo = " << omega_h.name() << endl
       << "# approx = " << approx << endl
       << "# p = " << p << endl;
  p_laplacian F (p, omega_h, approx);
  field uh = F.initial();
  Float tol = numeric_limits<Float>::epsilon();
  size_t max_iter = 500;
  int status = damped_newton (F, uh, tol, max_iter, &cerr);
  cout << setprecision(numeric_limits<Float>::digits10)
       << catchmark("p") << p << endl
       << catchmark("u") << uh;
  return status;
}
```

7.4.3 Comments

The file `damped-newton-generic.h` 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 file '`newton-backtrack.h`'. 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.

7.4.4 Running the program

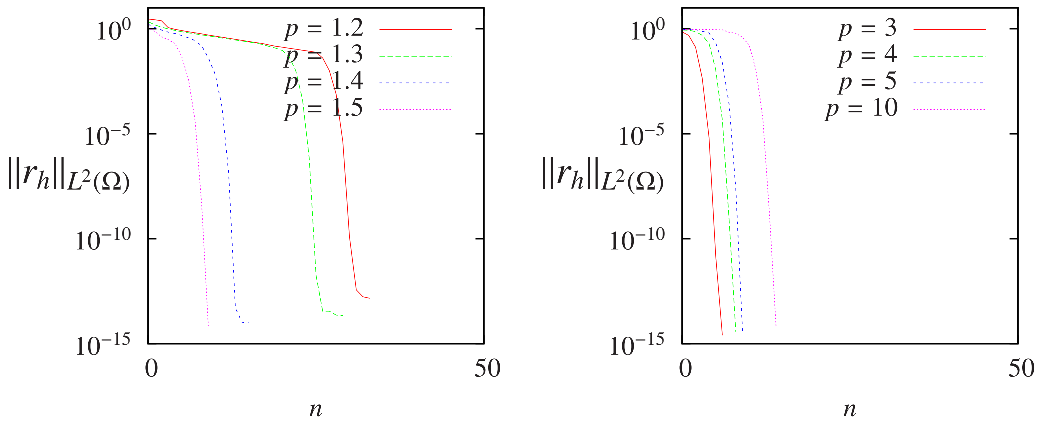


Figure 7.4: The damped Newton algorithm on the p-laplacian for $N = 2$: (a) convergence when $p < 2$; (b) when $p > 2$.

We assume that the previous code is contained in the file '`p-laplacian-damped-newton.cc`'. As usual, enter:

