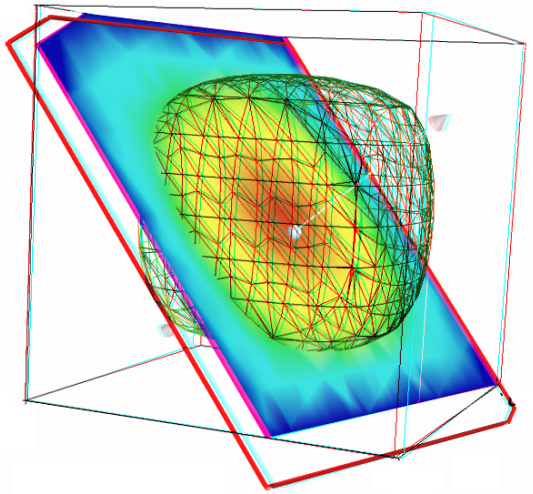
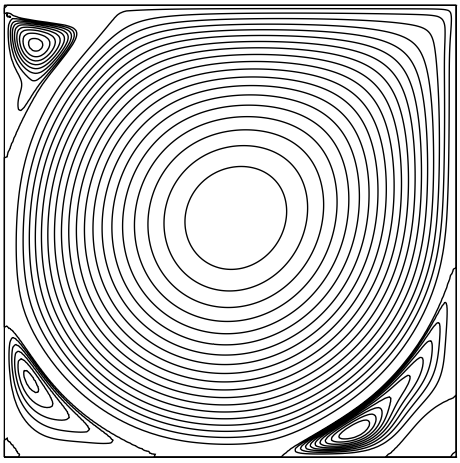
Efficient C++ finite element computing with Rheolef

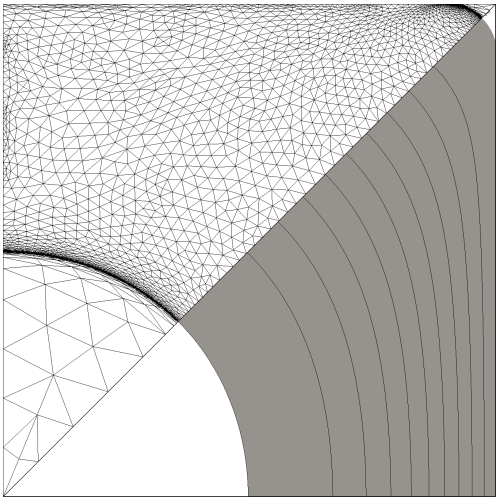
Pierre Saramito

version 7.2

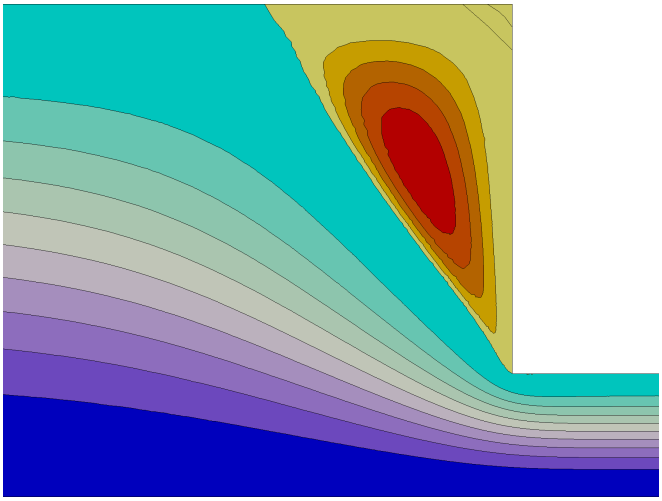




$Re = 10\,000$



$Bi = 0.5$



$We = 0.7$

Copyright (c) 2003-2018 Pierre Saramito

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

Introduction

Rheolef is a programming environment for finite element method computing. The reader is assumed to be familiar with (i) the c++ programming language and (ii) the finite element method. As a Lego game, the Rheolef bricks allow the user to solve most problems, from simple to complex multi-physics ones, in few lines of code. The concision and readability of codes written with Rheolef is certainly a major keypoint of this environment. Here is an example of a Rheolef code for solving the Poisson problem with homogeneous boundary conditions:

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

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

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

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

$V_h = X_h \cap H_0^1(\Omega)$

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

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

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

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

The right column shows the one-to-one line **correspondence between the code and the variational formulation**. Let us quote Stroustrup [2002], the conceptor of the c++ language:

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

Rheolef is an attempt to provide such a library in the field of finite element methods for partial differential equations. Rheolef provides both a c++ library and a set of unix commands for **shell** programming, providing data structures and algorithms [Wirth, 1985].

- Data structures fit the variational formulation concept: **field**, bilinear **form** and functional space, are c++ types for variables. They can be combined in algebraic expressions, as you write it on the paper.
- Algorithms refer to the most up-to-date ones: direct an iterative sparse matrix solvers for linear systems. They supports efficient distributed memory and parallel computations. Nonlinear c++ generic algorithms such as fixed point, damped Newton and continuation methods are also provided.

General *high order* piecewise polynomial finite element approximations are implemented, together with some mixed combinations for Stokes and incompressible elasticity. The *characteristic method* can be used for diffusion-convection problems while hyperbolic systems can be discretized by the discontinuous Galerkin method.

Contacts

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

Please send all patches, comments and bug reports by mail to

rheolef@grenet.fr

Contents

Notations **9**

1 Getting started **13**
1.1 The model problem . . . 13
1.1.1 Problem statement . . . 14
1.1.2 Approximation . . . 14
1.1.3 Comments . . . 15
1.1.4 How to compile the code . . . 16
1.1.5 How to run the program . . . 17
1.1.6 Advanced and stereo visualization . . . 17
1.1.7 High-order finite element methods . . . 19
1.1.8 Tridimensional computations . . . 19
1.1.9 Quadrangles, prisms and hexahedra . . . 20
1.1.10 Direct versus iterative solvers . . . 20
1.1.11 Distributed and parallel runs . . . 22
1.1.12 Non-homogeneous Dirichlet conditions . . . 24
1.2 Non-homogeneous Neumann boundary conditions for the Helmholtz operator . . . 32
1.3 The Robin boundary conditions . . . 34
1.4 Neumann boundary conditions for the Laplace operator . . . 35
1.5 Non-constant coefficients and multi-regions . . . 37

2 Fluids and solids computations **43**
2.1 The linear elasticity and the Stokes problems . . . 43
2.1.1 The linear elasticity problem . . . 43
2.1.2 Computing the stress tensor . . . 47
2.1.3 Mesh adaptation . . . 50
2.1.4 The Stokes problem . . . 53
2.1.5 Computing the vorticity . . . 57
2.1.6 Computing the stream function . . . 58
2.2 Nearly incompressible elasticity and the stabilized Stokes problems . . . 60
2.2.1 The incompressible elasticity problem . . . 60
2.2.2 The $P_1b - P_1$ element for the Stokes problem . . . 62
2.2.3 Axisymmetric geometries . . . 69
2.2.4 The axisymmetric stream function and stress tensor . . . 69
2.3 [New] Slip boundary conditions . . . 72

2.4 Time-dependent problems . . . 76
2.4.1 The heat equation . . . 76
2.4.2 The convection-diffusion problem . . . 79
2.5 The Navier-Stokes equations . . . 84

3 Advanced and highly nonlinear problems 93
3.1 Equation defined on a surface . . . 93
3.1.1 Approximation on an explicit surface mesh . . . 94
The Helmholtz-Beltrami problem . . . 94
The Laplace-Beltrami problem . . . 98
3.1.2 Building a surface mesh from a level set function . . . 101
3.1.3 The banded level set method . . . 104
3.1.4 Improving the banded level set method with a direct solver . . . 106
3.2 The highly nonlinear p-laplacian problem . . . 111
3.2.1 Problem statement . . . 111
3.2.2 The fixed-point algorithm . . . 111
3.2.3 The Newton algorithm . . . 119
3.2.4 The damped Newton algorithm . . . 125
3.2.5 Error analysis . . . 128
3.3 Continuation and bifurcation methods . . . 130
3.3.1 Problem statement and the Newton method . . . 131
3.3.2 Error analysis and multiplicity of solutions . . . 134
3.3.3 The Euler-Newton continuation algorithm . . . 137
3.3.4 Beyond the limit point : the Keller algorithm . . . 140

4 Discontinuous Galerkin methods 147
4.1 Linear first-order problems . . . 147
4.1.1 The stationary transport equation . . . 147
4.1.2 [New] The time-dependent transport equation . . . 150
4.1.3 [New] Example: the Zalesak slotted disk . . . 152
4.1.4 [New] Example: the Leveque vortex-in-box . . . 154
4.2 Nonlinear first-order problems . . . 157
4.2.1 Abstract setting . . . 157
4.2.2 Slope limiters . . . 158
4.2.3 Example: the Burgers equation . . . 161
4.3 Scalar second-order problems . . . 167
4.3.1 The Poisson problem with Dirichlet boundary conditions . . . 167
4.3.2 The Helmholtz problem with Neumann boundary conditions . . . 170
4.3.3 [New] Heterogeneous diffusion . . . 171
4.3.4 Nonlinear scalar hyperbolic problems with diffusion . . . 175
4.3.5 Example: the Burgers equation with diffusion . . . 175
4.4 Fluids and solids computations revisited . . . 181
4.4.1 The linear elasticity problem . . . 181
4.4.2 The Stokes problem . . . 183
4.5 The stationnary Navier-Stokes equations . . . 185

4.5.1 Problem statemment . . . 185
4.5.2 The discrete problem . . . 186
4.5.3 A conservative variant . . . 189
4.5.4 Newton solver . . . 192
4.5.5 Application to the driven cavity benchmark . . . 195
4.5.6 Upwinding . . . 196

5 Complex fluids 203
5.1 Yield slip at the wall . . . 203
5.1.1 Problem statement . . . 203
5.1.2 The augmented Lagrangian algorithm . . . 204
5.1.3 Newton algorithm . . . 210
5.1.4 Error analysis . . . 214
5.2 Viscoplastic fluids . . . 217
5.2.1 Problem statement . . . 217
5.2.2 The augmented Lagrangian algorithm . . . 217
5.2.3 Mesh adaptation . . . 222
5.2.4 Error analysis . . . 224
5.2.5 Error analysis for the yield surface . . . 225
5.3 Viscoelastic fluids . . . 230
5.3.1 A tensor transport equation . . . 230
5.3.2 The Oldroyd model . . . 233
5.3.3 The θ-scheme algorithm . . . 234
5.3.4 Flow in an abrupt ontraction . . . 239
5.3.5 [New] Stress diffusion . . . 246

6 [New] Hybrid discontinuous methods 249
6.1 The Raviart-Thomas element . . . 249
6.2 Hybrid discontinuous Galerkin (HDG) methods . . . 252
6.2.1 The Poisson problem . . . 252
6.2.2 Superconvergence of the Lagrange multiplier . . . 257
6.2.3 Superconvergence of the piecewise averaged solution . . . 258
6.2.4 Improving the solution by local postprocessing . . . 259
6.2.5 Improving the gradient with the Raviart-Thomas element . . . 262
6.3 Hybrid high order (HHO) methods . . . 265
6.3.1 The diffusion problem . . . 265
6.3.2 The reconstruction operator . . . 265
6.3.3 The projections . . . 269
6.3.4 The discrete problem statement . . . 270

A Technical appendices 279
A.1 How to write a variational formulation ? . . . 279
A.1.1 The Green formula . . . 279
A.1.2 The vectorial Green formula . . . 279
A.1.3 The Green formula on a surface . . . 280

A.2 How to prepare a mesh ? 280
A.2.1 Bidimensional mesh with `bamg` 281
A.2.2 Unidimensional mesh with `gmsh` 282
A.2.3 Bidimensional mesh with `gmsh` 282
A.2.4 Tridimensional mesh with `gmsh` 283

B GNU Free Documentation License **285**

List of example files **298**

List of commands **301**

Index **303**

Notations

Rheolef	mathematics	description
`d`	$d \in \{1,2,3\}$	dimension of the physical space
`interpolate(Vh,`*expr*`)`	$\pi_{V_h}(expr)$	interpolation in the space V_h
`integrate(omega,`*expr*`)`	$\int_\Omega expr \, \mathrm{d}x$	integration in $\Omega \subset \mathbb{R}^d$
`integrate(`omega, `on_local_sides(`*expr*`))`	$\sum_{K \in \mathscr{T}_h} \int_{\partial K} expr \, \mathrm{d}s$	integration on the local element sides
`dot(u,v)`	$\mathbf{u}.\mathbf{v} = \sum_{i=0}^{d-1} u_i v_i$	vector scalar product
`ddot(sigma,tau)`	$\sigma : \tau = \sum_{i,j=0}^{d-1} \sigma_{i,j} \tau_{i,j}$	tensor scalar product
`dddot(A,B)`	$A \vdots B = \sum_{i,j,k=0}^{d-1} A_{i,j,k} B_{i,j,k}$	tensor3 scalar product
`tr(sigma)`	$\mathrm{tr}(\sigma) = \sum_{i=0}^{d-1} \sigma_{i,i}$	trace of a tensor
`trans(sigma)`	σ^T	tensor transposition
`sqr(phi)` `norm2(phi)`	ϕ^2	square of a scalar
`norm2(u)`	$\lvert\mathbf{u}\rvert^2 = \sum_{i=0}^{d-1} u_i^2$	square of the vector norm
`norm2(sigma)`	$\lvert\sigma\rvert^2 = \sum_{i,j=0}^{d-1} \sigma_{i,j}^2$	square of the tensor norm
`abs(phi)` `norm(phi)`	$\lvert\phi\rvert$	absolute value of a scalar
`norm(u)`	$\lvert\mathbf{u}\rvert = \left(\sum_{i=0}^{d-1} u_i^2\right)^{1/2}$	vector norm
`norm(sigma)`	$\lvert\sigma\rvert = \left(\sum_{i,j=0}^{d-1} \sigma_{i,j}^2\right)^{1/2}$	tensor norm
`grad(phi)`	$\nabla\phi = \left(\frac{\partial\phi}{\partial x_i}\right)_{0 \leqslant i < d}$	gradient of a scalar field
`grad(u)`	$\nabla\mathbf{u} = \left(\frac{\partial u_i}{\partial x_j}\right)_{0 \leqslant i,j < d}$	gradient of a vector field
`div(u)`	$\mathrm{div}(\mathbf{u}) = \mathrm{tr}(\nabla\mathbf{u}) = \sum_{i=0}^{d-1} \frac{\partial u_i}{\partial x_i}$	divergence of a vector field
`D(u)`	$D(\mathbf{u}) = \left(\nabla\mathbf{u} + \nabla\mathbf{u}^T\right)/2$	symmetric part of the gradient of a vector field

Rheolef	mathematics	description
`curl(u)`	$\mathbf{curl}(\mathbf{u}) = \nabla \wedge \mathbf{u}$	curl of a vector field, when $d = 3$
`curl(phi)`	$\mathbf{curl}(\phi) = \left(\frac{\partial \phi}{\partial x_1}, -\frac{\partial \phi}{\partial x_0}\right)$	curl of a scalar field, when $d = 2$
`curl(u)`	$\mathrm{curl}(\mathbf{u}) = \frac{\partial u_1}{\partial x_0} - \frac{\partial u_0}{\partial x_1}$	curl of a vector field, when $d = 2$
`grad_s(phi)`	$\nabla_s \phi = P \nabla \phi$ where $P = I - \mathbf{n} \otimes \mathbf{n}$	tangential gradient of a scalar
`grad_s(u)`	$\nabla_s \mathbf{u} = \nabla \mathbf{u} P$	tangential gradient of a vector
`Ds(u)`	$D_s(\mathbf{u}) = P D(\mathbf{u}) P$	symmetrized tangential gradient
`div_s(u)`	$\mathrm{div}_s(\mathbf{u}) = \mathrm{tr}(D_s(\mathbf{u}))$	tangential divergence
`normal()`	$\mathbf{n}$	unit outward normal on $\Gamma = \partial\Omega$ or on an oriented surface Ω or on an internal oriented side S
`jump(phi)`	$[\![\phi]\!] = \begin{cases} \phi_{\vert K_0} - \phi_{\vert K_1} \\ \phi \ \text{ on } \partial\Omega \end{cases}$	jump accross side $S = \partial K_0 \cap K_1$ see section 4.1.1 page 147
`average(phi)`	$\{\!\{\phi\}\!\} = \begin{cases} (\phi_{\vert K_0} + \phi_{\vert K_1})/2 \\ \phi \ \text{ on } \partial\Omega \end{cases}$	average across $S = \partial K_0 \cap K_1$
`inner(phi)`	$= \begin{cases} \phi_{\vert K_0} \\ \phi \ \text{ on } \partial\Omega \end{cases}$	inner trace across $S = \partial K_0 \cap K_1$
`outer(phi)`	$= \begin{cases} \phi_{\vert K_1} \\ \phi \ \text{ on } \partial\Omega \end{cases}$	outer trace across $S = \partial K_0 \cap K_1$
`h_local()`	$h_K = \mathrm{meas}(K)^{1/d}$	length scale on an element K
`penalty()`	$\varpi_s = \max\left(\frac{\mathrm{meas}(\partial K_0)}{\mathrm{meas}(K_0)}, \frac{\mathrm{meas}(\partial K_1)}{\mathrm{meas}(K_1)}\right)$	penalty coefficient on S
`grad_h(phi)`	$(\nabla_h \phi)_{\vert K} = \nabla(\phi_{\vert K}), \forall K \in \mathcal{T}_h$	broken gradient
`div_h(u)`	$(\mathrm{div}_h \mathbf{u})_{\vert K} = \mathrm{div}(\mathbf{u}_{\vert K}), \forall K \in \mathcal{T}_h$	broken divergence of a vector field
`Dh(u)`	$(D_h(\mathbf{u}))_{\vert K} = D(\mathbf{u}_{\vert K}), \forall K \in \mathcal{T}_h$	broken symmetric part of

Rheolef	mathematics	description
		the gradient of a vector field
`sin(phi)`	$\sin(\phi)$	standard mathematical functions
`cos(phi)`	$\cos(\phi)$	extended to scalar fields
`tan(phi)`	$\tan(\phi)$	
`acos(phi)`	$\cos^{-1}(\phi)$	
`asin(phi)`	$\sin^{-1}(\phi)$	
`atan(phi)`	$\tan^{-1}(\phi)$	
`cosh(phi)`	$\cosh(\phi)$	
`sinh(phi)`	$\sinh(\phi)$	
`tanh(phi)`	$\tanh(\phi)$	
`exp(phi)`	$\exp(\phi)$	
`log(phi)`	$\log(\phi)$	
`log10(phi)`	$\log 10(\phi)$	
`floor(phi)`	$\lfloor\phi\rfloor$	largest integral not greater than ϕ
`ceil(phi)`	$\lceil\phi\rceil$	smallest integral not less than ϕ
`min(phi,psi)`	$\min(\phi,\psi)$	
`max(phi,psi)`	$\max(\phi,\psi)$	
`pow(phi,psi)`	ϕ^ψ	
`atan2(phi,psi)`	$\tan^{-1}(\psi/\phi)$	
`fmod(phi,psi)`	$\phi - \lfloor\phi/\psi + 1/2\rfloor\,\psi$	floating point remainder
`compose(f,phi)`	$f \circ \phi = f(\phi)$	applies an unary function f
`compose(f,phi1,...,phin)`	$f(\phi_1,\ldots,\phi_n)$	applies a n-ary function f, $n \geqslant 1$
`compose(phi,X)`	$\phi \circ X,\ \ X(x) = x + \mathbf{d}(x)$	composition with a characteristic

Chapter 1

Getting started

The first chapter 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 the model problem. The last two examples presents some special difficulties that appears in most problems: the first one introduce to problems with non-constant coefficients and the second one, a ill-posed problem where the solution is defined up to a constant.