```
make p-laplacian-damped-newton
mkgeo_grid -t 10 -boundary > square.geo
```

```
./p-laplacian-damped-newton square.geo P1 1.5 | field -
./p-laplacian-damped-newton square.geo P1 5.0 | field -
```

The algorithm is now quite robust: the convergence occurs for a large range of $p > 1$ values. and has been pushed until $p = 100$. The only limitation is due to machine roundoff when $p = 1.1$: the residual term reaches only 10^{-10} instead of 10^{-15}.

7.4.5 Robustness and mesh invariance

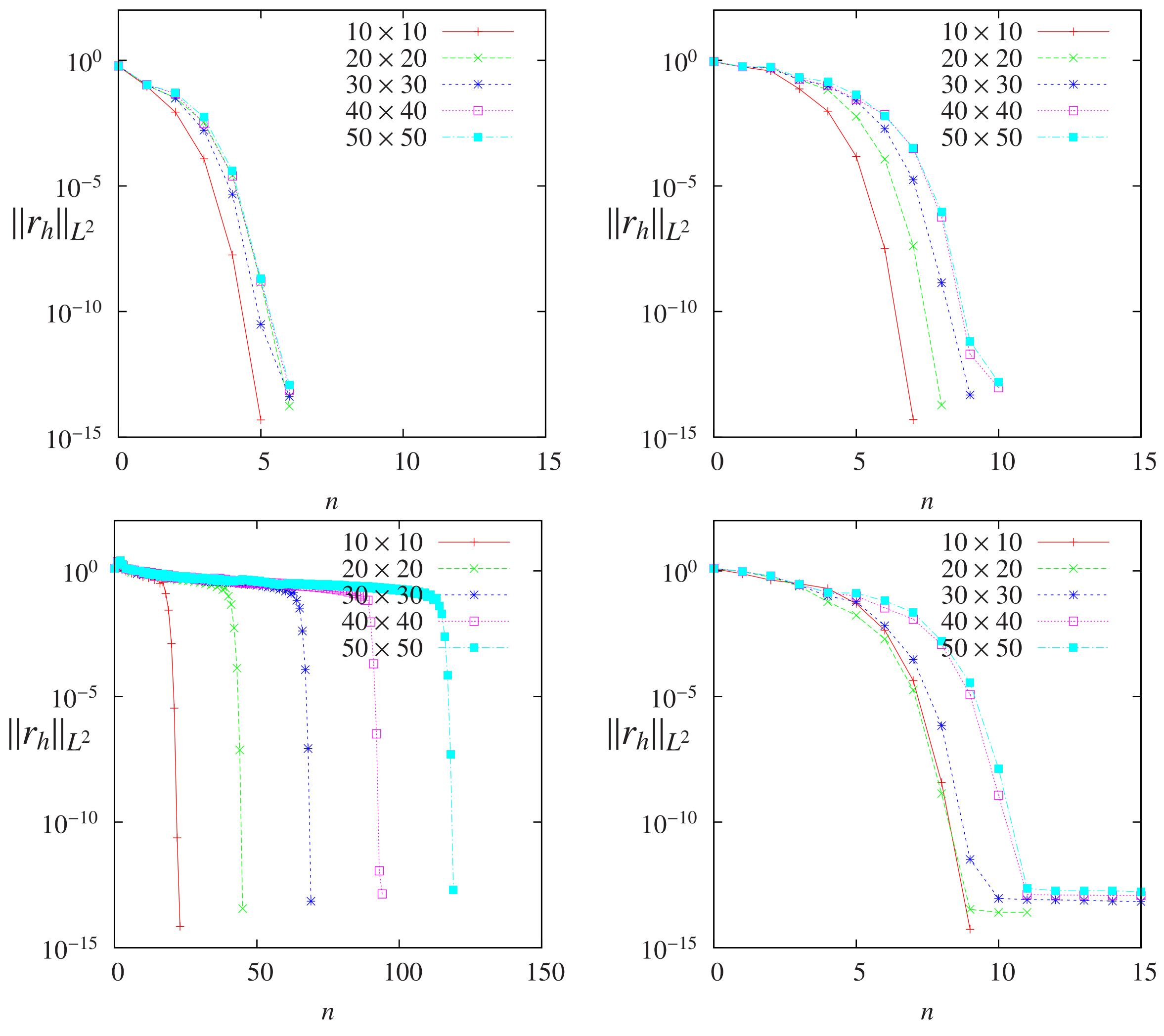


Figure 7.5: Convergence versus n for various meshes: (a) Newton algorithm when $p = 1.7$ and (b) $p = 1.6$; (c) Newton algorithm when $p = 1.5$; (d) damped-Newton algorithm when $p = 1.5$.

Fig. 7.5.a, 7.5.b and. 7.5.c show the convergence of the Newton method when $p = 1.7$, 1.6 and 1.5, respectively. Observe that the convergence is asymptotically invariant of the mesh when the element size decreases. The convergence is more difficult when p decreases to 1.5 and the Newton algorithm is no more mesh invariant. Fig. 7.5.d shows the convergence of the damped Newton method when $p = 1.5$: the convergence is now very fast and also mesh-invariant.

Chapter 8

Error analysis and singularities

This chapter is in progess...

8.1 Error analysis

(Source file: '`doc/usrman/dirichlet-nh-error.cc`')

8.1.1 Principe

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

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

$$\|u - u_h\|_{0,\infty,\Omega} \approx O(h^{k+1})$$

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

$$\|u - u_h\|_{0,2,\Omega} \leq \|u - \pi_h u\|_{0,2,\Omega} + \|u_h - \pi_h u\|_{0,2,\Omega}$$

Since $\|u - \pi_h u\|_{0,2,\Omega} \approx O(h^{k+1})$, we have just to check the $\|u_h - \pi_h u\|_{0,2,\Omega}$ error term.

8.1.2 Implementation

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

Float u (const point& x) { return sin(x[0]+x[1]+x[2]); }

int main(int argc, char**argv)
{
    field u_h;
    cin >> u_h;
    space Vh = u_h.get_space();
    field pi_h_u = interpolate(Vh, u);
    field eh = pi_h_u - u_h;
    form m(Vh, Vh, "mass");
    cout << "error_inf "  << eh.max_abs()       << endl;
    cout << "error_l2  "  << sqrt(m(eh,eh)) << endl;
    return 0;
}
```

8.1.3 Running the program

Remarks on step (b) the use of the `get_space` member function. Thus, the previous implementation does not depend upon the degree of the polynomial approximation.

After compilation, run the code by using the command:

```
dirichlet-nh square-h=0.1.geo P1 | dirichlet-nh-error
```

The two errors in L^∞ and L^2 are printed for a $h = 0.1$ quasi-uniform mesh.

Let *nelt* denotes the number of elements in the mesh. Since the mesh is quasi-uniform, we have $h \approx nelt^{\frac{1}{N}}$. Here $N = 2$ for our bidimensionnal mesh. The figure 8.1 plots in logarithmic scale the error versus $nelt^{\frac{1}{2}}$ for both P_1 (on the left) and P_2 (on the right) approximations.

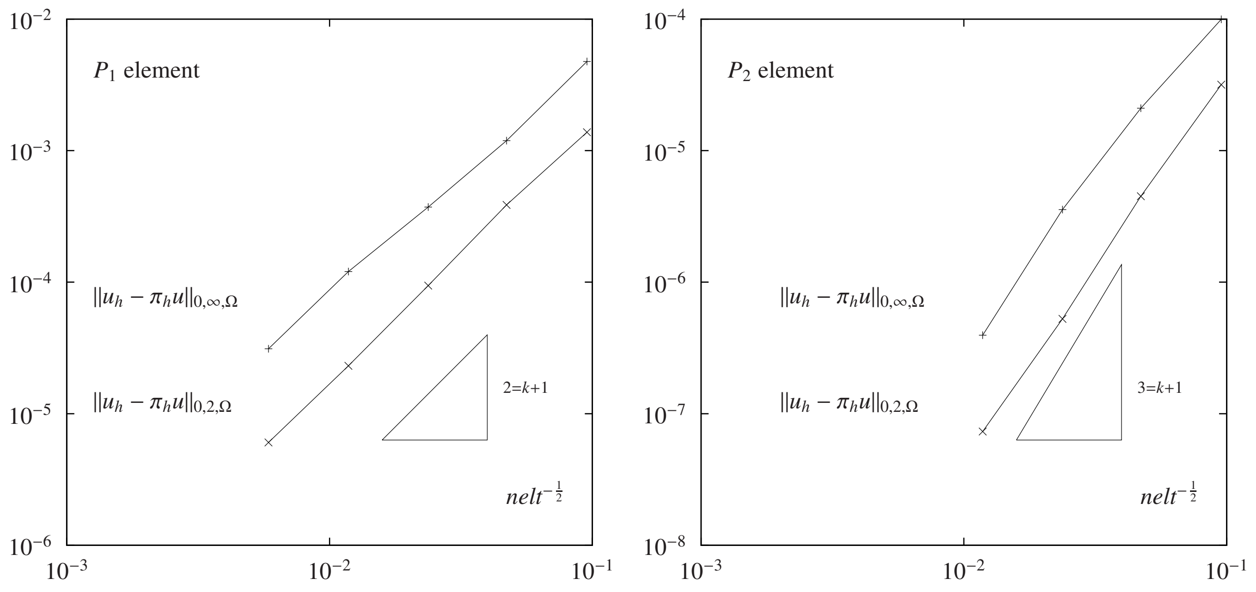


Figure 8.1: Error analysis in L^2 and L^∞ norms.

8.2 Computing the Gradient

(Source file: '`doc/usrman/d-dx.cc`')

Remark that the gradient $\mathbf{q}_h = \nabla u_h$ of a continuous picewise linear function u_h is a discontinuous piecewise constant function. Conversely, the gradient of a continuous picewise quadratic function is a discontinuous piecewise linear function.

8.2.1 Formulation

For all $i = 1 \ldots N$, the i-th component $q_{h,i}$ of the gradient belongs to the space:

$$T_h \; \{\phi_h \in L^2(\Omega);\; \phi_{h/K} \in P_{k-1},\; \forall K \in \mathcal{T}_h\}$$

By introducing the two following bilinear forms:

$$m_T(\psi_h, \phi_h) = \int_\Omega \psi_h \phi_h \, dx$$

$$b_i(u_h, \phi_h) = \int_\Omega \frac{\partial u_h}{\partial x_i} \phi_h \, dx$$

the gradient satisfies the following variational formulation:

$$m_T(q_{h,i}, \phi_h) = b_i(u_h, \phi_h), \ \forall \psi_h \in Th$$

Remark that the matrix associated to the $m_T(.,,)$ bilinear form is block-diagonal. Each block is associated to an element, and can be inverted at the element level. The following code uses this property on step (e), as the `"inv_mass"` form.