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

1.1 The model problem

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

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

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

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

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

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

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

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

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

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

1.1.1 Problem statement

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

$$-\Delta u = 1 \text{ in } \Omega \tag{1.1}$$
$$u = 0 \text{ on } \partial\Omega \tag{1.2}$$

where Δ denotes the Laplace operator. The variational formulation of this problem expresses (see appendix A.1.1 for details):
(VF): find $u \in H_0^1(\Omega)$ such that:

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

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

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

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

1.1.2 Approximation

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

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

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

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

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

File 1.1: `dirichlet.cc`

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

1.1.3 Comments

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

Let us now comment the code, line by line.

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

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

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

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

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

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

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

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

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

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

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

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

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

The homogeneous Dirichlet conditions are declared on the boundary.

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

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

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

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

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

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

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

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

```
problem p (a);
p.solve (lh, uh);
```

Finally, the problem related to the bilinear form a and the right-hand-side `lh` is solved and `uh` contains the solution. The field is printed to standard output:

```
dout << uh;
```

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

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

$$u_h = \sum_i u_i \varphi_i$$

Thus, the field u_h is completely represented by its coefficients (u_i). The coefficients (u_i) of this combination are grouped into to sets: some have zero values, from the boundary condition and are related to *blocked* coefficients, and some others are *unknown*. Blocked coefficients are stored into the `uh.b` array while unknown one are stored into `uh.u`. Thus, the restriction of the bilinear form $a(.,.)$ to $X_h \times X_h$ can be conveniently represented by a block-matrix structure:

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

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

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

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

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

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

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

The resolution of this linear system for the `a.uu` matrix is then performed via the `solver` class: this call is performed by the `problem` class. For more details, see the Rheolef reference manual related to the `problem` and `solver` classes, available on the web site and via the unix command:

```
man problem
man solver
```

1.1.4 How to compile the code

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

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

Then, enter:

```
make dirichlet
```

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

1.1.5 How to run the program

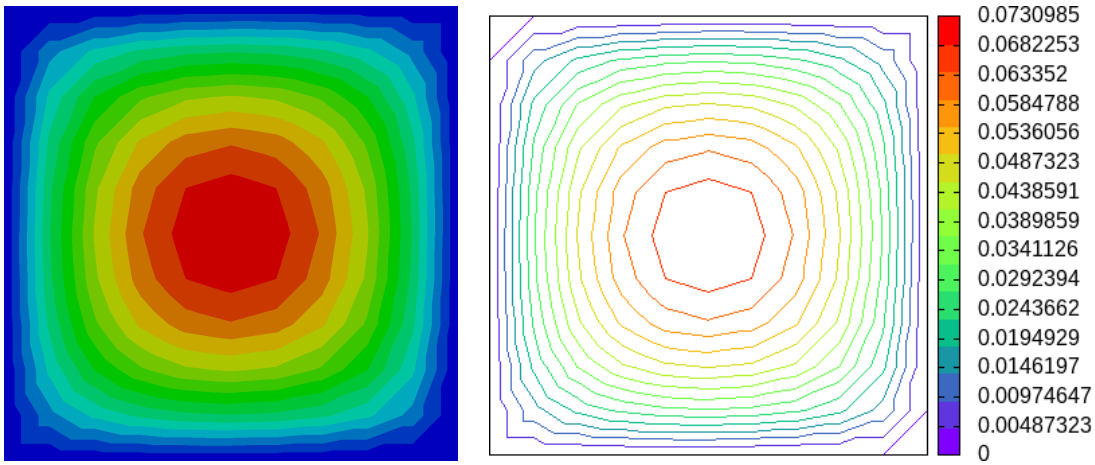


Figure 1.1: Solution of the model problem for $d = 2$ with the P_1 element: visualization (left) with `paraview` as filled isocontours ; (right) with `gnuplot` as unfilled isocontours.

Enter the commands:

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

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

The next commands perform the computation and visualization:

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

The result is hown on Fig. 1.1. By default, the visualization appears in a `paraview` window. If you are in trouble with this software, you can switch to the simpler `gnuplot` visualization mode:

```
field square.field -gnuplot
```

1.1.6 Advanced and stereo visualization

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

```
field square.field -bw
field square.field -gray
field square.field -elevation
field square.field -elevation -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. In the book, stereo figures are indicated by the  logo in the right margin. See http://en.wikipedia.org/wiki/Anaglyph_image for more and https://

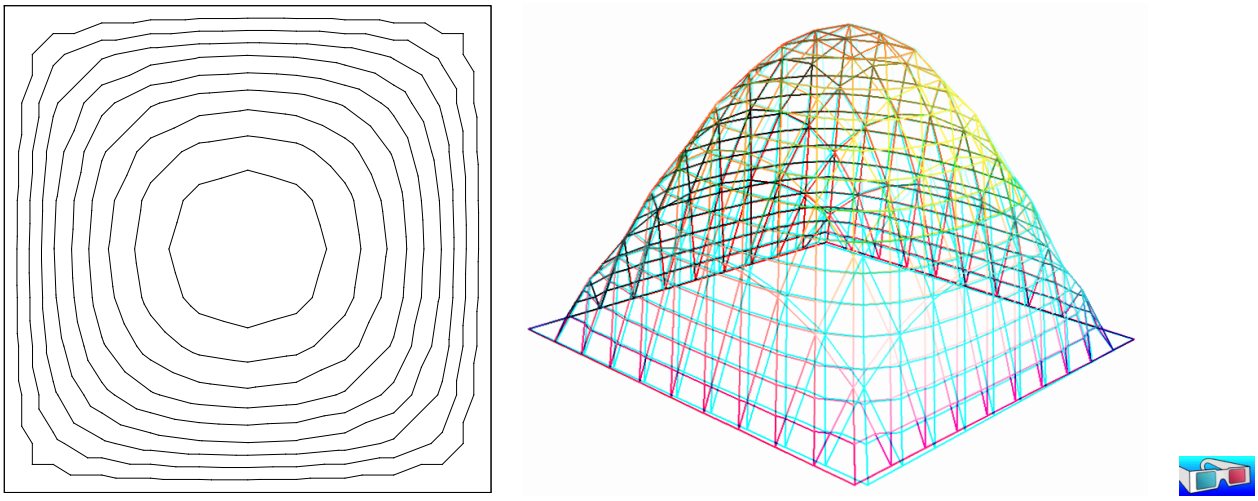


Figure 1.2: Alternative representations of the solution of the model problem ($d = 2$ and the P_1 element): (left) in black-and-white; (right) in elevation and stereoscopic anaglyph mode.

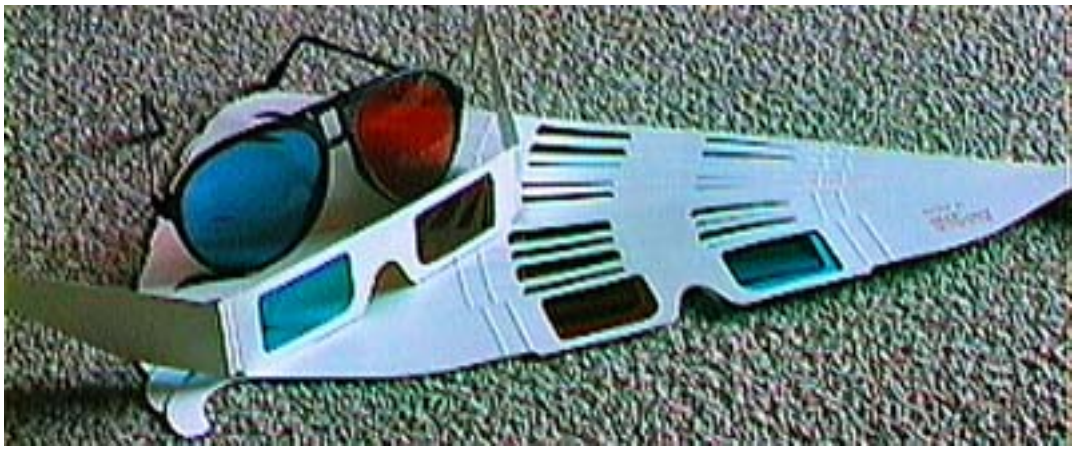


Figure 1.3: Red-cyan anaglyph glasses for the stereoscopic visualization.

www.ecolofrance.com/56-lunettes-anaglyphe-rouge-cyan-carton-pas-cher for how to find anaglyph red-cyan glasses. For simplicity, it would perhaps prefer to switch to the `gnuplot` render:

```
field square.field -gnuplot
field square.field -gnuplot -bw
field square.field -gnuplot -gray
```

Please, consult the **Rheolef** reference manual page for more on the unix commands `field`, `geo` and `mkgeo_grid`. The manual is available both on the web site:

```
firefox https://www-ljk.imag.fr/membres/Pierre.Saramito/rheolef
```

and as individual unix man pages:

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

See also

```
man rheolef
```

The complete list of **Rheolef** man page is obtained by:

```
man -k rheolef
```

1.1.7 High-order finite element methods

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

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

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

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

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

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

1.1.8 Tridimensional computations

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

```
mkgeo_grid -T 10 > cube.geo
geo cube.geo
geo cube.geo -fill
geo cube.geo -cut
geo cube.geo -shrink
geo cube.geo -shrink -cut
```

The 3D visualization bases on the `paraview` render. These commands present some cuts (`-cut`) inside the internal mesh structure: a simple click on the central arrow draws the cut plane normal vector or its origin, while the red square allows a translation. The following command draws the mesh with all internal edges (`-full`), together with the stereoscopic anaglyph (`-stereo`):

```
geo cube.geo -stereo -full
```

Then, we perform the computation and the visualization:

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

The visualization presents an isosurface. Also here, you can interact with the cutting plane. On the `Properties` of the `paraview` window, select `Contour`, change the value of the isosurface and click on the green `Apply` button. Finally exit from the visualization and explore the stereoscopic anaglyph mode (see Fig. 1.4, right):

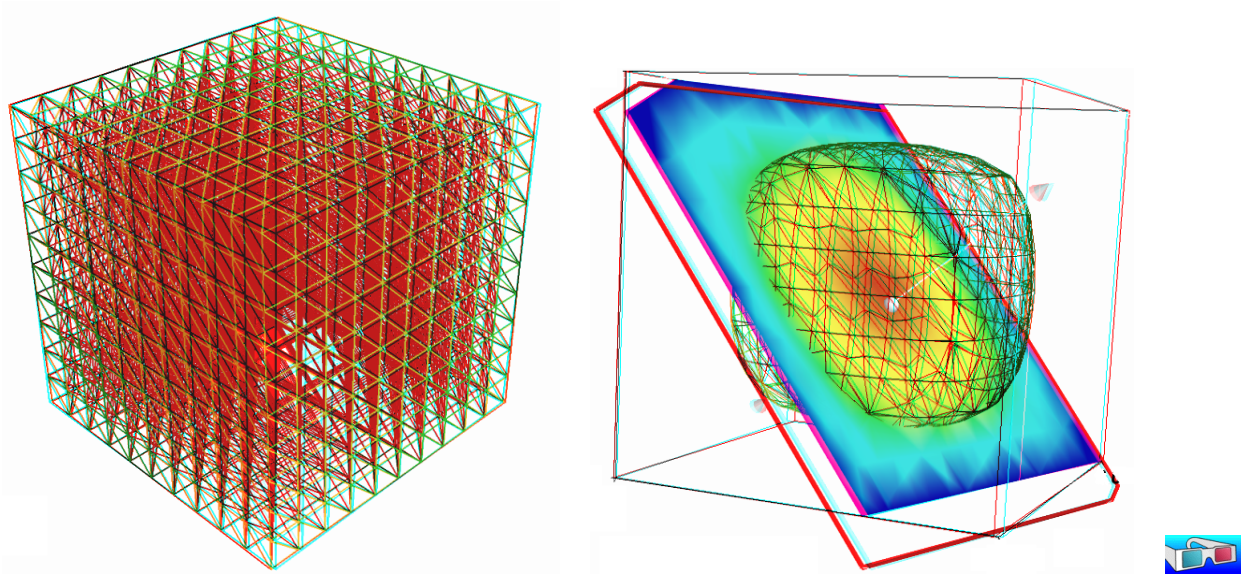


Figure 1.4: Solution of the model problem for $d = 3$ and the P_1 element : (left) mesh; (right) isovalue, cut planes and stereo anaglyph renderings.

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

It is also possible to add a second isosurface (`Contour`) or a cutting plane (`Slice`) to this scene by using the corresponding `Properties` menu. Finally, the following command, with the `-volume` option, allows a 3D color light volume graphical rendering:

```
field cube.field -volume
```

After this exploration of the 3D visualization capacities of our environment, let us go back to the Dirichlet problem and perform the `P2` approximation:

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

1.1.9 Quadrangles, prisms and hexahedra

Quadrangles and hexahedra are also supported in meshes:

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

Note also that the one-dimensional exact solution expresses explicitly:

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

In the two-and three dimensional cases, an explicit expression of the solution, based on Fourier expansion, is also known (see e.g. Saramito and Roquet, 2001, annex B, page 5411).

1.1.10 Direct versus iterative solvers

In order to measure the performances of the solver, the `dirichlet.cc` (page 14) has been modified as:

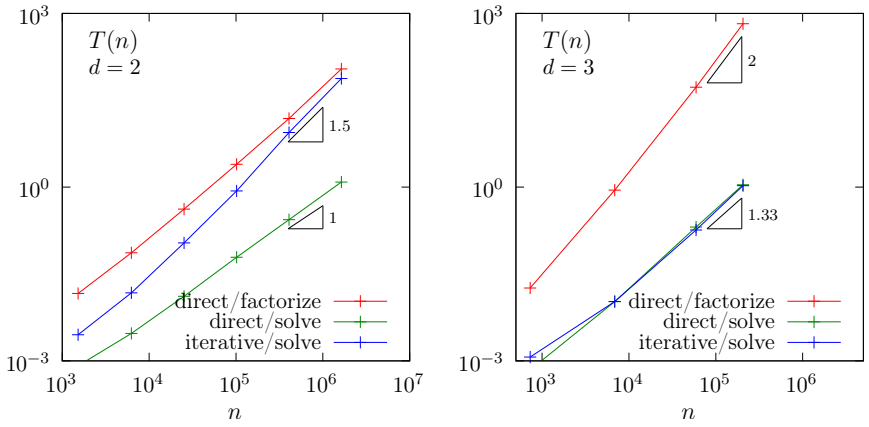


Figure 1.5: Compared performance between direct and iterative solvers: (left) $d = 2$; (right) $d = 3$.

```
double t0 = dis_wall_time();
solver_option sopt;
sopt.iterative = false; // or true
sopt.tol       = 1-5;   // when iterative
problem p (a, sopt);
double t_factorize = dis_wall_time() - t0;
p.solve (lh, uh);
double t_solve = dis_wall_time() - t0 - t_factorize;
derr << "time " << t_factorize << " " << t_solve << endl;
```

The `dis_wall_time` function returns the wall-clock time in seconds, while the `solver_option sopt` enables to choose between a direct or an iterative solver method: by default **Rheolef** selects a direct method when $d \leqslant 2$ and an iterative one when $d = 3$. For a 3D mesh, the compilation and run writes:

```
make dirichlet
mkgeo_grid -T 60 > cube-60.geo
./dirichlet cube-60.geo P1 > cube-60.field
```

Fig. 1.5 plots the performances of the direct and iterative solvers used in **Rheolef**. The computing time $T(n)$ is represented versus size n of the linear system, says $Ax = b$. Note that for a `square-`k`.geo` or `cube-`k`.geo` mesh, the size of the linear system is $n = (k-1)^d$. For the direct method, two times are represented: first, the time spend to *factorize* $A = LDL^T$, where L is lower triangular and D is diagonal, and second, the time used to *solve* $LDL^T = x$ (in three steps: solving $Lz = b$, then $Dy = z$ and finally $L^T x = y$). For the iterative method, the conjugate gradient algorithm is considered, without building any preconditioner, so there is nothing to initialize, and only one time is represented. The tolerance on the residual term is set to 10^{-5}.

In the bidimensional case, the iterative solver presents asymptotically, for large n, a computing time similar to the factorization time of the direct solver, roughly $\mathcal{O}(n^{3/2})$ while the time to solve by the direct method is dramatically lower, roughly $\mathcal{O}(n)$. As the factorization can be done one time for all, the direct method is advantageous most of the time.

In the three dimensional case, the situation is different. The factorization time is very time consuming roughly $\mathcal{O}(n^2)$, while the time to solve for both direct and iterative methods behave as $\mathcal{O}(n^{4/3})$. Thus, the iterative method is clearly advantageous for three-dimensionnal problems. Future works will improve the iterative approach by building preconditioners.

The asymptotic behaviors of direct methods strongly depends upon the ordering strategy used for the factorization. For the direct solver, **Rheolef** was configured with the `mumps` [Amestoy et al., 2001, 2006] library and `mumps` was configured with the parallel `scotch` [Pellegrini, 2010]

ordering library. For a regular grid and in the bidimensional case, there exists a specific ordering called nested disection [Hoffman et al., 1973, George, 1973] that minimize the fillin of the sparse matrix during the factorization. For three-dimensional case this ordering is called nested multisection [Ashcraft and Liu, 1998]. Asymptotic computing time for these regular grid are then explicity known versus the grid size n:

d	direct/factorize	direct/solve	iterative
1	n	n	n^2
2	$n^{3/2}$	$n \log n$	$n^{3/2}$
3	n^2	$n^{4/3}$	$n^{4/3}$

The last column gives the asymptotic computing time for the conjugate gradient on a general mesh [Saramito, 2013a]. Remark that these theoretical results are consistent with numerical experiments presented on Fig. 1.5. In conclusion, the best strategy is to choose a direct method when $d \leqslant 2$ and an iterative one when $d = 3$: this is the default behavior with **Rheolef**.

1.1.11 Distributed and parallel runs

For large meshes, a computation in a distributed and parallel environment is profitable:

```
mpirun -np  8 ./dirichlet cube-60.geo P1 > cube-60.field
mpirun -np 16 ./dirichlet cube-60.geo P1 > cube-60.field
```

The computing time $T(n,p)$ depends now upon the linear system size n and the number of processes p. For a fixed size n, the speedup $S(p)$ when using p processors is defined by the ratio of the time required by a sequential computation with the time used by a parallel one: $S(p) = T(n,1)/T(n,p)$. The speedup is presented on Fig 1.6 for the two phases of the computation: the assembly phase and the solve one, and for $d = 2$ (direct solver) and 3 (iterative solver). The ideal speedup $S(p) = p$ and the null speedup $S(p) = 1$ are represented by dotted lines. Observe on Fig 1.6 that for too small meshes, using too much processes is not profitable, as more time is spend by communications rather by computations, especially for the solve phase. Conversely, when the mesh size increases, using more processes leads to a remarkable speedup for both $d = 2$ and 3. The largest mesh used here contains about three millions of elements. The speedup behavior is roughly linear up to a critical number of processor denotes as p_c. Then, there is a transition to a plateau (the Amdahl's law), where communications dominate. Note that p_c increases with the mesh size: larger problems lead to a higher speedup. Also p_c increases also with the efficiency of communications.

Present computation times are measured on a BullX DLC supercomputer (Bull Newsca) composed of nodes having two intel sandy-bridge processors and connected to a FDR infiniband non-blocking low latency network. The assembly phase corresponds to `dirichlet.cc` (page 14) line 7 to 13 and the solve phase to lines 14 and 15.

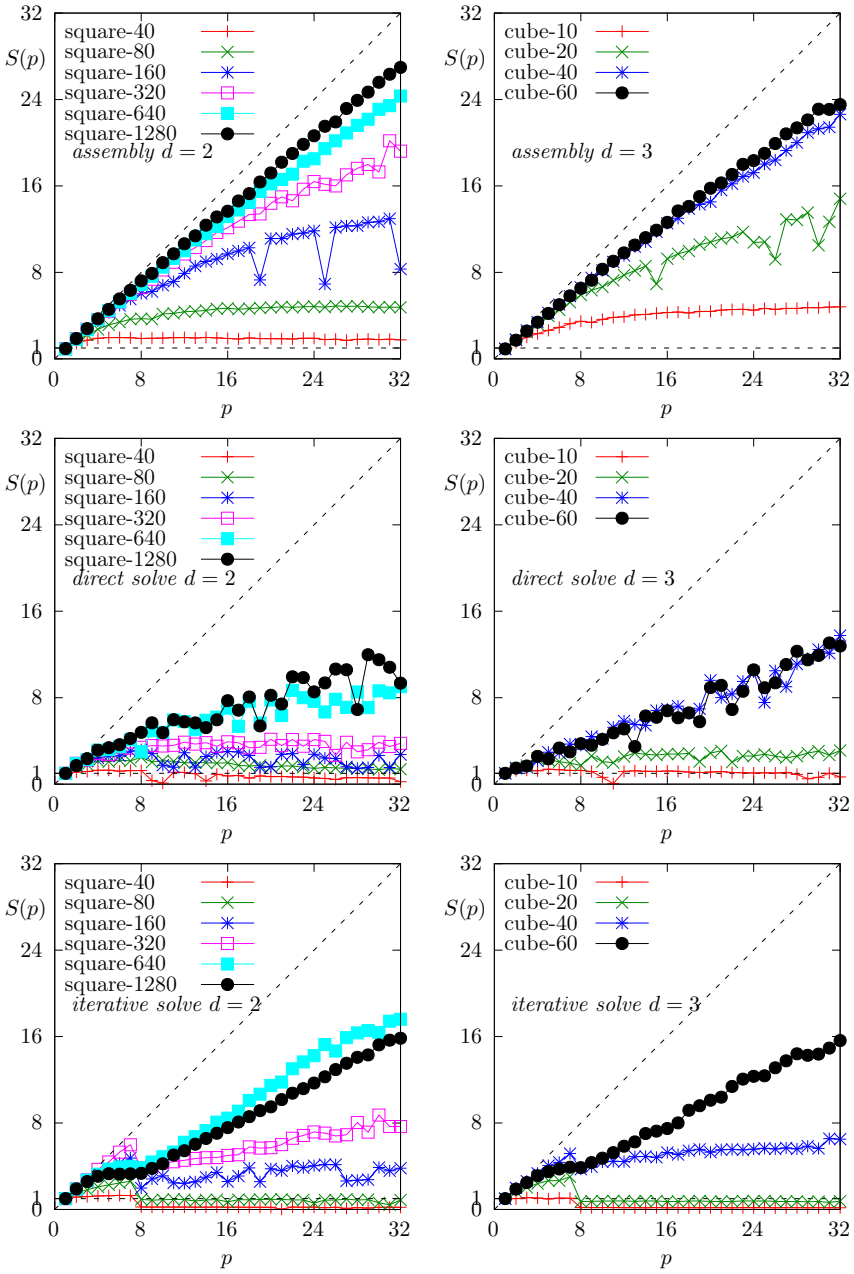


Figure 1.6: Distributed and massively parallel resolution of the model problem with P_1 element: speedup $S(p)$ versus the number of processors p during : (left-right) for $d = 2$ and 3, respectively ; (top) the assembly phase ; (center-bottom) the solve phase, direct and iterative solvers, respectively.

1.1.12 Non-homogeneous Dirichlet conditions

Formulation

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

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

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

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

where

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

Approximation

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

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

Then, we introduce:

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

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

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

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

File 1.2: `dirichlet-nh.cc`

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

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

$$f(x) = d\,\pi^2 \prod_{i=0}^{d-1} \cos(\pi x_i) \quad \text{and} \quad g(x) = \prod_{i=0}^{d-1} \cos(\pi x_i) \tag{1.4}$$

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

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

The following C++ code implement this problem by using the concept of *function object*, also called *class-function* (see e.g. Musser and Saini, 1996a). A convenient feature is the ability for function objects to store auxiliary parameters, such as the space dimension d for f here, or some constants, as π for f and g.

File 1.3: `cosinusprod_laplace.h`

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

Comments

The class `point` describes the coordinates of a point $(x_0, \ldots, x_{d-1}) \in \mathbb{R}^d$ as a d-uplet of `Float`. The `Float` type is usually a `double`. This type depends upon the **Rheolef** configuration (see Saramito, 2012a, installation instructions), and could also represent some high precision floating point class. The `dirichlet-nh.cc` code looks like the previous one `dirichlet.cc` related to homogeneous boundary conditions. Let us comments the changes. The dimension d comes from the geometry Ω:

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

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

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

Note that the function f that depends upon the dimension d parameter, is implemented by a *functor*, i.e. a C++ `class` that possesses the `operator()` member function. The space W_h of piecewise P_k functions defined on the boundary $\partial\Omega$ is defined by:

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

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

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

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

How to run the program

First, compile the program:

```
make dirichlet-nh
```

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

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

for the bidimensional case:

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

and for the tridimensional case:

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

The optional `-volume` allows a 3D color light volume graphical rendering. Here, the `P1` approximation can be replaced by `P2`, `P3`, etc, by modifying the command-line argument.

Error analysis

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

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

providing the approximate solution u_h uses P_k continuous finite element method, $k \geqslant 1$. Here, $\|.\|_{0,2,\Omega}$, $\|.\|_{0,\infty,\Omega}$ and $\|.\|_{1,2,\Omega}$ denotes as usual the $L^2(\Omega)$, $L^\infty(\Omega)$ and $H^1(\Omega)$ norms.

The following code implement the computation of the error.

File 1.4: `cosinusprod_error.cc`

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
#include "cosinusprod.h"
int main(int argc, char**argv) {
  environment rheolef(argc, argv);
  Float err_u_linf_expected = (argc > 1) ? atof(argv[1]) : 1e+38;
  field uh; din >> uh;
  geo omega = uh.get_geo();
  space Xh = uh.get_space();
  size_t k = Xh.degree();
  size_t d = Xh.get_geo().dimension();
  integrate_option iopt;
  iopt.set_order(min(3*(k+1)+4,size_t(17)));
  Float err_u_l2 = sqrt(integrate (omega, sqr(uh-u_exact(d)), iopt));
  string opts = Xh.get_basis().option().stamp();
  space Xh1 (omega, "P"+to_string(k+1)+"d"+opts);
  field euh = lazy_interpolate (Xh1, uh-u_exact(d));
  Float err_u_linf = euh.max_abs();
  Float err_u_h1  = sqrt(integrate (omega, norm2(grad_h(euh)), iopt));
  dout << "err_u_l2   " << err_u_l2 << endl
       << "err_u_linf " << err_u_linf   << endl
       << "err_u_h1   " << err_u_h1 << endl;
  return (err_u_linf <= err_u_linf_expected) ? 0 : 1;
}
```

File 1.5: `cosinusprod.h`

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

Running the program

```
make dirichlet-nh cosinusprod_error
```

After compilation, run the code by using the command:

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

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

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

Let $n_{\rm el}$ denotes the number of elements in the mesh. Since the mesh is quasi-uniform, we have $h \approx n_{\rm el}^{\frac{1}{d}}$ where d is the physical space dimension. Here $d = 2$ for our bidimensional mesh. Figure 1.7 plots in logarithmic scale the error versus $n_{\rm el}^{\frac{1}{2}}$ for both P_k approximations, $k = 1, 2, 3$ and the various norms. Observe that the error behaves as predicted by the theory.

Curved domains

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

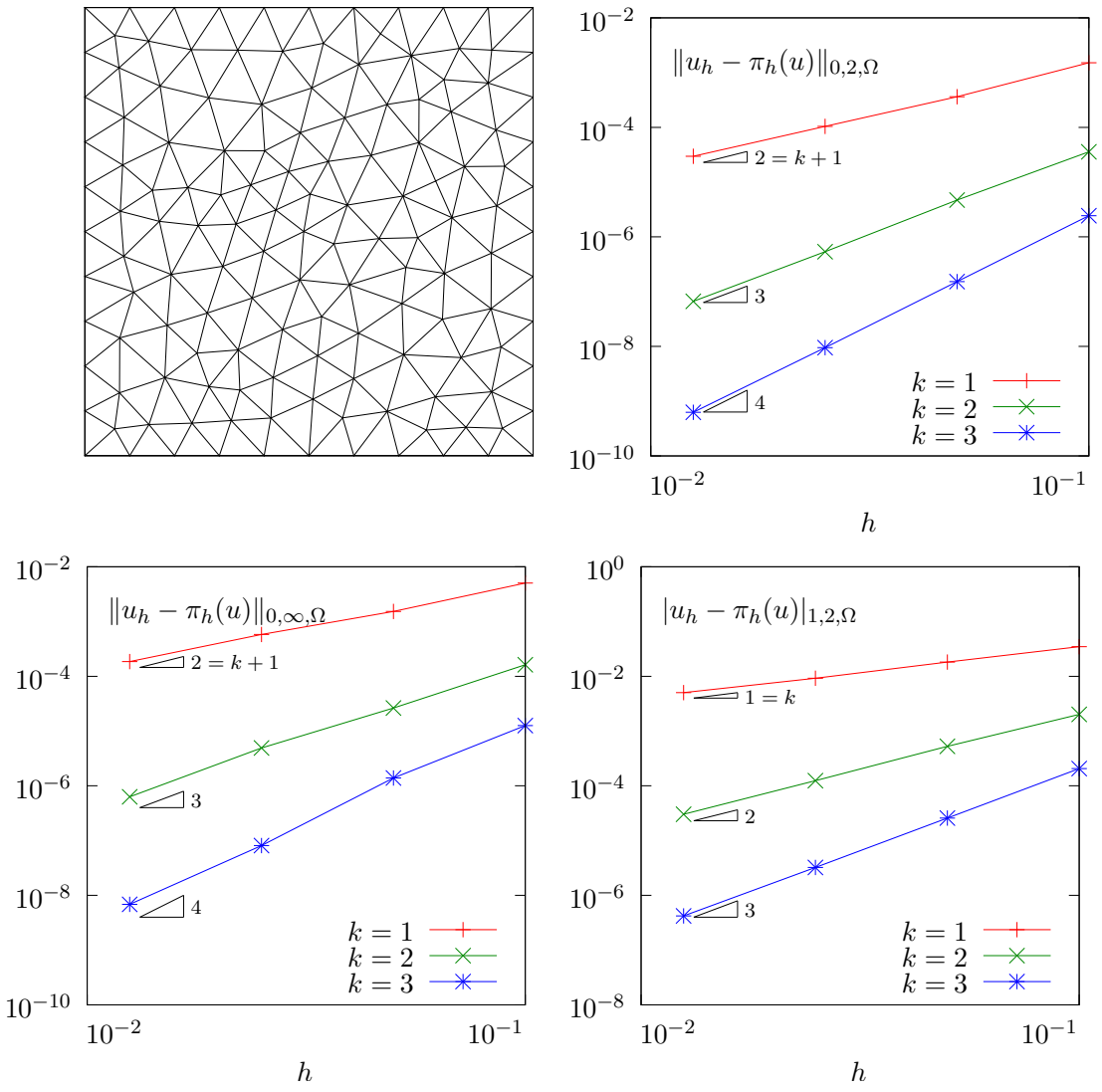


Figure 1.7: Strait geometry: error analysis in L^2, L^∞ and H^1 norms.

File 1.6: `cosinusrad_laplace.h`

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

File 1.7: `cosinusrad.h`

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

First, generate the test source file and compile it:

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

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

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

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

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

Observe Fig. 1.8: for meshes based on triangles: the error behave as expected when $k = 1, 2, 3, 4$. These features are available for arbitrarily P_k high order approximations and three-dimensional geometries. In practice, the current implementation is efficient up to $k = 5$: for higher polynomial degrees, a more specific implementation is in development.

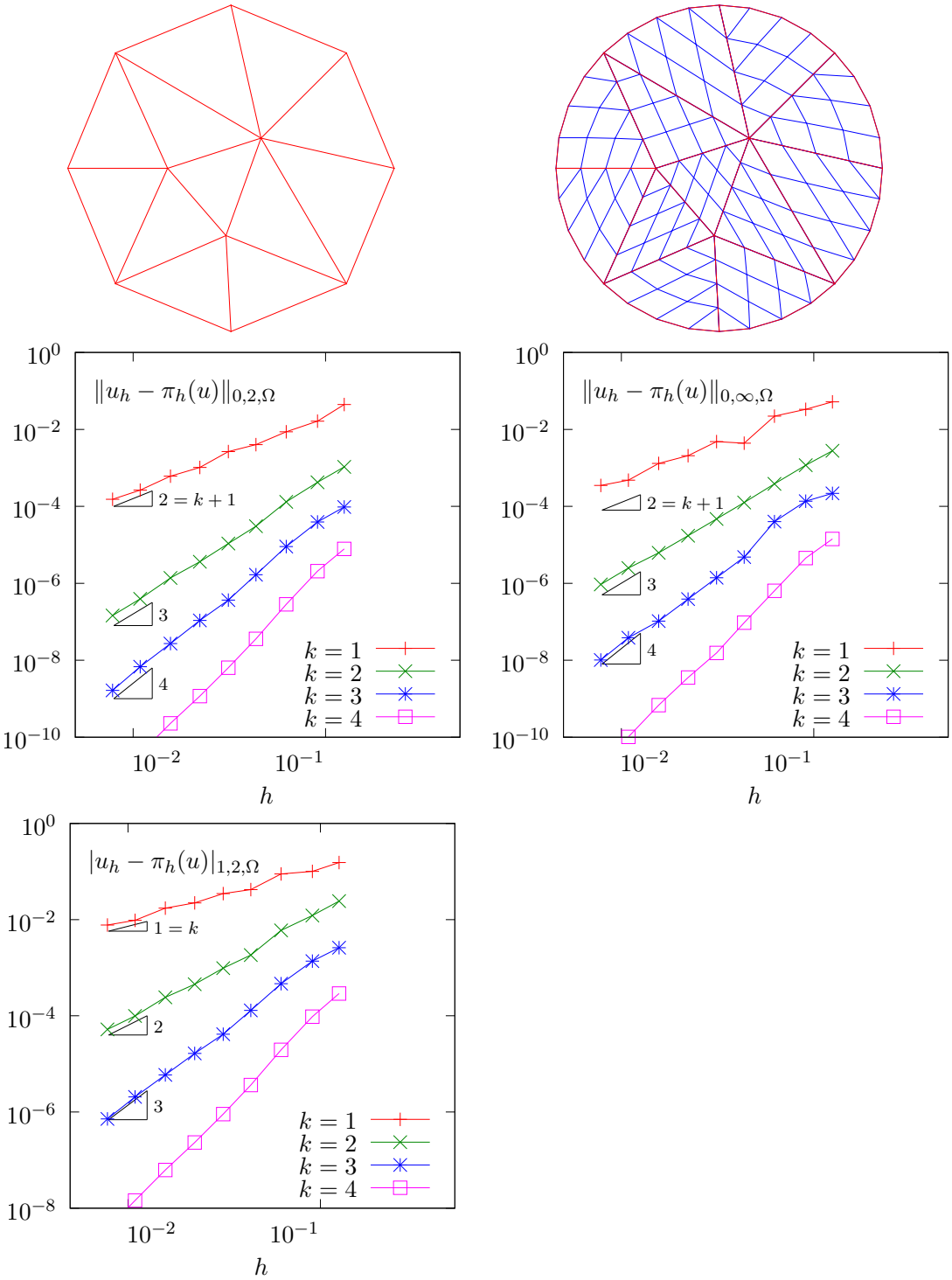


Figure 1.8: Curved domains (triangles): error analysis in L^2, L^∞ and H^1 norms.