8.2.2 Implementation

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

string get_approx_grad (const space &Vh)
{
    string Pk = Vh.get_approx();
    if (Pk == "P1") return "P0";
    if (Pk == "P2") return "P1d";
    cerr << "unexpected approximation " << Pk << endl;
    exit (1);
}
int main(int argc, char**argv)
{
    field uh;
    cin >> uh;
    space Vh      = uh.get_space();
    geo   omega_h = Vh.get_geo();
    string approx_grad = get_approx_grad(Vh);
    space Th (omega_h, approx_grad);
    form b(Vh, Th, "d_dx0");
    form inv_mass (Th, Th, "inv_mass");
    field du_dx0 = inv_mass*(b*uh);
    int digits10 = numeric_limits<Float>::digits10;
    cout << setprecision(digits10) << du_dx0;
}
```

8.3 Error analysis for the gradient

(Source file: ‘`doc/usrman/demo2-error-grad.cc`’)

The following estimation holds for the gradient:

$$\|\nabla u - \nabla u_h\|_{0,2,\Omega} \approx O(h^k)$$

$$\|\nabla u - \nabla u_h\|_{0,\infty,\Omega} \approx O(h^k)$$

The figure 8.2 plots in logarithmic scale the error for the gradient versus $nelt^{\frac{1}{2}}$ for both P_1 (on the left) and P_2 (on the right) approximations.

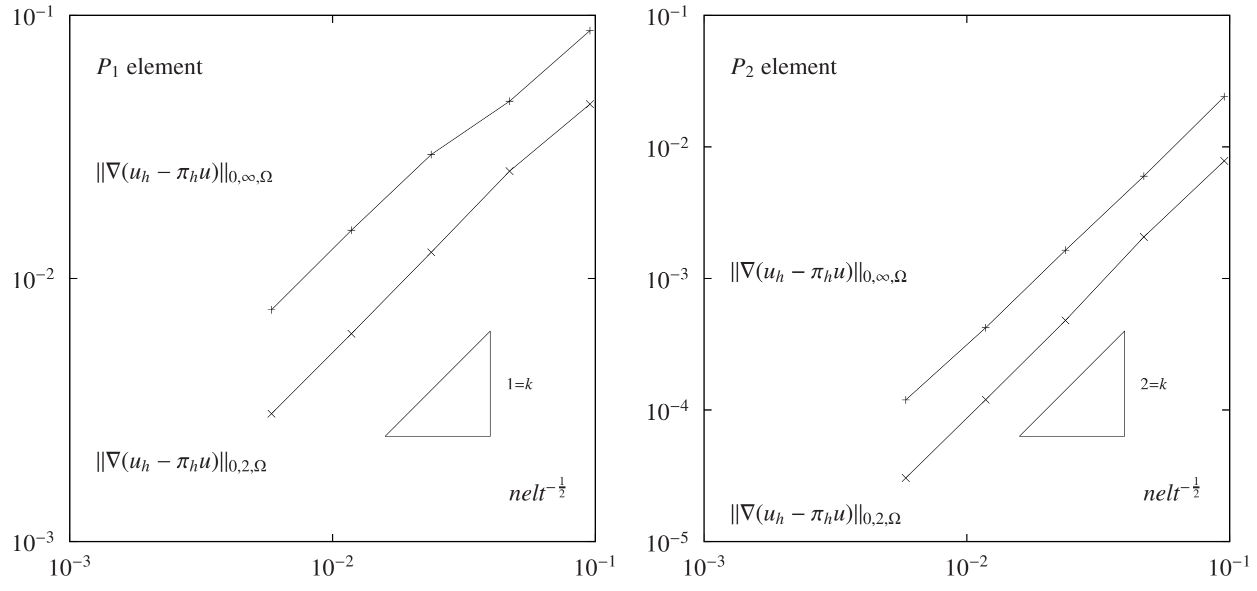


Figure 8.2: Error analysis for the gradiend: H^1 and $W^{1,\infty}$ norms.

8.3.1 Implementation

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

Float du_dxi (const point& x) { return cos(x[0]+x[1]+x[2]); }

int main(int argc, char**argv)
{
    field qh;
    cin >> qh;
    space Th = qh.get_space();
    field pi_h_q = interpolate(Th, du_dxi);
    field eh = pi_h_q - qh;
    form m(Th, Th, "mass");
    cerr << "error_inf "  << eh.max_abs()       << endl;
    cerr << "error_l2  "  << sqrt(m(eh,eh)) << endl;
    if (argc > 1) cout << eh;
    return 0;
}
```

8.4 A problem with singularity

(Source file: ‘`doc/usrman/crack.cc`’)

8.4.1 Formulation

Let us now consider the Laplace operator in a non-convex domain. The domain is a disk with a missing part, thus there exists a re-entrant corner (see Figure, on the left). The angle associated to the re-entrant corner is denoted by ω and whe suppose $\pi < \omega \le 2\pi$. We assume homogeneous Dirichlet conditions on the boundaries corresponding to the radius, and denoted by Γ_0. The rest of the boundary is related to the envelop boundary and denoted by Γ_e.

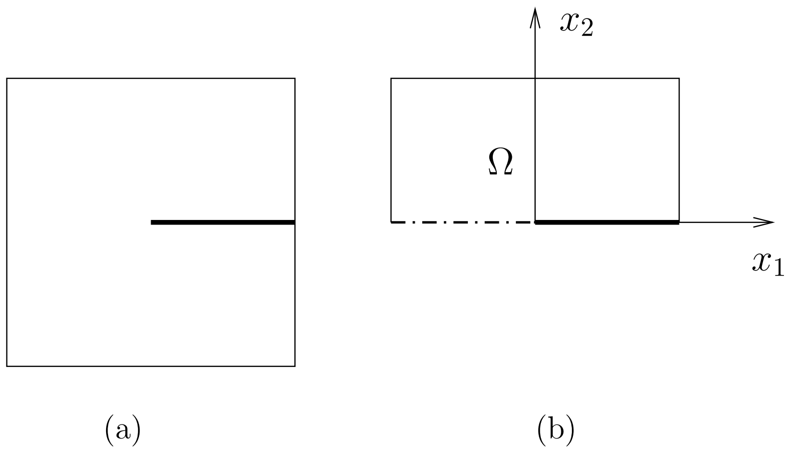


Figure 8.3: The domain of computation: (a) with a crack; (b) by using the Ox_1 symetry.

The problem expresses:

find u defined in Ω such that

$$\begin{array}{lcl} -\Delta u &=& 0 \text{ in } \Omega \\ u &=& 0 \text{ on } \Gamma_0 \\ u &=& g \text{ on } \Gamma_e \end{array}$$

where g is expressed by the use of polar coordinates:

$$g(r,\theta) = r^{\alpha}\sin(\alpha\theta)$$

and

$$\alpha = \frac{\pi}{\omega}$$

The parameter α is the intensity of the singularity at $x = 0$ (see e.g. [36]). This problem is convenient since the exact solution is known: $u = r^{\alpha}\sin(\alpha\theta)$. The problem for $\omega = 2\pi$ is related to a crack (see Figure, on the right) and the intensity is $\alpha = 1/2$. Notes that there is a symmetry axis, and the domain can be reduced to $y > 0$. The symmetry axis is associated to a homogeneous Neumann condition for $x < 0$, and the corresponding boundary is denoted by Γ_s. For the problem reduced to the half domain, the boundary conditions changes from homogeneous Neumann to homogeneous Dirichlet, while the half domain is convex.

The problem can be implemented as the example of the previous sections. In order to put Dirichlet condition on the boundary domains `"crack"` and `"envelop"`, we block the related degrees of freedom:

```
Vh.block ("envelop");
Vh.block ("crack");
```

while degrees of freedom on the `"symmetry"` domain, associated to homogeneous Neumann condition, are unknown.

8.4.2 Implementation

```
// usage: crach <geo> [-diff|-exact]
```

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

Float alpha = 0.5; // the singularity at r=0

Float u_crack(const point& x)
{
    Float r = sqrt(x[0]*x[0] + x[1]*x[1]);
    if (1 + r == Float(1)) return 0;
    Float theta = atan2(x[1], x[0]);  // in ]-pi,pi]
    return pow(r,alpha)*sin(alpha*theta);
}
int main(int argc, char**argv)
{
    geo omega(argv[1]);
    space Vh (omega, argv[2]);
    Vh.block ("envelop");
    Vh.block ("crack");
    form a(Vh, Vh,"grad_grad");
    ssk<Float> fact = ldlt(a.uu);
    field u = interpolate(Vh, u_crack);
    field uh (Vh);
    uh.b = u.b;
    uh.u = fact.solve(- (a.ub*uh.b));
    field eh = u - uh;
    form m(Vh, Vh, "mass");
    cerr << "error_l2       " << sqrt(m(eh, eh))  << endl;
    cerr << "error_infinity " << eh.max_abs()  << endl;

    // H1 error
    string approx_grad = (strcmp(argv[2],"P1") == 0) ? "P0" : "P1d";
    space Th (omega, approx_grad);
    form d_dx0(Vh, Th, "d_dx0");
    form d_dx1(Vh, Th, "d_dx1");
    form inv_mt (Th, Th, "inv_mass");
    field deh_dx0 = inv_mt*(d_dx0*eh);
    field deh_dx1 = inv_mt*(d_dx1*eh);
    form mt (Th, Th, "mass");
    cerr << "error_h1       " << sqrt(mt(deh_dx0, deh_dx0)
                                     +mt(deh_dx1, deh_dx1))
         << endl;