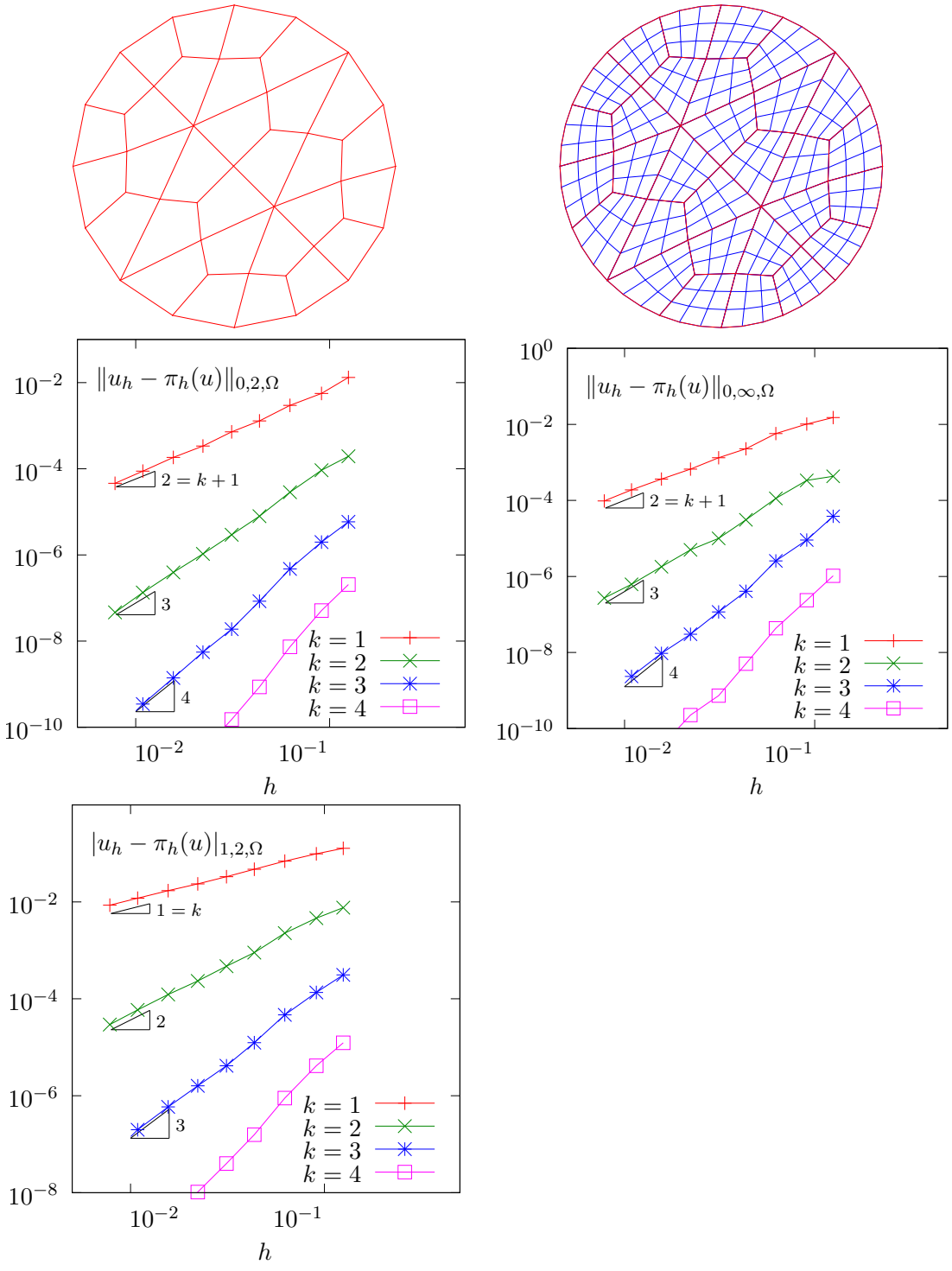


Figure 1.9: Curved domains (quadrangles): error analysis in L^2, L^∞ and H^1 norms.

1.2 Non-homogeneous Neumann boundary conditions for the Helmholtz operator

Formulation

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

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

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

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

where

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

Approximation

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

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

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

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

File 1.8: `neumann-nh.cc`

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

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

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

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

$$u(x) = \prod_{i=0}^{d-1} \sin(\pi x_i) \tag{1.5}$$

File 1.9: `sinusprod_helmholtz.h`

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

Comments

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

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

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

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

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

How to run the program

First, compile the program:

```
make neumann-nh
```

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

1.3 The Robin boundary conditions

Formulation

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

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

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

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

where

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

Approximation

As usual, let

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

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

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

File 1.10: `robin.cc`

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

Comments

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

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

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

How to run the program

First, compile the program:

```
make robin
```

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

1.4 Neumann boundary conditions for the Laplace operator

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

Formulation

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

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

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

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

Since this problem only involves the derivatives of u, it is defined up to a constant: it is then clear that its solution is never unique [Girault and Raviart, 1986, p. 11]. A discrete version of this problem could be solved iteratively by the conjugate gradient or the MINRES algorithm [Paige and Saunders, 1975]. In order to solve it by a direct method, we get round this difficulty by seeking u in the following space

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

where

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

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

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

where

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

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

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

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

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

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 $\lambda \,\mathrm{meas}(\Omega) = \ell(v_0)$. From the definition of $\ell(.)$ and the compatibility condition between the data f and g, we get $\lambda = 0$. Note that the saddle point problem extends to the case when f and g does not satisfies the compatibility condition, and in that case $\lambda = \ell(v_0)/\mathrm{meas}(\Omega)$.

Approximation

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

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

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

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

File 1.11: `neumann-laplace.cc`

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
size_t d;
Float f (const point& x) { return 1; }
Float g (const point& x) { return -0.5/d; }
int main(int argc, char**argv) {
  environment rheolef (argc, argv);
  geo omega (argv[1]);
  d = omega.dimension();
  space Xh (omega, argv[2]);
  trial u (Xh); test v (Xh);
  form  a  = integrate (dot(grad(u),grad(v)));
  field b  = integrate(v);
  field lh = integrate (f*v) + integrate ("boundary", g*v);
  form  A = {{   a,      b},
             { trans(b), 0}};
  field Bh = {   lh,     0};
  field Uh (Bh.get_space(), 0);
  A.set_symmetry(true);
  problem p (A);
  p.solve (Bh, Uh);
  dout << Uh[0];
}
```

Comments

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

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

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

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

Let

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

The problem admits the following matrix form:

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

The matrix that represents the bilinear fo and its right-hand side are assembled as:

```
form  A = {{   a,       b},
           { trans(b), 0}};
field Bh = {   lh,      0};
```

Both the pairs $\mathcal{U} = (u_h, \lambda)$ and $\mathcal{B} = (b, 0)$ belong to the vectorial space $X_h \times \mathbb{R}$. and $\mathcal{B} = (b, 0)$ belong to the Then, the variable `Uh` could be declared as:

```
field Uh (Bh.get_space(), 0);
```

Note that the matrix $\mathcal{A}$ is symmetric and non-singular, but indefinite : it admits eigenvalues that are either strictly positive or strictly negative. While the Choleski factorization is not possible, its variant the LDL^T one is performed, thanks to the `ldlt` function:

```
A.set_symmetry(true);
problem p (A);
p.solve (Bh, Uh);
```

Then, the `uh` field is extracted as the first component of the the `Uh` one:

```
dout << Uh[0];
```

How to run the program

As usual, enter:

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

1.5 Non-constant coefficients and multi-regions

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

Formulation

Let us consider a diffusion problem with a non-constant diffusion coefficient η in a domain bounded $\Omega \subset \mathbb{R}^d$, $d = 1, 2, 3$:

(P): find u defined in Ω such that:

$$
\begin{aligned}
-\mathrm{div}(\eta \boldsymbol{\nabla} u) &= f \text{ in } \Omega && (1.6)\\
u &= 0 \text{ on } \Gamma_{\mathrm{left}} \cup \Gamma_{\mathrm{right}} && (1.7)\\
\frac{\partial u}{\partial n} &= 0 \text{ on } \Gamma_{\mathrm{top}} \cup \Gamma_{\mathrm{bottom}} \text{ when } d \geqslant 2 && (1.8)\\
\frac{\partial u}{\partial n} &= 0 \text{ on } \Gamma_{\mathrm{front}} \cup \Gamma_{\mathrm{back}} \text{ when } d = 3 && (1.9)
\end{aligned}
$$

where f is a given source term. We consider here the important special case when η is piecewise

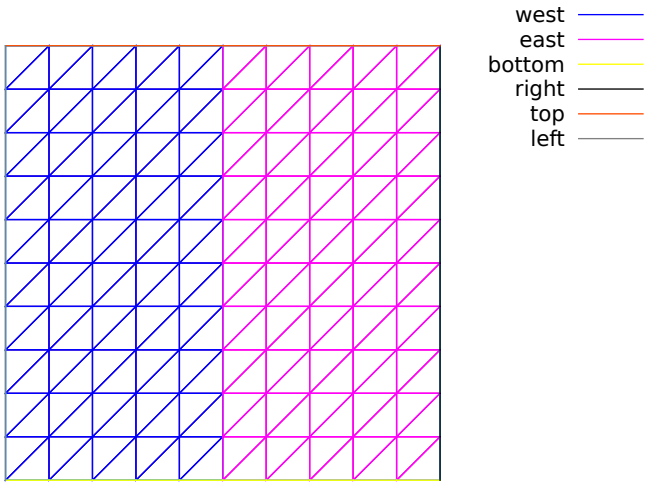


Figure 1.10: Transmission problem: the domain Ω partition: (Ω_{west} and Ω_{east}).

constant:

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

where $(\Omega_{\mathrm{west}}, \Omega_{\mathrm{east}})$ is a partition of Ω in two parts (see Fig. 1.10). This is the so-called **transmission** problem: the solution and the flux are continuous on the interface $\Gamma_0 = \partial\Omega_{\mathrm{east}} \cap \partial\Omega_{\mathrm{west}}$ between the two domains where the problem reduce to a constant diffusion one:

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

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

The variational formulation of this problem expresses:

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

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

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

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

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

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

File 1.12: `transmission.cc`

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
int main(int argc, char**argv) {
  environment rheolef (argc, argv);
  geo omega (argv[1]);
  string approx = (argc > 2) ? argv[2] : "P1d";
  Float epsilon = (argc > 3) ? atof(argv[3]) : 1e-2;
  space Xh (omega, approx);
  Xh.block ("left");
  Xh.block ("right");
  string eta_approx = "P" + to_string(Xh.degree()-1) + "d";
  space Qh (omega, eta_approx);
  field eta_h (Qh);
  eta_h ["east"] = 1;
  eta_h ["west"] = epsilon;
  trial u (Xh); test v (Xh);
  form  a  = integrate (eta_h*dot(grad(u),grad(v)));
  field lh = integrate (v);
  field uh (Xh);
  uh["left"] = 0; uh["right"] = 0;
  problem p (a);
  p.solve (lh, uh);
  dout << catchmark("epsilon") << epsilon << endl
       << catchmark("u")       << uh;
}
```

Comments

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

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

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

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

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

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

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

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

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

How to run the program ?

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

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

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

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

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

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

Since the exact solution is a piecewise second order polynomial and the change in the diffusion coefficient value fits the element boundaries, we obtain the exact solution for all the degrees of freedom of any P_k approximation, $k \geqslant 1$, as shown on Fig. 1.11 when $k = 1$. Moreover, when $k \geqslant 2$ then $u_h = u$ since X_h contains the exact solution u. The two dimensional case corresponds

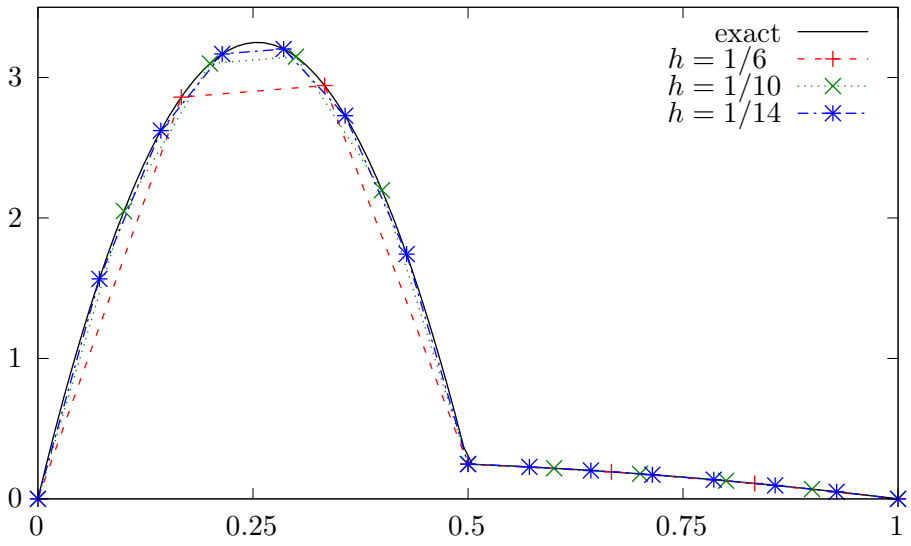


Figure 1.11: Transmission problem with $\varepsilon = 10^{-2}$, $d = 1$, P_1 approximation.

to the commands:

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

while the tridimensional to

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

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

Chapter 2

Fluids and solids computations

2.1 The linear elasticity and the Stokes problems

2.1.1 The linear elasticity problem

Formulation

The total Cauchy stress tensor expresses:

$$\sigma(\mathbf{u}) = \lambda \operatorname{div}(\mathbf{u}).I + 2\mu D(\mathbf{u}) \tag{2.1}$$

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

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

$$\begin{array}{rcl} -\mathbf{div}\,\sigma(\mathbf{u}) & = & \mathbf{f} \text{ in } \Omega, \\ \dfrac{\partial \mathbf{u}}{\partial \mathbf{n}} & = & 0 \text{ on } \Gamma_{\text{top}} \cup \Gamma_{\text{right}} \\ \mathbf{u} & = & 0 \text{ on } \Gamma_{\text{left}} \cup \Gamma_{\text{bottom}}, \\ \mathbf{u} & = & 0 \text{ on } \Gamma_{\text{front}} \cup \Gamma_{\text{back}}, \text{ when } d = 3 \end{array} \tag{2.2}$$

where $\mathbf{f} = (0, -1)$ when $d = 2$ and $\mathbf{f} = (0, 0, -1)$ when $d = 3$. The Lamé coefficients are assumed to satisfy $\mu > 0$ and $\lambda + \mu > 0$. Since the problem is linear, we can suppose that $\mu = 1$ without any loss of generality. recall that, in order to avoid mistakes with the `C++` style indexes, we denote by $(x_0, \ldots, x_{d-1})$ the Cartesian coordinate system in $\mathbb{R}^d$.

For $d = 2$ we define the boundaries:

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

and for $d = 3$:

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

These boundaries are represented on Fig. 2.1.

The variational formulation of this problem expresses:

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

$$a(\mathbf{u}, \mathbf{v}) = l(\mathbf{v}), \ \forall \mathbf{v} \in \mathbf{V}, \tag{2.3}$$

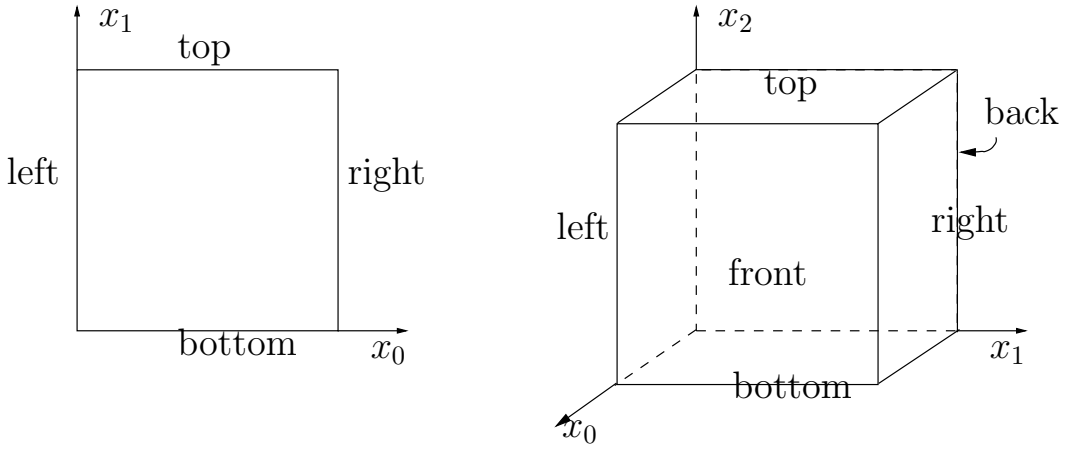


Figure 2.1: The boundary domains for the square and the cube.

where

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

Approximation

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

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

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

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

File 2.1: embankment.cc

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

File 2.2: embankment.icc

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

Comments

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

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

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

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

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

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

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

How to run the program

Compile the program as usual (see page 16):

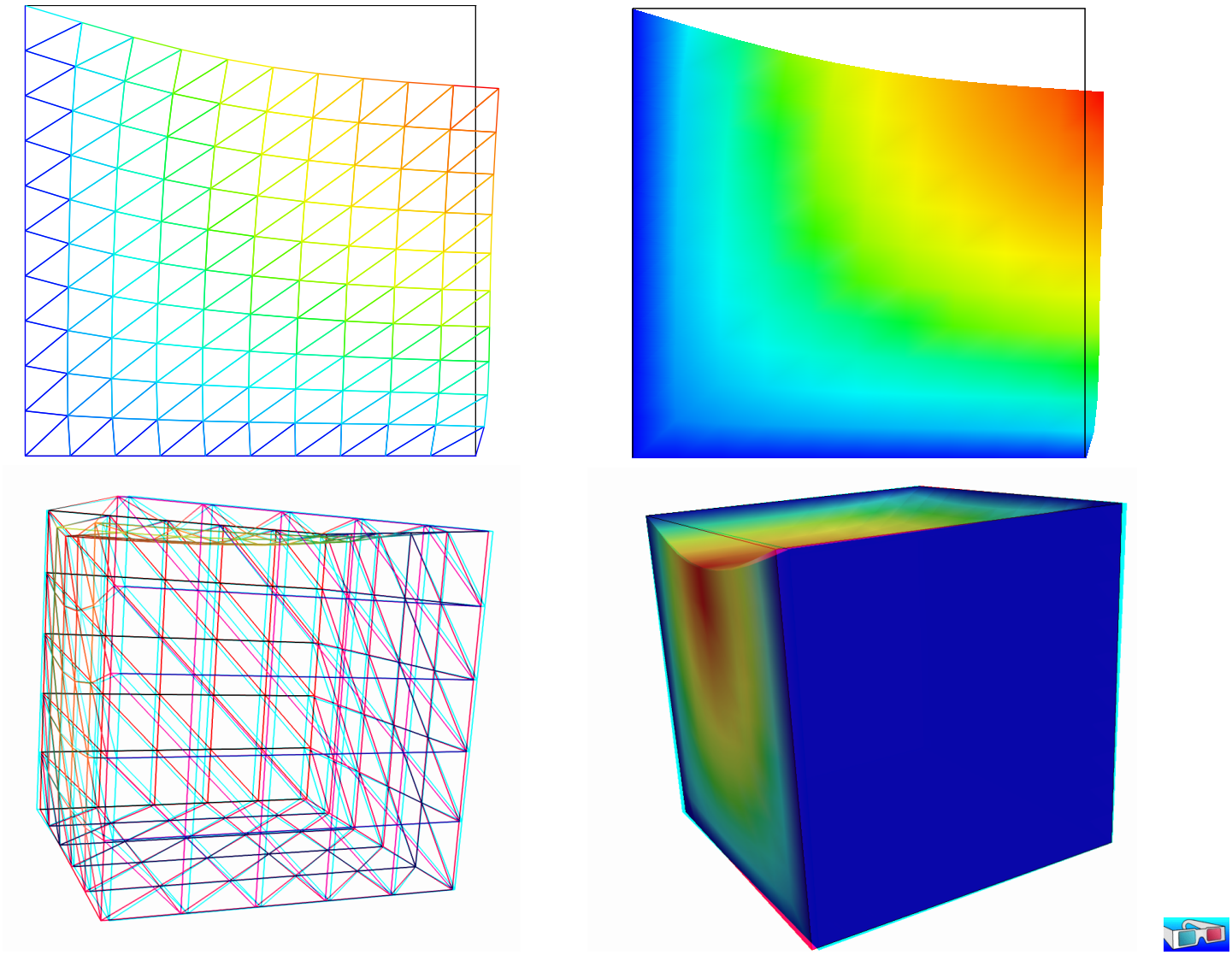


Figure 2.2: The linear elasticity for $\lambda = 1$ and $d = 2$ and $d = 3$: both wireframe and filled surfaces ; stereoscopic anaglyph mode for 3D solutions.

```
make embankment
```

and enter the commands:

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

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

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

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

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

It bases on the default `paraview` render for 2D and 3D geometries. The view is shown on Fig. 2.2. If you are in trouble with `paraview`, you can switch to the simpler `gnuplot` render:

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

The unix manual for the `field` command is available as:

```
man field
```

A specific field component can be also selected for a scalar visualization:

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

Next, perform a P_2 approximation of the solution:

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

Finally, let us consider the three dimensional case

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

The two last commands show the solution in 3D stereoscopic anaglyph mode. The graphic is represented on Fig. 2.2. The P_2 approximation writes:

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

2.1.2 Computing the stress tensor

Formulation and approximation

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

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

This computation expresses:

find σ_h such that:

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

where

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

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

File 2.3: `stress.cc`

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

Comments

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

How to run the program

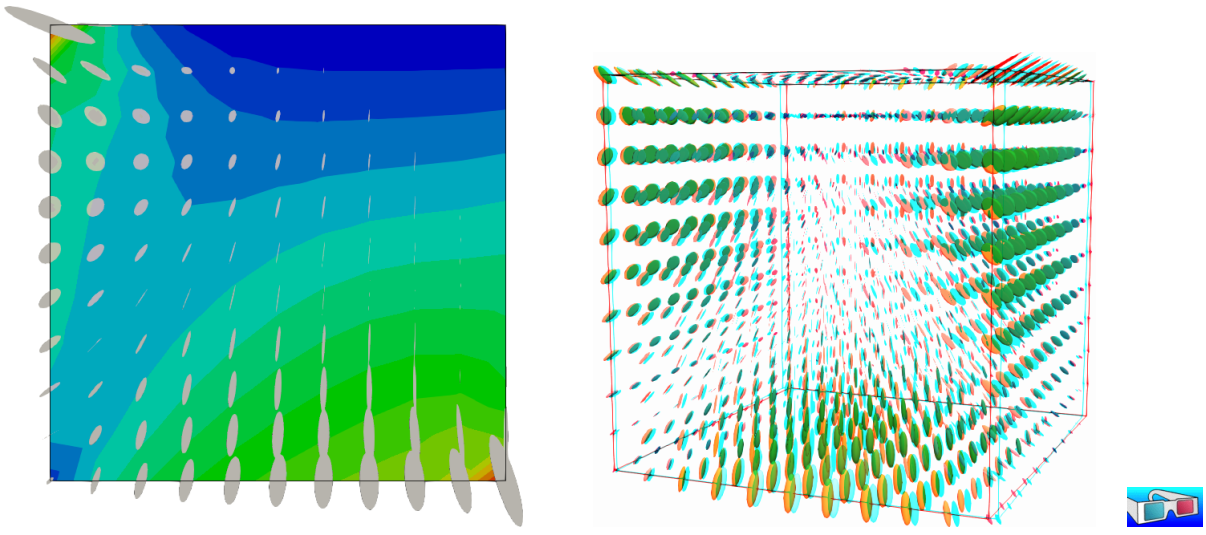


Figure 2.3: The stress tensor visualization (linear elasticity $\lambda = 1$).

First, compile the program:

```
make stress
```

The visualization for the stress tensor as ellipses writes:

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

This visualization based on `paraview` requires the `TensorGlyph` feature, available since `paraview-5.0`. Recall that the stress, as a derivative of the deformation, is P0 (resp. P1d) and discontinuous when the deformation is P1 (resp. P2) and continuous. The approximate stress tensor field is projected as a continuous piecewise linear space, using the `-proj` option. Conversely, the 3D visualization bases on ellipsoids:

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

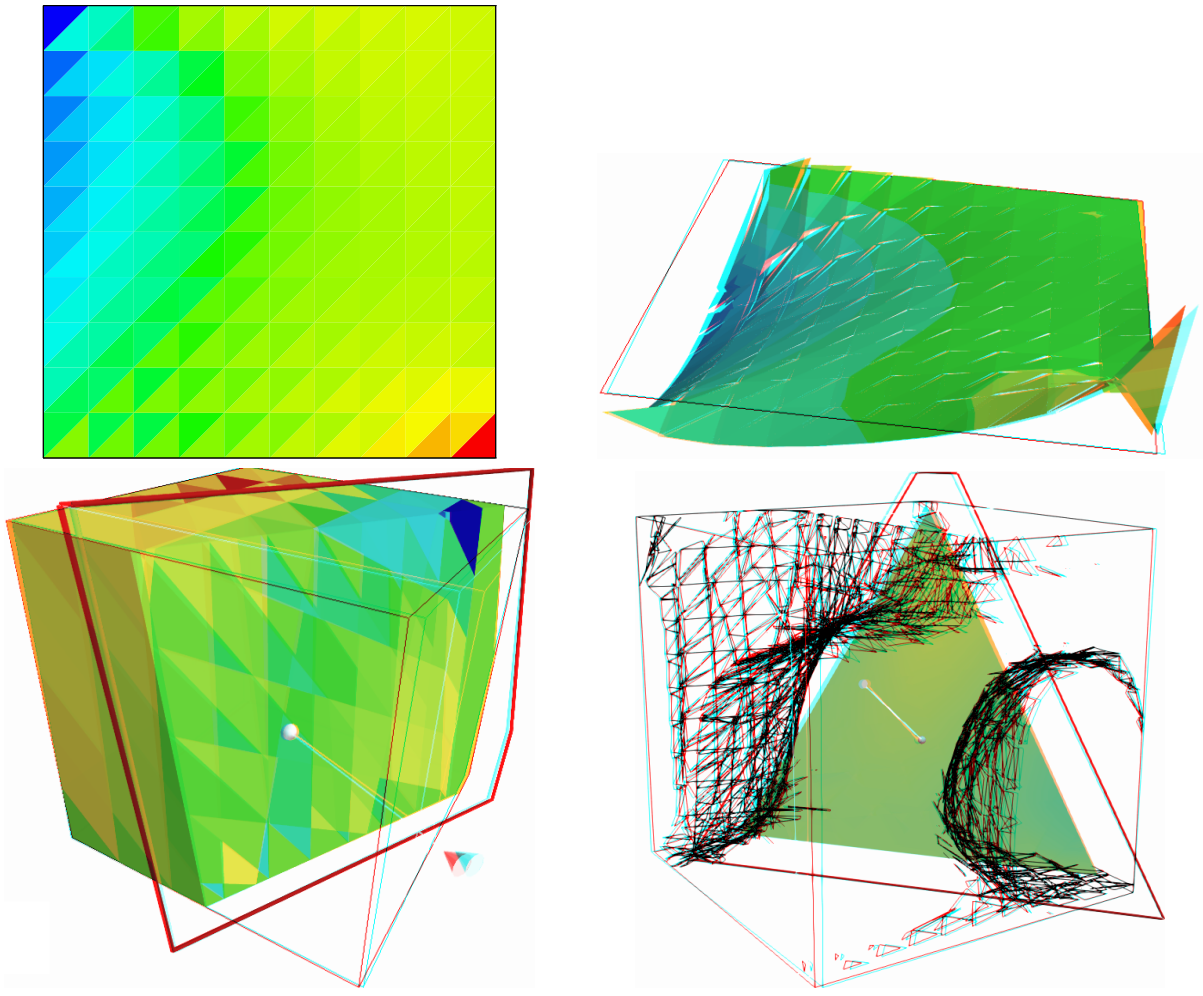


Figure 2.4: The σ_{01} stress component (linear elasticity $\lambda = 1$): $d = 2$ (top) and $d = 3$ (bottom) ; P_0 (left) and P_1 discontinuous approximation (right).

You can observe a discontinuous constant or piecewise linear representation of the approximate stress component σ_{01} (see Fig. 2.4):

```
field square-stress-P1.field -comp 01
field square-stress-P2.field -comp 01 -elevation
field square-stress-P2.field -comp 01 -elevation -stereo
```

Note that the `-stereo` implies the `paraview` render: this feature available with `paraview`. The approximate stress field can be also projected on a continuous piecewise space:

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

The tridimensional case writes simply (see Fig. 2.4):

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

and also the P1-projected versions write:

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

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

2.1.3 Mesh adaptation

The main principle of the auto-adaptive mesh writes [Hecht, 2006, Borouchaki et al., 1995, Castro-Diaz et al., 1997, Vallet, 1990, Roquet et al., 2000]:

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

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

Let us apply this principle to the elasticity problem.

File 2.4: `embankment_adapt.cc`

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

File 2.5: `elasticity_solve.icc`

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

File 2.6: `elasticity_criterion.icc`

```
field elasticity_criterion (Float lambda, const field& uh) {
  string grad_approx = "P" + to_string(uh.get_space().degree()-1) + "d";
  space Xh  (uh.get_geo(), grad_approx);
  if (grad_approx == "P0d") return lazy_interpolate (Xh, norm(uh));
  space T0h  (uh.get_geo(), grad_approx);
  size_t d = uh.get_geo().dimension();
  tensor I = tensor::eye (d);
  return lazy_interpolate (T0h, sqrt(2*norm2(D(uh)) + lambda*sqr(div(uh))));
}
```

Comments

The criterion is here:

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

The `elasticity_criterion` function compute it as

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

when using P_1, and as

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

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

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

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

How to run the program

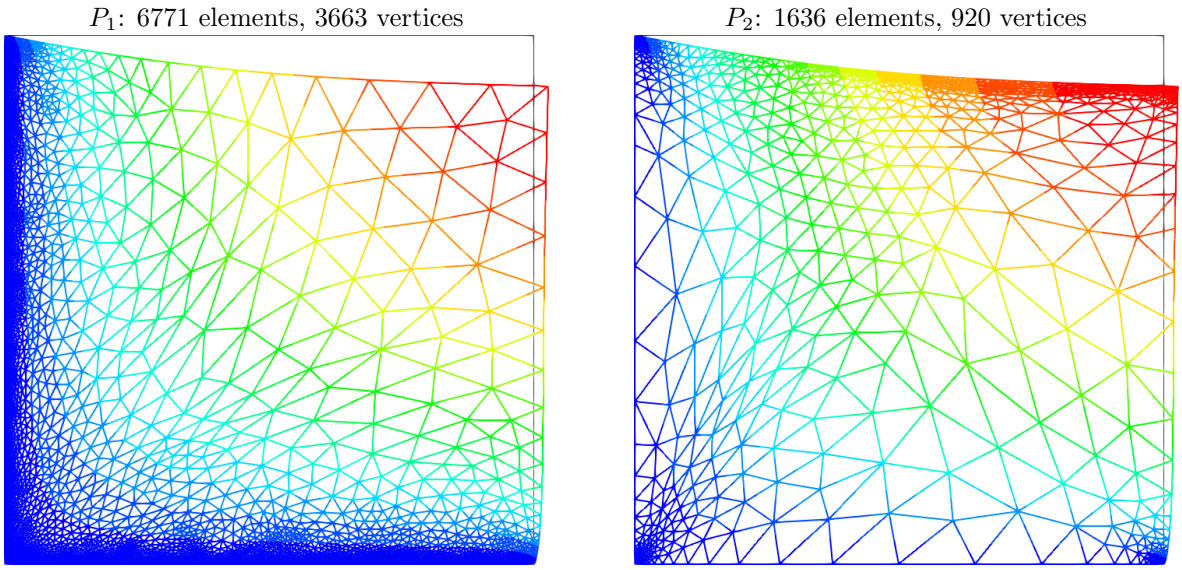


Figure 2.5: Adapted meshes: the deformation visualization for P_1 and P_2 approximations.

The compilation command writes:

```
make embankment_adapt
```

The mesh loop adaptation is initiated from a `bamg` mesh (see also appendix A.2.1).

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

The last command line argument is the target error. The code performs a loop of five mesh adaptations: the corresponding meshes are stored in files, from `square-001.geo.gz` to `square-005.geo.gz`, and the associated solutions in files, from `square-001.field.gz` to `square-005.field.gz`. The additional '`.gz`' suffix expresses that the files are compressed using `gzip`.

```
geo square-005.geo
field square-005.field -nofill
```

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

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

Then, the visualization writes:

```
geo square-005.geo
field square-005.field -nofill
```

A one-dimensional mesh adaptive procedure is also possible:

```
gmsh -1 line.mshcad -format msh2 -o line.msh
msh2geo line.msh > line.geo
geo line.geo
./embankment_adapt line P2 1e-3
geo line-005.geo
field line-005.field -comp 0 -elevation
```

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

2.1.4 The Stokes problem

Formulation

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

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

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

where $D(\mathbf{u}) = (\boldsymbol{\nabla}\mathbf{u} + \boldsymbol{\nabla}\mathbf{u}^T)/2$. The boundaries are represented on Fig. 2.1, page 44.
The variational formulation of this problem expresses:

(VFS) *find* $\mathbf{u} \in \mathbf{V}(1)$ *and* $p \in L_0^2(\Omega)$ *such that:*

$$\begin{array}{rcrcl} a(\mathbf{u},\mathbf{v}) & + & b(\mathbf{v},p) & = & 0, \ \forall \mathbf{v} \in \mathbf{V}(0), \\ b(\mathbf{u},q) & & & = & 0, \ \forall q \in L_0^2(\Omega), \end{array}$$

where

$$\begin{array}{rcl} a(\mathbf{u},\mathbf{v}) & = & \displaystyle\int_\Omega 2D(\mathbf{u}) : D(\mathbf{v})\, dx, \\ b(\mathbf{v},q) & = & \displaystyle -\int_\Omega \mathrm{div}(\mathbf{v})\, q\, dx. \\ \mathbf{V}(\alpha) & = & \{\mathbf{v} \in (H^1(\Omega))^2;\ \mathbf{v} = 0 \text{ on } \Gamma_{\text{left}} \cup \Gamma_{\text{right}} \cup \Gamma_{\text{bottom}} \text{ and } \mathbf{v} = (\alpha, 0) \text{ on } \Gamma_{\text{top}}\}, \quad \text{when } d = 2, \\ \mathbf{V}(\alpha) & = & \{\mathbf{v} \in (H^1(\Omega))^3;\ \mathbf{v} = 0 \text{ on } \Gamma_{\text{left}} \cup \Gamma_{\text{right}} \cup \Gamma_{\text{bottom}}, \\ & & \qquad \mathbf{v} = (\alpha, 0, 0) \text{ on } \Gamma_{\text{top}} \text{ and } v_2 = 0 \text{ on } \Gamma_{\text{back}} \cup \Gamma_{\text{front}}\}, \quad \text{when } d = 3, \\ L_0^2(\Omega) & = & \{q \in L^2(\Omega);\ \displaystyle\int_\Omega q\, dx = 0\}. \end{array}$$

Approximation

The Hood and Taylor [1973] finite element approximation of the Stokes problem is considered. We introduce a mesh $\mathcal{T}_h$ of Ω and the following finite dimensional spaces:

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

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

$$\begin{array}{lcll}
a(\mathbf{u}_h, \mathbf{v}) + b(\mathbf{v}, p_h) &=& 0, & \forall \mathbf{v} \in \mathbf{V}_h(0),\\
b(\mathbf{u}_h, q) &=& 0, & \forall q \in Q_h.
\end{array} \tag{2.4}$$

File 2.7: `cavity.h`

```
struct cavity {
  static space velocity_space (const geo& omega, string approx) {
    space Xh (omega, approx, "vector");
    Xh.block("top");  Xh.block("bottom");
    if (omega.dimension() == 3) {
      Xh.block("back"); Xh.block("front");
      Xh[1].block("left");  Xh[1].block("right");
    } else {
      Xh.block("left"); Xh.block("right");
    }
    return Xh;
  }
  static field velocity_field (const space& Xh, Float alpha=1) {
    field uh (Xh, 0.);
    uh[0]["top"] = alpha;
    return uh;
  }
  static space streamf_space (geo omega, string approx) {
    string valued = (omega.dimension() == 3) ? "vector" : "scalar";
    space Ph (omega, approx, valued);
    Ph.block("top");  Ph.block("bottom");
    if (omega.dimension() == 3) {
      Ph.block("back"); Ph.block("front");
    } else {
      Ph.block("left"); Ph.block("right");
    }
    return Ph;
  }
  static field streamf_field (space Ph) {
    return field(Ph, 0);
  }
};
```

File 2.8: stokes_cavity.cc

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

Comments

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

The problem admits the following matrix form:

$$\begin{pmatrix} \mathtt{a} & \mathtt{trans(b)} \\ \mathtt{b} & 0 \end{pmatrix} \begin{pmatrix} \mathtt{uh} \\ \mathtt{ph} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

An initial value for the pressure field is provided:

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

The solve call to the Stokes problem writes:

```
  problem_mixed stokes (a, b);
  stokes.solve (field(Xh,0), field(Qh,0), uh, ph);
```

The two first arguments of the `solve` member function represents the zero right-hand-side of the problem. For two-dimensional geometries ($d = 2$), this system is solved by a direct method (see Saramito, 2016b, p. 41). Conversely, for tridimensional geometries ($d = 3$), it is solved by the preconditioned conjugate gradient algorithm (see Saramito, 2016b, p. 56). In that case, the preconditioner is by default the mass matrix `mp` for the pressure space: as showed by Klawonn [1998], 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 on the web site and via the unix command:

```
  man problem_mixed
```

How to run the program

We assume that the previous code is contained in the file 'stokes_cavity.cc'. Then, compile the program as usual (see page 16):

```
  make stokes_cavity
```

and enter the commands:

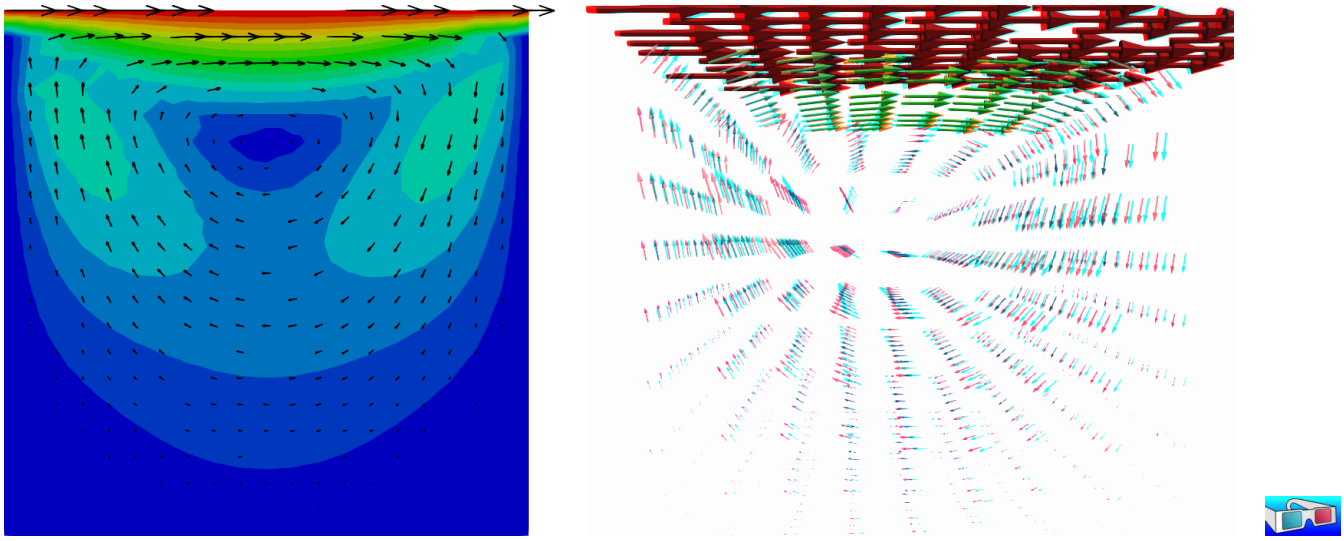


Figure 2.6: The velocity visualization for $d = 2$ and $d = 3$ with stereo anaglyph.

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

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

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

Run also some scalar visualizations:

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

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

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

and observe the convergence.

Finally, let us consider the three dimensional case:

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

and the corresponding visualization:

```
field cube.field -velocity -scale 3
field cube.field -comp 0
field cube.field -comp 1
field cube.field -comp 2
field cube.field -mark p
```

2.1.5 Computing the vorticity

Formulation and approximation

When $d = 2$, we define [Girault and Raviart, 1986, page 30] for any distributions ϕ and $\mathbf{v}$:

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

and when $d = 3$:

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

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

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

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

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

The approximate variational formulation writes:

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

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

File 2.9: `vorticity.cc`

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

Comments

As for the stress tensor (see `stress.cc`, page 48), the vorticity is obtained by a direct interpolation of the u_h first derivatives and its approximation is *discontinuous* at inter-element boundaries.

How to run the program

For $d = 2$, just enter:

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

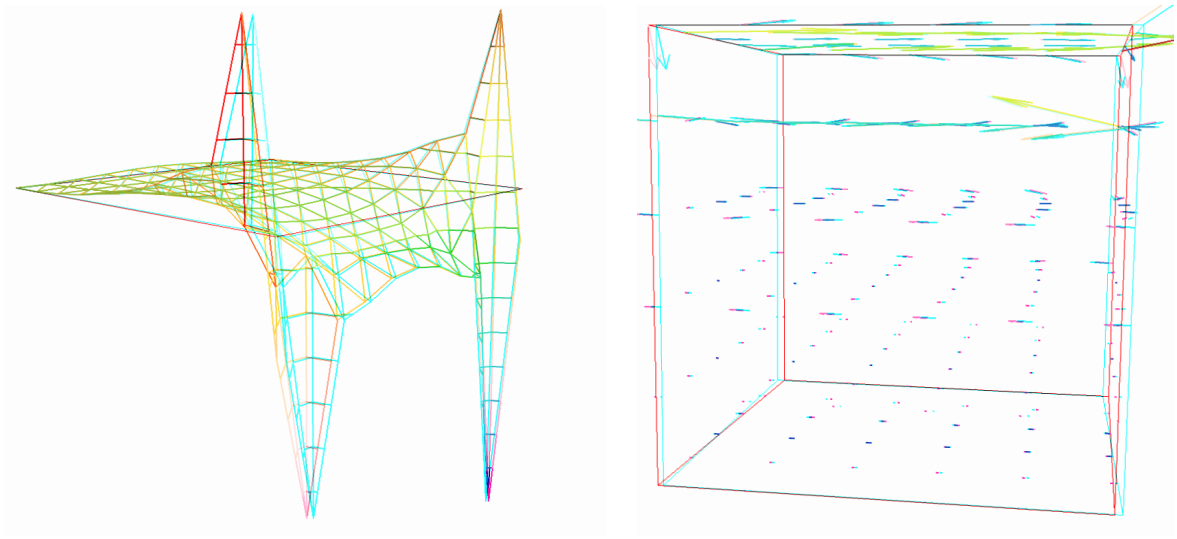


Figure 2.7: The vorticity: elevation view for $d = 2$ and vector representation for $d = 3$ (with anaglyph).

and you observe a discontinuous piecewise linear representation of the approximate vorticity. Also, the vorticity presents two sharp peaks at the upper corners of the driven cavity: the vorticity is unbounded and the peaks will increase with mesh refinements. This singularity of the solution is due to the boundary condition for the first component of the velocity u_0 that jumps from zero to one at the corners. The approximate vorticity field can also be projected on a continuous piecewise linear space, using the `-proj` option (See Fig. 2.7 left):

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

For $d = 3$, the whole vorticity vector can also be visualized (See Fig. 2.7 right):

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

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

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

2.1.6 Computing the stream function

Formulation and approximation

When $d = 3$, the stream function is a vector-valued field $\boldsymbol{\psi}$ that satisfies [Girault and Raviart, 1986, page 90]: $\mathbf{curl}\,\boldsymbol{\psi} = \mathbf{u}$ and $\operatorname{div}\boldsymbol{\psi} = 0$. From the identity:

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

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

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

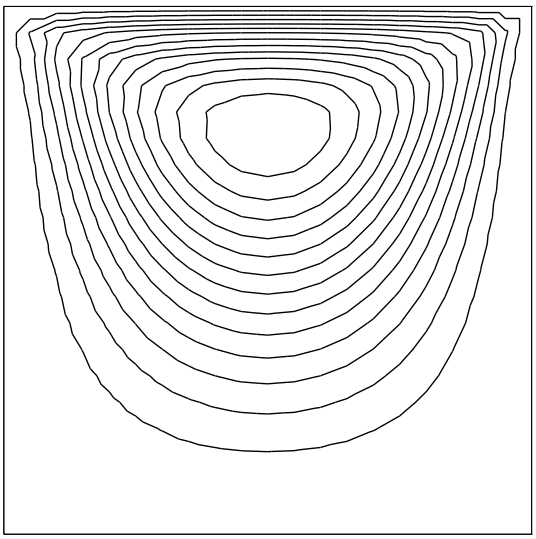
When $d = 2$, the stream function ψ is a scalar-valued field the solution of the following problem [Girault and Raviart, 1986, page 88]:

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

File 2.10: `streamf_cavity.cc`

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
#include "cavity.h"
int main (int argc, char** argv) {
  environment rheolef (argc, argv);
  field uh;
  din >> uh;
  string approx = "P" + to_string(uh.get_space().degree());
  space Ph = cavity::streamf_space (uh.get_geo(), approx);
  space Xh = uh.get_space();
  size_t d = uh.get_geo().dimension();
  trial u (Xh), psi (Ph); test phi (Ph);
  form a = (d == 3) ? integrate (ddot(grad(psi),grad(phi)))
                    : integrate ( dot(grad(psi),grad(phi)));
  form b = (d==3) ? integrate (dot(curl(u),phi))
                  : integrate (curl(u)*phi);
  field psi_h = cavity::streamf_field (Ph);
  field lh = b*uh;
  problem p (a);
  p.solve (lh, psi_h);
  dout << catchmark("psi") << psi_h;
}
```

How to run the program



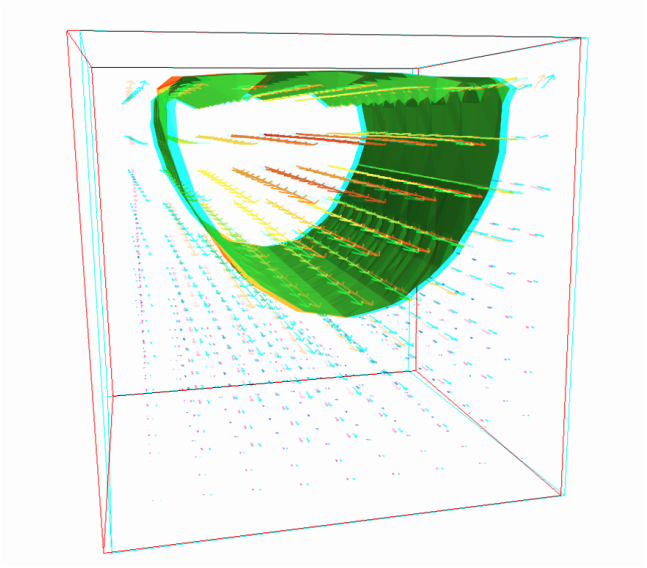


Figure 2.8: The stream function visualization: isolines for $d = 2$, and combined vectors and isonorm surface for $d = 3$.

For $d = 2$, just enter (see Fig. 2.8 left):

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

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

```
./streamf_cavity < cube.field | field -velocity -scale 10 -
```

The second component of the stream function is showed by:

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

The combined representation of Fig. 2.8.right has been obtained in two steps. First, enter:

```
./streamf_cavity < cube.field | field -comp 1 -noclean -noexecute -name psi1 -
./streamf_cavity < cube.field | field -velocity -scale 10 -
```

The `-noclean -noexecute` options cause the creation of the '`.vtk`' file for the second component, without running the `paraview` visualization. Next, in the `paraview` window associated to the whole stream function, select the `File->Open` menu and load '`psi1.vtk`' and click on the green button `Apply`. Finally, select the `Filters/Common/Contours` menu: the isosurface appears. Observe that the 3D stream function is mainly represented by its second component.

2.2 Nearly incompressible elasticity and the stabilized Stokes problems

2.2.1 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 :

$$p = -\lambda \operatorname{div} \mathbf{u}$$

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

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

The variational formulation of this problem expresses:
(VFE) *find* $\mathbf{u} \in V(1)$ *and* $p \in L^2(\Omega)$ *such that:*

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

where

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

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

Approximation

As for the Stokes problem, the Hood and Taylor [1973] 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 2.11: `incompressible-elasticity.cc`

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

Comments

The problem admits the following matrix form:

$$\begin{pmatrix} \mathtt{a} & \mathtt{trans(b)} \\ \mathtt{b} & -\mathtt{c} \end{pmatrix} \begin{pmatrix} \mathtt{uh} \\ \mathtt{ph} \end{pmatrix} = \begin{pmatrix} \mathtt{lh} \\ \mathtt{0} \end{pmatrix}$$

The problem is similar to the Stokes one (see page 55). This system is solved by:

```
  problem_mixed elasticity (a, b, c);
  elasticity.solve (lh, field(Qh,0), uh, ph);
```

For two-dimensional problems, a direct solver is used by default. In the three-dimensional case, an iterative algorithm is the default: the preconditioned conjugate gradient. The preconditioner is here the mass matrix `mp` for the pressure. As showed by Klawonn [1998], 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

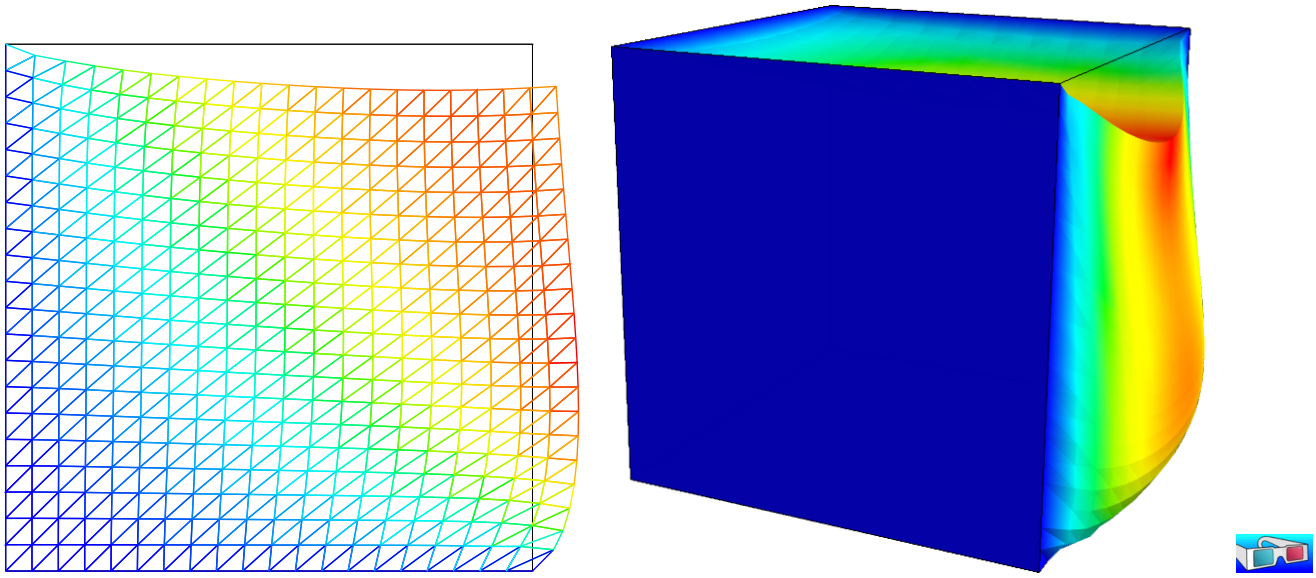


Figure 2.9: The incompressible linear elasticity ($\lambda = +\infty$) for $N = 2$ and $N = 3$.

We assume that the previous code is contained in the file '`incompressible-elasticity.cc`'. Compile the program as usual (see page 16):

```
make incompressible-elasticity
```

and enter the commands:

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

mkgeo_grid -T 10 > cube.geo
./incompressible-elasticity cube.geo 0 > cube.field
field cube.field -fill -scale 2
```

The visualization is performed as usual: see section 2.1.1, page 45. Compare the results on FIG. 2.9, obtained for $\lambda = +\infty$ with those of FIG. 2.2, page 46, obtained for $\lambda = 1$.

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

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

Formulation and approximation

Let us go back to the Stokes problem. In section 2.1.4, page 53, the Taylor-Hood finite element was considered. Here, we turn to the mini-element proposed by Arnold, Brezzi, and Fortin [1984], also well-known as the *P1-bubble* element. This element is generally less precise than the Taylor-Hood one, but becomes popular, mostly because it is easy to implement in two and three dimensions and furnishes a P_1 approximation of the velocity field. Moreover, this problem develops some links with stabilization technique and will presents some new **Rheolef** features.

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

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

where $B(K) = \text{vect}(\lambda_1 \times \ldots \times \lambda_{d+1})$ and λ_i are the barycentric coordinates of the simplex K. The $B(K)$ space is related to the *bubble* local space. The approximate problem is similar to (2.4), page 54, up to the choice of finite dimensional spaces.

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

We consider the abrupt contraction geometry:

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

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

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

where

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

The matrix structure is similar to those of the Taylor-Hood element (see section 2.1.4, page 53). Since $\mathbf{X}_h = \mathbf{X}_h^{(1)} \oplus \mathbf{B}_h$, any element $u_h \in X_h$ can be written as a sum $u_h = u_{1,h} + u_{b,h}$ where $u_{1,h} \in \mathbf{X}_h^{(1)}$ and $u_{b,h} \in \mathbf{B}_h$. Remark that

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

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

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

An alternative and popular implementation of this element eliminates the unknowns related to the bubble components (see e.g. Abdalass, 1987, page 24). Remark that, on any element $K \in \mathcal{T}_h$,

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

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

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

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

where $\tilde{L}_p = L_p - A_b^{-1} L_p$ and $C = B_b A_b^{-1} B_b^T$. Remarks that the matrix structure is similar to those of the nearly incompressible elasticity (see 2.2.1, page 2.2.1). This reduced matrix formulation of the $P_1 b - P1$ element is similar to the direct $P_1 - P_1$ stabilized element, proposed by Brezzi and Pitkäranta [1984].