    int digits10 = numeric_limits<Float>::digits10;
    cout << setprecision(digits10);
    if (argc == 2) {
        cout << uh;
    } else {
        if (strcmp(argv[2],"-exact") == 0) { cout << u; }
        else if (strcmp(argv[2],"-diff") == 0) { cout << u-uh; }
        else { cout << uh; }
    }
    return 0;
}
```

8.4.3 Analysis

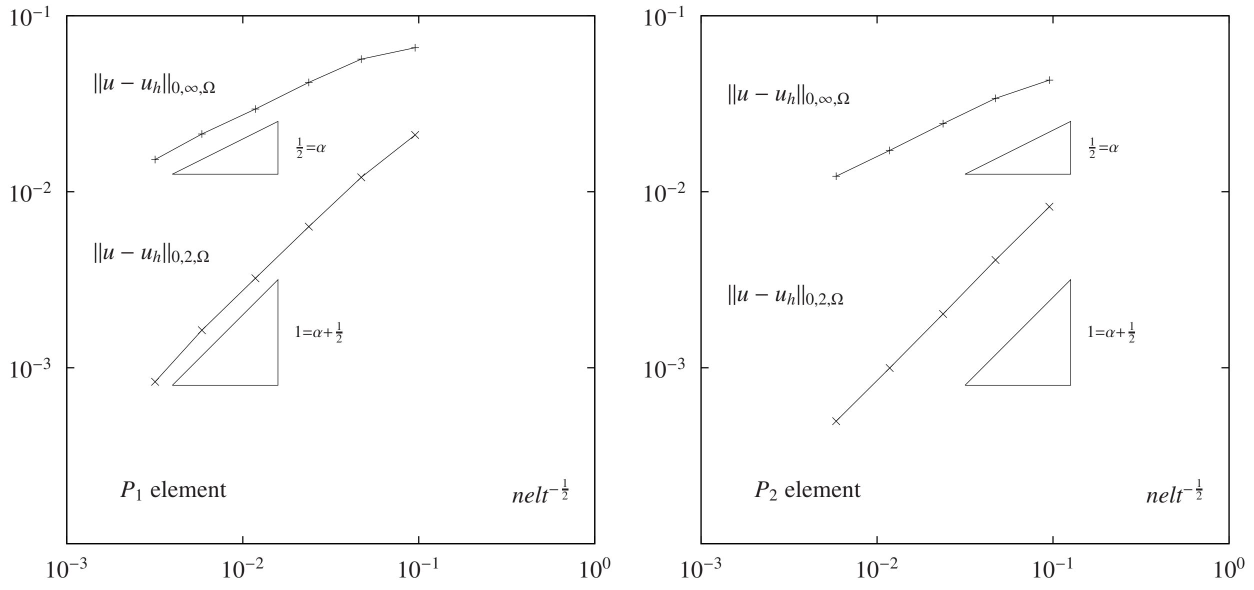


Figure 8.4: Error analysis for a domain with a crack.

The figure plots the error in L^2 and L^∞ norms for both P_1 (on the left) and P_2 (on the right) by using a familly of unifom meshes. First, remarks that the order of convergence is not improved by using P_2 elements. This case is different from the situation when the solution was regular. More precisely, the following error estimates holds:

$$\|u-u_h\|_{0,2,\Omega} \approx \mathcal{O}(h^{\min(\alpha+\frac{1}{2},k+1)})$$

$$\|u-u_h\|_{0,\infty,\Omega} \approx O(h^{\min(\alpha,k+1)})$$

The finite element mesh adaptation is an efficient way to restaure the optimal convergence rate versus the mesh size, as we will see in the next paragraph.

Bibliography

[1] S. Brenner and R. Scott. *The mathematical theory of finite element methods.* Springer-Verlag, 1991. 7, 81

[2] P. Saramito and N. Roquet. An adaptive finite element method for viscoplastic fluid flows in pipes. *Comput. Meth. Appl. Mech. Engrg.,* (to appear), 2000. 14

[3] V. Girault and P. A. Raviart. *Finite Element Methods for Navier-Stokes Equations—Theory and Algorithms.* Springer Verlag, 1986. 23, 44, 46

[4] C. C. Paige and M. A. Saunders. Solution of sparse indefinite systems of linear equations. *SIAM J. Numer. Anal.*, 12(4):617–629, 1975. http://www.stanford.edu/group/SOL/software.html. 23

[5] Y. Saad. *Iterative methods for sparse linear systems.* SIAM, Philadelphia, USA, second edition, 2000. 25

[6] N. Roquet, R. Michel, and P. Saramito. Estimations d'erreur pour un fluide viscoplastique par éléments finis P_k et maillages adaptés. *C. R. Acad. Sci. Paris, série I, to appear.* 37

[7] N. Roquet. *Simulation numérique d'écoulements de fluides viscoplastiques par un algorithme de Lagrangien augmenté et une méthode d'éléments finis incompressibles.* Université d'Orsay Paris-Sud, Prébublication 97-21, 1997. 37

[8] F. Hecht. *Bidimensional anisotropic mesh generator, version 0.67.* INRIA, october 1998. `ftp://ftp.inria.fr/INRIA/Projects/Gamma/bamg/`. 37, 109

[9] H. Borouchaki, P. L. George, F. Hecht, P. Laug, and E. Saltel. Delaunay mesh generation governed by metric specifications. Part I: Algorithms. *Finite Elem. Anal. Des.*, 25:61–83, 1997. 37

[10] M. J. Castro-Diaz, F. Hecht, B. Mohammadi, and O. Pironneau. Anisotropic unstructured mesh adaption for flow simulations. *Int. J. Numer. Methods Fluids*, 25:475–491, 1997. 37

[11] M. G. Vallet. *Génération de maillages anisotropes adaptés. Application à la capture de couches limites.* Rapport de Recherche n^0 1360, INRIA, 1990. 37