File 2.12: `stokes_contraction_bubble.cc`

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
#include "contraction.h"
int main(int argc, char**argv) {
  environment rheolef (argc, argv);
  geo  omega (argv[1]);
  space X1h = contraction::velocity_space (omega, "P1");
  space Bh  (omega, "bubble", "vector");
  space Qh  (omega, "P1");
  trial u1 (X1h), ub (Bh), p (Qh);
  test  v1 (X1h), vb (Bh), q (Qh);
  form b1 = integrate (-div(u1)*q);
  form bb = integrate (-div(ub)*q);
  form a1 = integrate (2*ddot(D(u1),D(v1)));
  integrate_option iopt;
  iopt.invert = true;
  form inv_ab = integrate (2*ddot(D(ub),D(vb)), iopt);
  form c = bb*inv_ab*trans(bb);
  field u1h = contraction::velocity_field (X1h);
  field ph (Qh, 0);
  problem_mixed stokes (a1, b1, c);
  stokes.solve (field(X1h,0), field(Qh,0), u1h, ph);
  dout << catchmark("inv_lambda") << 0 << endl
       << catchmark("u")  << u1h
       << catchmark("p")  << ph;
}
```

Comments

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

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

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

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

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

File 2.13: `contraction.h`

```
struct contraction {
  struct base {
     base (geo omega) : c(0), umax(0), cartesian(true) {
       c = omega.xmax()[1];
       string sys_coord = omega.coordinate_system_name();
       cartesian = (sys_coord == "cartesian");
       umax = cartesian ? 3/(2*c) : 4/sqr(c);
     }
     Float c, umax;
     bool cartesian;
  };
  struct u_upstream: base {
    u_upstream (geo omega) : base(omega) {}
    Float operator() (const point& x) const {
      return base::umax*(1-sqr(x[1]/base::c)); }
  };
  static space velocity_space (geo omega, string approx) {
    space Xh (omega, approx, "vector");
    Xh.block ("wall");
    Xh.block ("upstream");
    Xh[1].block ("axis");
    Xh[1].block ("downstream");
    return Xh;
  }
  static field velocity_field (space Xh) {
    geo omega = Xh.get_geo();
    string approx = "P" + to_string(Xh.degree());
    space Wh  (omega["upstream"], approx);
    field uh (Xh, 0.);
    uh[0]["upstream"] = lazy_interpolate (Wh, u_upstream(omega));
    return uh;
  }
  static space streamf_space (geo omega, string approx) {
    space Ph (omega, approx);
    Ph.block("upstream");
    Ph.block("wall");
    Ph.block("axis");
    return Ph;
  }
  struct psi_upstream: base {
    psi_upstream (geo omega) : base(omega) {}
    Float operator() (const point& x) const {
      Float y = (x[1]/base::c);
      if (base::cartesian) {
        return  (base::umax*base::c)*(y*(1-sqr(y)/3) - 2./3);
      } else {
        return 0.5*(base::umax*sqr(base::c))*(sqr(y)*(1-sqr(y)/2) - 0.5);
      }
    }
  };
  static field streamf_field (space Ph) {
    geo omega = Ph.get_geo();
    space Wh (omega["upstream"], Ph.get_approx());
    field psi_h (Ph, 0);
    psi_upstream psi_up (omega);
    psi_h["upstream"] = lazy_interpolate (Wh, psi_up);
    psi_h["wall"] =  0;
    psi_h["axis"] = -1;
    return psi_h;
  }
};
```

Without loss of generality, we assume that the half width of the downstream channel is assumed to be equal to one. The Poiseuille velocity upstream boundary condition `u_upsteam` is then scaled such that the downstream average velocity is equal to one. By this way, the flow rate in the half upstream and downstream channel are also equal to one. The stream function is defined up to a constant: we assume that it is equal to -1 on the axis of symmetry: by this way, it is equal to

zero on the wall.

The file '`contraction.h`' also contains a treatment of the axisymmetric variant of the geometry: this case will be presented in the next paragraph. Note also the automatic computation of the geometric coordinate system and contraction ratio c from the input mesh, as:

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

These parameters are transmitted via a base class to the class-function that computes the Poiseuille upstream flow boundary condition.

How to run the program

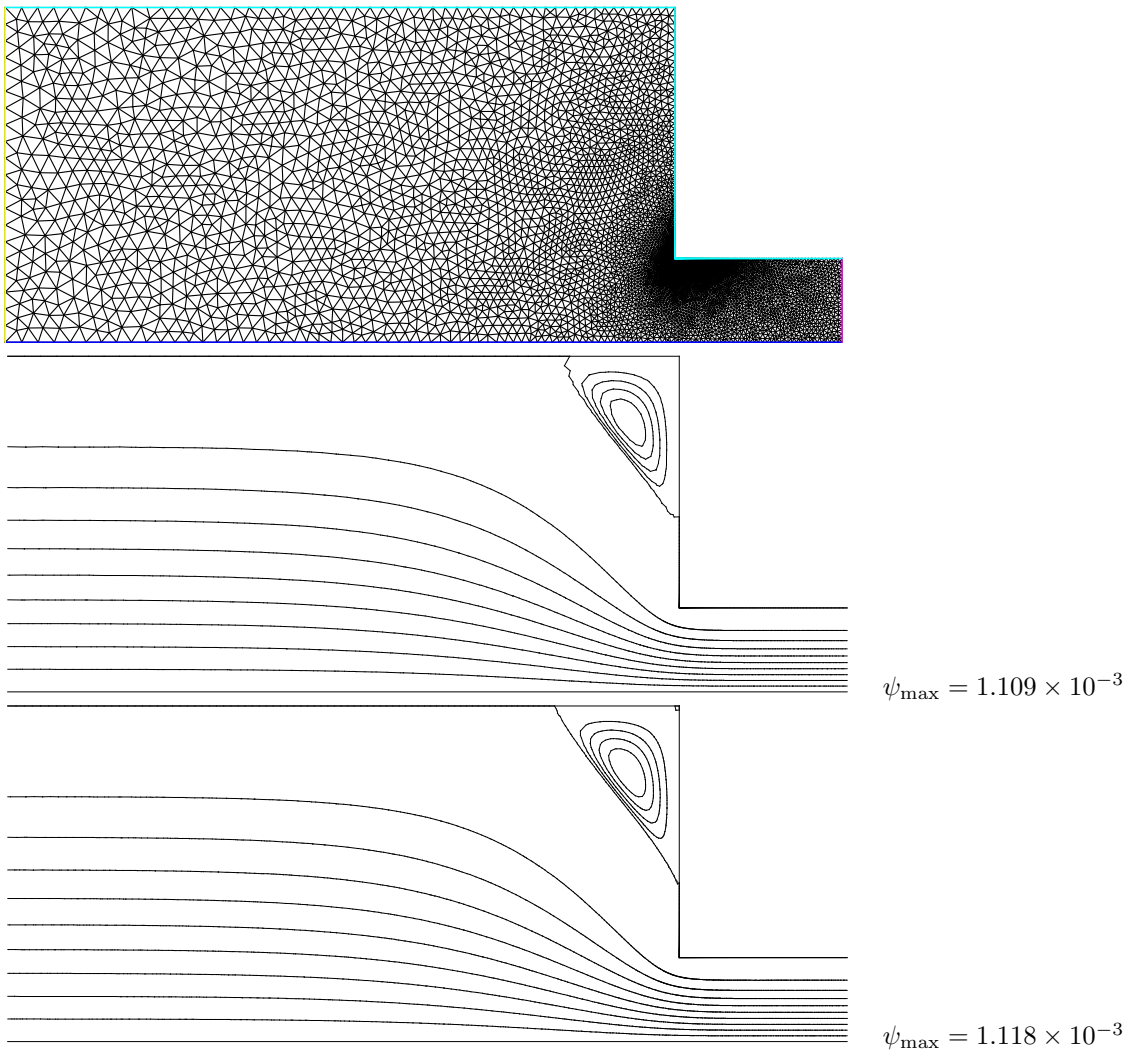


Figure 2.10: Solution of the Stokes problem in the abrupt contraction: (top) the mesh; (center) the P_1 stream function associated to the P_1b-P_1 element; (bottom) the P_2 stream function associated to the P_2-P_1 Taylor-Hood element.

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 build from the geometry

description file '`contraction.mshcad`', which is provided in the example directory of the **Rheolef** distribution. The building mesh procedure is presented with details in appendix A.2, page A.2.

```
gmsh -2 contraction.mshcad -format msh2 -o contraction.msh
msh2geo contraction.msh > contraction.geo
geo contraction.geo
```

The mesh is represented on Fig. 2.10.top. Then, the computation and the visualization writes:

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

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

File 2.14: `streamf_contraction.cc`

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
#include "contraction.h"
int main (int argc, char** argv) {
  environment rheolef (argc, argv);
  field uh;
  din >> uh;
  const geo& omega = uh.get_geo();
  size_t d = omega.dimension();
  Float c = omega.xmax()[1];
  string approx = "P" + to_string(uh.get_space().degree());
  space Ph = contraction::streamf_space (omega, approx);
  field psi_h = contraction::streamf_field (Ph);
  integrate_option iopt;
  iopt.ignore_sys_coord = true;
  const space& Xh = uh.get_space();
  trial psi (Ph), u (Xh);
  test   xi (Ph), v (Xh);
  form a = (d == 3) ? integrate (ddot(grad(psi), grad(xi)))
                    : integrate ( dot(grad(psi), grad(xi)), iopt);
  field lh = integrate (dot(uh,bcurl(xi)));
  problem p (a);
  p.solve (lh, psi_h);
  dout << catchmark("psi") << psi_h;
}
```

Note the usage of the optional parameter `iopt` to the `integrate` function.

```
iopt.ignore_sys_coord = true;
```

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

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

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

The stream function ψ (see also section 2.1.6) is computed and visualized as:

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

The P_1 stream function is represented on Fig. 2.10.center. The stream function is zero along the wall and the line separating the main flow and the vortex located in the outer corner of the contraction. Thus, the isoline associated to the zero value separates the main flow from the vortex. In order to observe this vortex, an extra `-n-iso-negative 10` option is added: ten isolines are drawn for negatives values of ψ, associated to the main flow, and `n_iso-10` for the positives values, associated to the vortex.

A similar computation based on the Taylor-Hood $P_2 - P_1$ element is implemented in `stokes_contraction.cc`. The code is similar, up to the boundary conditions, to `stokes_cavity.cc` (see page 54): thus it is not listed here but is available in the **Rheolef** example directory.

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

The associated P_2 stream function is represented on Fig. 2.10.bottom. Observe that the two solutions are similar and that the vortex activity, defined as $\psi_{\max}$, is accurately computed with the two methods (see also Saramito, 1990, Fig. 5.11.a, page 143).

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

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

2.2.3 Axisymmetric geometries

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

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

Remark the additional line in the header:

```
coordinate_system zr
```

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

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

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

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

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

2.2.4 The axisymmetric stream function and stress tensor

In the axisymmetric case, the velocity field $\mathbf{u} = (u_z, u_r)$ can be expressed in terms of the Stokes stream function ψ by (see Batchelor Batchelor, 1967, p.453 and Wikipedia, 2012):

$$\mathbf{u} = (u_z, u_r) = \left(\frac{1}{r}\frac{\partial \psi}{\partial r}, \quad -\frac{1}{r}\frac{\partial \psi}{\partial z} \right) \tag{2.5}$$

Recall that in the axisymmetric case:

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

Thus, from this definition, in axisymmetric geometries $\mathbf{u} \neq \mathbf{curl}\,\psi$ and the definition of ψ differs from the 2D plane or 3D cases (see section 2.1.6, page 58).

Let us turn to a variational formulation in order to compute ψ from $\mathbf{u}$. For any $\xi \in H^1(\Omega)$, let us multiply (2.5) by $\mathbf{v} = (\partial_r \xi,\ -\partial_z \xi)$ and then integrate over Ω with the $r\,\mathrm{d}r\,\mathrm{d}z$ weight. For any known $\mathbf{u}$ velocity field, the problem writes:

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

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

where we have introduced the following bilinear forms:

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

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

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

and

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

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

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

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

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

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

As an example, let us reconsider the contraction geometry (see section 2.2.2, page 62), extended in the axisymmetric case. In that case, the functional space is defined by:

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

with

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

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

The following unix commands generate the axisymmetric geometry:

```
gmsh -2 contraction.mshcad -format msh2 -o contraction.msh
msh2geo -zr contraction.msh > contraction-zr.geo
more contraction-zr.geo
geo contraction-zr.geo
```

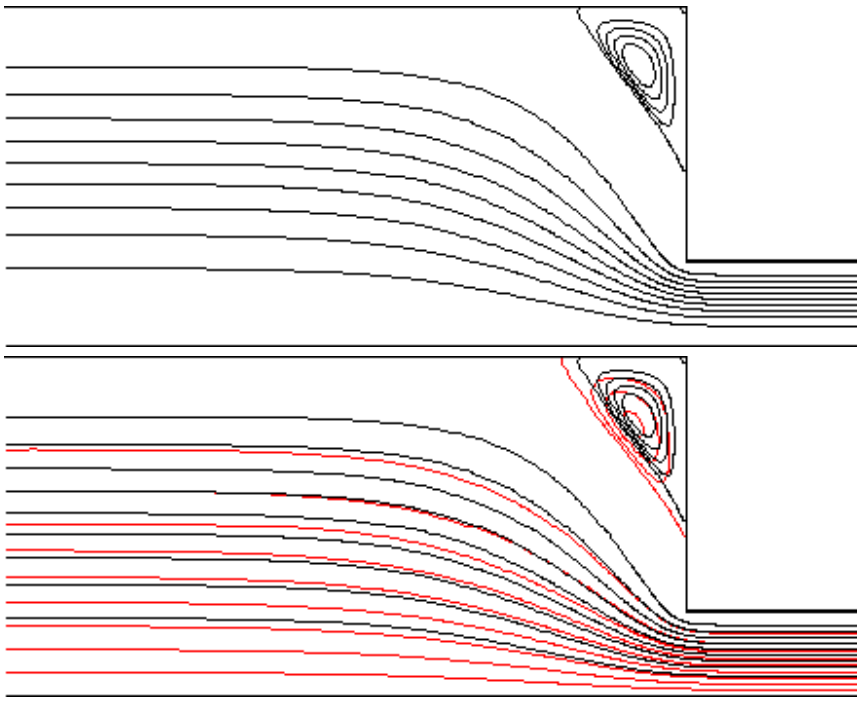


Figure 2.11: Solution of the axisymmetric Stokes problem in the abrupt contraction: (top) the P_2 stream function associated to the $P2 - P_1$ element; (bottom) comparison with the 2D Cartesian solution (in red).

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

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

The solution is represented on Fig. 2.11: it slightly differs from the 2D Cartesian solution, as computed in the previous section (see Fig. 2.10). The vortex size is smaller but its intensity $\psi_{\max} = 1.84 \times 10^{-3}$ is higher. Despite the stream functions looks like similar, the plane solutions are really different, as we can observe from a cut of the first component of the velocity along the axis (see Fig. 2.12):

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

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

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

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

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

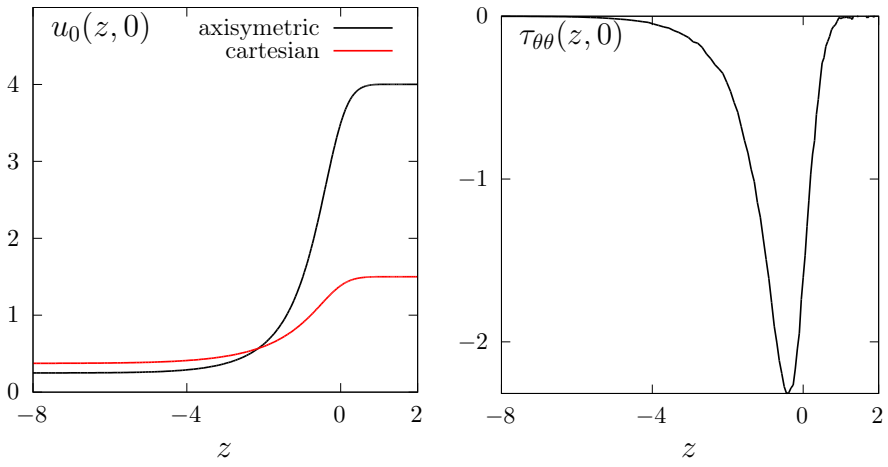


Figure 2.12: Solution of the plane and axisymmetric Stokes problem in the abrupt contraction: cut along the axis of symmetry: (left): u_0; (right) $\tau_{\theta\theta}$.

Thus, there is an additional non-zero component $\tau_{\theta\theta}$ that is automatically integrated into the computations in **Rheolef**. The incompressibility relation leads to $\mathrm{tr}(\tau) = \tau_{zz} + \tau_{rr} + \tau_{\theta\theta} = 0$. Here $\sigma_{\mathrm{tot}} = -p.I + \tau$ is the total Cauchy stress tensor (by a dimensionless procedure, the viscosity can be taken as one). By reusing the `stress.cc` code (see page 48) we are able to compute the tensor components:

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

The visualization along the axis of symmetry for the $\tau_{\theta\theta}$ component is obtained by (see Fig. 2.12):

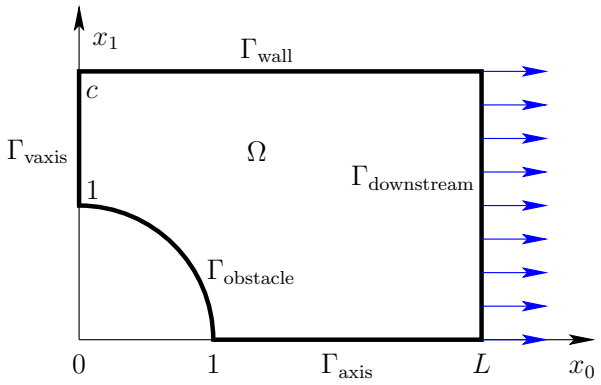
```
field contraction-zr-P1-tau.field -comp 22 -proj -cut -normal 0 1 -origin 0 1e-15 -gnuplot
```

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

2.3 [New] Slip boundary conditions

Formulation

We consider an approximation of the Stokes problem with slip boundary conditions. As a test case, we consider the flow around a circular obstacle, as represented on Fig. 2.13. We assume a permanent flow and: due to the symmetries versus upstream/downstream and with respect to the horizontal axis, the computational domain reduces to the quarter of the geometry.
This problem writes [Verfürth, 1987]:



$$\Gamma_{\text{wall}} \cup \Gamma_{axis} \cup \Gamma_{obstacle} : \boldsymbol{u}.\boldsymbol{n} = 0 \text{ and } \boldsymbol{\sigma}_{nt} = 0$$
$$\Gamma_{\text{vaxis}} : \boldsymbol{u}_t = 0 \text{ and } \sigma_{nn} = 0$$
$$\Gamma_{downstream} : \boldsymbol{u} = (1, 0)$$

Figure 2.13: Slip boundary conditions for the flow around an obstacle.