[12] 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. 40, 50, 65

[13] 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. 42, 51

[14] D. N. Arnold, F. Brezzi, and M. Fortin. A stable finite element for the Stokes equations. 1983. 52

[15] E. M. Abdalass. *Résolution performante du problème de Stokes par mini-éléments, maillages auto-adaptatifs et méthodes multigrilles - applications.* Thѐse de l'École Centrale de Lyon, 1987. 54

[16] 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. 54

[17] O. Pironneau. *Méthode des éléments finis pour les fluides.* Masson, 1988. 61

[18] H. Rui and M. Tabata. A second order characteristic finite element scheme for convection diffusion problems. *Numer. Math. (*to appear*)*, 2001. 61

[19] D. R. Musser and A. Saini. *STL tutorial and reference guide.* Addison-Wesley, 1996. 62

[20] 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. 64

[21] 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. 64

[22] Cahouet and J.-P. Chabard. Some fast 3d finite element solvers for the generalized Stokes problem. *Int. J. Numer. Meth. Fluids*, 8(8):869–895, 1988. 66

[23] G. M. Kobelkov and M. A. Olshanskii. Effective preconditioning of Uzawa type schemes for a generalized Stokes problem. *Numer. Math.*, 86:443–470, 2000. http://www.mathcs.emory.edu/ molshan/ftp/pub/gstokes.pdf. 66

[24] 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. 71, 127

[25] 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. http://www.cours.polymtl.ca/mec6616/cavite_2.pdf. 71

[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. 71

[27] J. Donea and A. Huerta. *Finite element methods for flow problems.* Wiley, New-York, 2003. 71

[28] 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. 72

[29] 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. 72

[30] 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. 72

[31] 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. 72

[32] 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. 72

[33] H. Brezis. *Analyse fonctionnelle. Théorie et application.* Masson, Paris, 1983. 81

[34] Jr. J. E. Dennis and R. B. Schnablel. *Numerical methods for unconstraint optimization and nonlinear equations.* Prentice Hall, Englewood Cliff, N. J., 1983. 91

[35] 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. 91

[36] P. Grisvard. Problèmes aux limites dans les polygônes. mode d'emploi. Buletin de la direction des études et recherches, série C, EDF, 1986. 99

[37] 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. 109, 112

Part IV

Technical appendices

Appendix A

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 [8], and the `gmsh` bi- and tridimensional mesh generator, written by Christophe Geuzaine and Jean-Franois Remacle [37].

A.1 Bidimensionnal mesh with `bamg`

We first create a '`my-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 [8] for the complete file format description. Next, enter the mesh generator commands:

```
bamg -g my-square.bamgcad -o my-square.bamg
```

Then, create the file '`my-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 my-square.bamg my-square.dmn > my-square.geo
```

This command creates a 'my-square.geo' file. Look at the mesh via the command:

```
geo my-square
```

This presents the mesh it in a graphical form, usually with mayavi. You can switch to the gnuplot renders:

```
geo my-square -gnuplot
```

A finer mesh could be generated by:

```
bamg -coef 0.5 -g my-square.bamgcad -o my-square-0.5.bamg
```

One inconvenience of bamg is that it does not provide multi-region support. However, if the two regions desired in a mesh are separated with a boundary named, e.g., interface, and that there exists another boundary domain in the mesh which is away from one of the regions, say top, rheolef can label the two regions and create two domains for them:

```
omega.build_subregion(omega["top"],omega["interface"],"north","south");
```

where region north will have domain top in its boundary, while region south is away from it, and separated from north by interface. To do this, the file 'my-square.dmn' can be modified to have 5 domain names, appending the name interface, and the file 'my-square.bamgcad' can be modified in this way:

The domains omega["north"] and omega["south"] can be used in the same way as is done in section 6.

A.2 Bidimensionnal mesh with `gmsh`

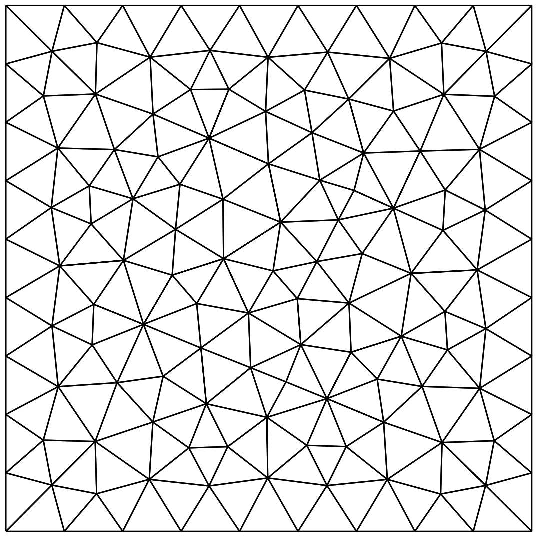
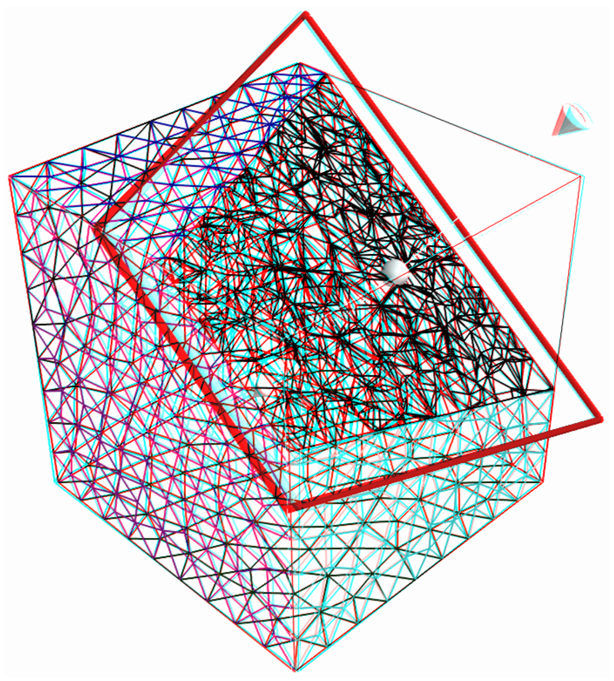



Figure A.1: Visualization of the `gmsh` meshes '`my-square.geo`' and '`my-cube.geo`'.

We first create a '`my-square.mshcad`' file:

```
Mesh.Algorithm = 5;
Mesh.Optimize = 1;
// Mesh.OptimizeNetgen = 1; // uncomment if gmsh is compiled with netgen
h_local = 0.1;
Point(1) = {0, 0, 0, h_local};
Point(2) = {1, 0, 0, h_local};
Point(3) = {1, 1, 0, h_local};
Point(4) = {0, 1, 0, h_local};
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("bottom") = {1};
Physical Line("right")  = {2};
Physical Line("top")    = {3};
Physical Line("left")   = {4};
Physical Line("boundary") = {1,2,3,4};
Physical Surface("interior") = {6};
```

This is an uniform mesh with element size $h = 0.1$. We refer to the `gmsh` documentation [37] for the complete file format description. Next, enter the mesh generator commands:

```
gmsh -2 my-square.mshcad -o my-square.msh
```

Then, enter the translation command:

```
msh2geo my-square.msh > my-square.geo
```

This command creates a '`my-square.geo`' file. Look at the mesh via the command:

```
geo my-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.

A.3 Tridimensionnal mesh with `gmsh`

First, create a '`my-cube.mshcad`' file:

```
h_local = 0.1;
Point(1) = {0, 0, 0, h_local};
Point(2) = {1, 0, 0, h_local};
Point(3) = {1, 1, 0, h_local};
Point(4) = {0, 1, 0, h_local};
Point(5) = {0, 0, 1, h_local};
Point(6) = {1, 0, 1, h_local};
Point(7) = {1, 1, 1, h_local};
Point(8) = {0, 1, 1, h_local};
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} ;
Physical Surface("bottom") = {31};
Line Loop(22) = {5,6,7,8};
Plane Surface(32) = {22} ;
Physical Surface("top") = {32};
Line Loop(23) = {1,10,-5,-9};
Plane Surface(33) = {23} ;
Physical Surface("left") = {33};
Line Loop(24) = {2,11,-6,-10};
```

```
    Plane Surface(34) = {24} ;
    Physical Surface("front") = {34};
    Line Loop(25) = {12,-7,-11,3};
    Plane Surface(35) = {25} ;
    Physical Surface("right") = {35};
    Line Loop(26) = {9,-8,-12,4};
    Plane Surface(36) = {26} ;
    Physical Surface("back") = {36};
    Surface Loop(41) = {31,32,33,34,35,36};
    Volume(51) = {41};
    Physical Volume("interior") = {51};
```

Next, enter the mesh generator commands:

```
        gmsh -3 my-cube.mshcad -o my-cube.msh
```

Then, enter the translation command:

```
        msh2geo my-cube.msh > my-cube.geo
```

This command creates a ‘`my-cube.geo`’ file. Look at the mesh via the command:

```
        geo my-cube
        geo my-cube.geo -cut -normal 0 0 1 -origin 0.5 0.5 0.5
```

The second command allows to see inside the mesh.

Appendix B

GNU Free Documentation License

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.

B.1 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

B.2 Verbatim Copying

B.3 Copying in Quantity

B.4 Modifications

B.5 Combining Documents

B.6 Collections of Documents

B.7 Aggregation With Independent Works

B.8 Translation

B.9 Termination

B.10 Future Revisions of This License

ADDENDUM: How to use this License for your documents

Index of concepts

Index of file formats

Index of example files

Index of programs

List of Figures