(P): *find* $\boldsymbol{u}$ *and* p*, defined in* Ω*, such that*

$$\begin{cases} \boldsymbol{\sigma} = -p\boldsymbol{I} + 2D(\boldsymbol{u}) & \text{in } \Omega & \text{(2.6a)} \\ -\mathbf{div}\,\boldsymbol{\sigma} = 0 & \text{in } \Omega & \text{(2.6b)} \\ -\operatorname{div}\boldsymbol{u} = 0 & \text{in } \Omega & \text{(2.6c)} \\ \boldsymbol{u}.\boldsymbol{n} = 0 \text{ and } \boldsymbol{\sigma}_{nt} = 0 & \text{on } \Gamma_{wall} \cup \Gamma_{axis} \cup \Gamma_{obstacle} & \text{(2.6d)} \\ \boldsymbol{u}_t = 0 \text{ and } \sigma_{nn} = 0 & \text{on } \Gamma_{vaxis} & \text{(2.6e)} \\ \boldsymbol{u} = \boldsymbol{e}_0 & \text{on } \Gamma_{downstream} & \text{(2.6f)} \end{cases}$$

with the notation $\boldsymbol{v}_t = \boldsymbol{v} - (\boldsymbol{v}.\boldsymbol{n})\boldsymbol{n}$, $\tau_{nn} = (\boldsymbol{\tau}\boldsymbol{n}).\boldsymbol{n}$ and $\boldsymbol{\tau}_{nt} = \boldsymbol{\tau}\boldsymbol{n} - \tau_{nn}\boldsymbol{n}$ for any vector $\boldsymbol{v}$ and tensor $\boldsymbol{\tau}$. Observe that Γ_{wall}, Γ_{axis} and Γ_{vaxis} are parallel to the coordinate axis: the corresponding slip boundary condition writes also

$$\begin{aligned} u_{d-1} = 0 \text{ and } \sigma_{i,d-1} = 0 \quad &\text{on } \Gamma_{wall} \cup \Gamma_{axis}, \quad 1 \leqslant i \leqslant d-2 \\ u_i = 0 \text{ and } \sigma_{0,0} = 0 \quad &\text{on } \Gamma_{vaxis}, \quad 1 \leqslant i \leqslant d-1 \end{aligned}$$

It remains one slip boundary condition on the curved boundary domain $\Gamma_{obstacle}$: imposing such a boundary condition is the main difficulty of this problem. There are mainly three approaches for the imposition of this boundary condition: (i) the regularization ; (ii) the Lagrange multiplier weak imposition ; (iii) the strong imposition. The main drawback of the Lagrange multiplier weak imposition approach is the discretization of the Lagrange multiplier λ, that should satisfy the inf-sup condition [Verfürth, 1987, 1991, Liakos, 2001, Caglar and Liakos, 2009]. An alternative is to introduce stabilization terms (Verfürth, 1991, p. 621, eqn (4.1)), but the resulting problem is no more symmetric. The strong imposition [Verfürth, 1985] requires some modifications of the finite element basis along the slip boundary: this promising feature is in development in the **Rheolef** library. The rest of this section focuses on the regularization approach.

The main idea of the regularization approach is to replace the slip boundary condition (2.6d) on the curved boundary domain $\Gamma_{obstacle}$ by Robin one:

$$\boldsymbol{\sigma}_{\boldsymbol{nt}} + \varepsilon^{-1}\boldsymbol{u}.\boldsymbol{n} = 0 \quad \text{on } \Gamma_{obstacle}$$

where $\varepsilon > 0$ is the regularization parameter. It leads to the following variational formulation:

$(FV)_\varepsilon$: *find* $\boldsymbol{u} \in V(1)$ *and* $p \in L^2(\Omega)$ *such that*

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

with

$$\begin{aligned} a(\boldsymbol{u},\boldsymbol{v}) &= \int_\Omega 2D(\boldsymbol{u}) : 2D(\boldsymbol{v})\,\mathrm{d}x + \varepsilon^{-1} \int_{\partial\Omega} (\boldsymbol{u}.\boldsymbol{n})(\boldsymbol{v}.\boldsymbol{n})\,\mathrm{d}s \\ b(\boldsymbol{v},q) &= -\int_\Omega q \operatorname{div} \boldsymbol{v}\,\mathrm{d}x \\ V(\alpha) &= \Big\{ \boldsymbol{v} \in \left(H^1(\Omega)\right)^d \;/\; \boldsymbol{v} = \alpha \boldsymbol{e}_0 \text{ on } \Gamma_{downstream} \\ &\qquad \text{and } u_i = 0 \text{ on } \Gamma_{vaxis}, \quad 1 \leqslant i \leqslant d-1 \\ &\qquad \text{and } u_{d-1} = 0 \text{ on } \Gamma_{wall} \cup \Gamma_{axis} \Big\} \end{aligned}$$

Approximation

The curved boundary $\partial\Omega$ is replaced polynomial approximation $\partial\mathscr{T}_h$. Then, a natural procedure would be to replace the the normal $\boldsymbol{n}$ on $\partial\Omega$ by the normal $\boldsymbol{n}_h$ on $\partial\mathscr{T}_h$ in the previous expression of the bilinear form a. This approach do not converge in general and this counter-intuitive feature is called the Babuška paradox, see e.g. Verfürth [1985, p. 473]. We have to deal with either the exact normal $\boldsymbol{n}$ or a more accurate approximation $\widetilde{\boldsymbol{n}}_h$ of $\boldsymbol{n}$. In the present case, as the exact normal $\boldsymbol{n}$ is known, we use it.

Otherwise, the space $V(\alpha)$ is completely standard, for any $\alpha \in \mathbb{R}$ and can be approximated by classical Lagrange finite elements. Thus, the Hood and Taylor [1973] finite element approximation of the Stokes problem is considered. The present implementation '`stokes_obstacle_slip_regul.cc`' supports both the 2D Cartesian and the 3D axisymmetric cases, so the 3D Cartesian case is not implemented here. Note that the 2D Cartesian case corresponds to a 3D cylindrical obstacle while the 3D axisymmetric case corresponds to a spherical obstacle. The '`streamf_obstacle_slip_move.cc`' computes the stream function the modified velocity field $\widetilde{\boldsymbol{u}}_h = \boldsymbol{e}_0 - \boldsymbol{u}_h$ relative to the moving obstacle.

File 2.15: `stokes_obstacle_slip_regul.cc`

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
point n_exact (const point& x) { return -x; }
int main(int argc, char**argv) {
  environment rheolef (argc, argv);
  geo  omega (argv[1]);
  Float eps = (argc > 2) ? atof(argv[2]) : 1e-7;
  space Xh (omega, "P2", "vector");
  Xh.block ("downstream");
  Xh[1].block ("wall");
  Xh[1].block ("axis");
  Xh[1].block ("vaxis");
  space Qh  (omega, "P1");
  trial u (Xh), p (Qh);
  test  v (Xh), q (Qh);
  form a  = integrate (2*ddot(D(u),D(v)))
          + (1/eps)*integrate("obstacle",dot(u,n_exact)*dot(v,n_exact));
  form b  = integrate (-div(u)*q);
  field uh (Xh,0);
  uh[0]["downstream"] = 1;
  field ph (Qh, 0);
  problem_mixed stokes (a, b);
  stokes.solve (field(Xh,0), field(Qh,0), uh, ph);
  dout << catchmark("u")  << uh
       << catchmark("p")  << ph;
}
```

File 2.16: streamf_obstacle_slip_move.cc

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
int main (int argc, char** argv) {
  environment rheolef (argc, argv);
  field uh;
  din >> uh;
  point e0 (1,0);
  uh = lazy_interpolate (uh.get_space(), e0 - uh);
  string approx = "P" + to_string(uh.get_space().degree());
  const geo& omega = uh.get_geo();
  space Ph (omega, approx);
  Ph.block ("wall");
  Ph.block ("axis");
  Ph.block ("downstream");
  field psi_h (Ph,0.);
  trial psi (Ph);
  test   xi (Ph);
  integrate_option iopt;
  iopt.ignore_sys_coord = true;
  form a = integrate ( dot(grad(psi), grad(xi)), iopt);
  field lh = integrate (dot(uh,bcurl(xi)));
  problem p (a);
  p.solve (lh, psi_h);
  dout << catchmark("psi") << psi_h;
}
```

How to run the program

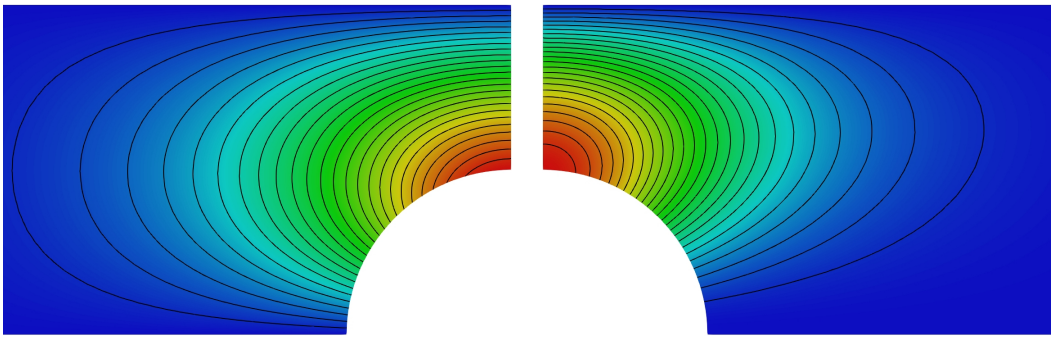


Figure 2.14: Slip boundary conditions for the flow around a cylinder (left) and sphere (right): isovalues of the stream function.

The run is detailed in the axisymmetric case. The mkgeo_obstacle script generates the mesh of the geometry:

```
mkgeo_obstacle -zr -name obstacle-zr
geo obstacle-zr.geo
```

Then the compilation and run writes:

```
make stokes_obstacle_slip_regul ./streamf_obstacle_slip_move
./stokes_obstacle_slip_regul obstacle-zr > obstacle-zr.field
field -velocity obstacle-zr.field
./streamf_obstacle_slip_move < obstacle-zr.field | field -bw -n-iso 25 -
```

Observe on Fig. 2.14 that the trajectories, as represented by the stream function, differ in the Cartesian an axisymmetric cases.

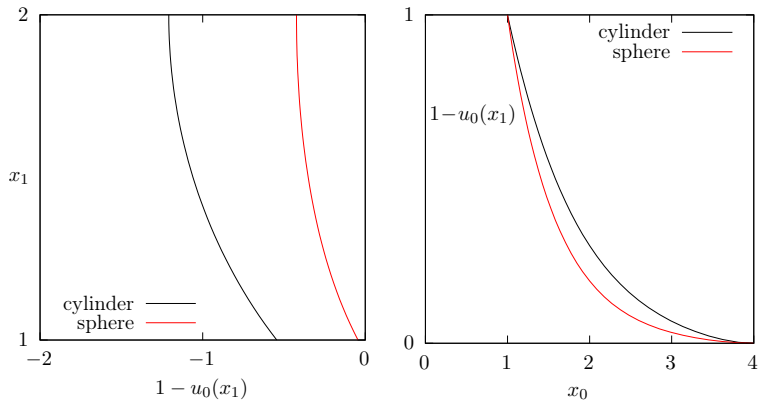


Figure 2.15: Slip boundary conditions for the flow around an obstacle: horizontal relative velocity along the two axis: (left) along Ox_1 on the top of the obstacle ; (right) along Ox_0 on the front.

Let us turn to the cut of the relative velocity along the horizontal and vertical axis, named respectively `axis` and `vaxis` by the mesh:

```
field obstacle-zr.field -domain vaxis -comp 0 -gnuplot -elevation
field obstacle-zr.field -domain  axis -comp 0 -gnuplot -elevation
```

Observe on Fig. 2.15.left that the relative velocity is negative on the top of the obstacle. Indeed, when the obstacle moves right, the fluid moves from the front of the obstacle, rotates around the obstacle and goes to the back. Thus, the fluid moves in the negatives direction when it is on the top of the obstacle. Also, observe that the fluid is more accelerated when it flows around a cylinder than around a sphere. Fig. 2.15.right shows that the perturbation caused by the fluid decreases faster for a sphere than for a cylinder.

2.4 Time-dependent problems

2.4.1 The heat equation

Formulation

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

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

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

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

Approximation

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

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

where $u^n \approx u(n\Delta t)$ and $u^{(0)} = 0$. The variational formulation of the time-discretized problem writes:

$(VF)_n$: *Let u^n being known, find $u^{n+1} \in H_0^1(\Omega)$ such that*

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

where

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

This is a Poisson-like problem. The discretization with respect to space of this problem is similar to those presented in section 1.1.1, page 14.

File 2.17: `heat.cc`

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

Comments

Note the use of the `branch` class:

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

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

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

is equivalent to the formatted output

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

How to run the program

We assume that the previous code is contained in the file '`heat.cc`'. Then, compile the program as usual (see page 16):

```
make heat
```

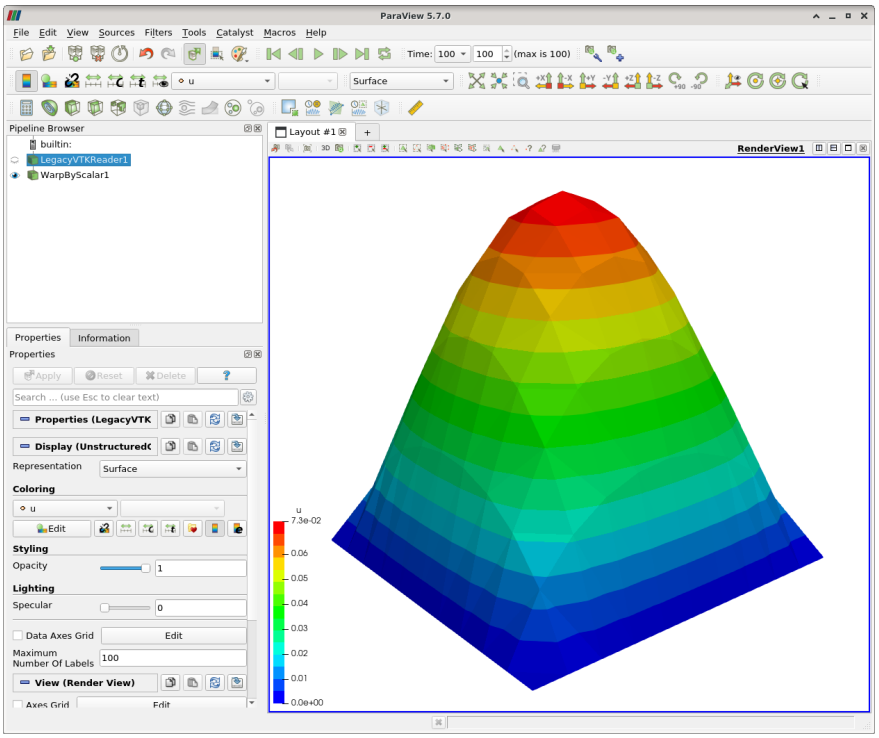


Figure 2.16: Animation of the solution of the heat problem.

For a one dimensional problem, enter the commands:

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

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

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

For a tridimensional one:

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

How to run the animation

```
branch line.branch
```

A `gnuplot` window appears. Enter `q` to exit the window. For a bidimensional case, simply enter:

```
branch square.branch -elevation
```

A window appears, that looks like a video player. Then, click on the video play button, at the top of the window.

To generate an animation file, go to the `File->save animation` menu and enter as file name `square` and as file type `avi`. The animation file `square.avi` can now be started from any video player, such as `vlc`:

```
vlc square.avi
```

For the tridimensional case, the animation feature is similar:

```
branch box.branch
branch box.branch -volume
```

2.4.2 The convection-diffusion problem

Formulation

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

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

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

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

Time approximation

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

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

where $\Delta t > 0$, $\phi^n \approx \phi(n\Delta t)$ and $\phi^{(0)} = \phi_0$.
Let $t_n = n\Delta t$, $n \geqslant 0$. The term $X^n(x)$ is the position at t_n of the particle that is in x at t_{n+1} and is transported by $\mathbf{u}^n$. Thus, $X^n(x) = X(t_n, x)$ where $X(t,x)$ is the solution of the differential equation

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

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. Pironneau, 1988), and is known as the method of characteristic in the finite difference context and as the Lagrange-Galerkin in the finite element one. The efficient evaluation of $\phi_h \circ X^n(x)$ in an unstructured mesh involves a hierarchical d-tree (quadtree, octree) data structure for the localization of the element K of the mesh that contains x. When $d = 3$ requires also sophisticated geometric predicates to test whether $x \in K$ without rounding errors, and avoid to conclude that no elements contains a point x close to ∂K up to rounding errors. This problems is addressed in **Rheolef** based on the `cgal` library.
The following code implements the classical rotating Gaussian hill test case (see e.g. Rui and Tabata, 2001).

File 2.18: `convect.cc`

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

Comments

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

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

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

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

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

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

A test case is described by Pironneau and Tabata [2010]: we take $\Omega =]-2,2[^d$ and $T = 2\pi$. This problem provides an example for a convection-diffusion equation and a known analytical solution:

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

where $\lambda = 4\nu/t_0 \geqslant 0$ with $t_0 > 0$ and $\nu \geqslant 0$, $x_0(t)$ is the moving center of the hill and $r(t) = 1/(t_0 + 4\nu t)$. The source term is time-dependent: $\sigma(t) = \lambda - 2d\nu r(t)$ and has been adjusted such

that the right-hand side is zero. The moving center of the hill $x_0(t)$ is associated to the velocity field $\mathbf{u}(t,x)$ as:

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

File 2.19: `rotating-hill.h`

```
struct u {
  point operator() (const point & x) const {
    return (d == 1) ? point(u0) : point(x[1], -x[0]); }
  u (size_t d1) : d(d1), u0 (0.5/acos(Float(-1))) {}
  protected: size_t d; Float u0;
};
struct phi {
  Float operator() (const point& x) const {
    return exp(-4*nu*(t/t0) - dist2(x,x0t())/(t0+4*nu*t)); }
  phi (size_t d1, Float nu1, Float t1=0) : d(d1), nu(nu1), t(t1),
    u0 (0.5/acos(Float(-1))), x0(-0.5,0) {}
  static Float sigma(size_t d, Float nu1, Float t=0) {
    return 4*nu1/t0 - 2*d*nu1/(t0 + 4*nu1*t); }
  point x0t() const {
    if (d == 1) return point(x0[0] + u0*t);
    return point( x0[0]*cos(t) + x0[1]*sin(t),
                  -x0[0]*sin(t) + x0[1]*cos(t)); }
  point d_x0t_dt() const {
    if (d == 1) return point(u0);
    return point(-x0[0]*sin(t) + x0[1]*cos(t),
                  -x0[0]*cos(t) - x0[1]*sin(t)); }
  protected: size_t d; Float nu, t, u0; point x0;
  static constexpr Float t0 = 0.2;
};
```

Note the use of a class-function `phi` for the implementation of $\phi(t)$ as a function of x. Such programming style has been introduced in the *standard template library* [Musser and Saini, 1996b], which is a part of the standard `C++` library. By this way, for a given t, $\phi(t)$ can be interpolated as an usual function on a mesh.

How to run the program

We assume that the previous code is contained in the file '`convect.cc`'. Then, compile the program as usual (see page 16):

```
make convect
```

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

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

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

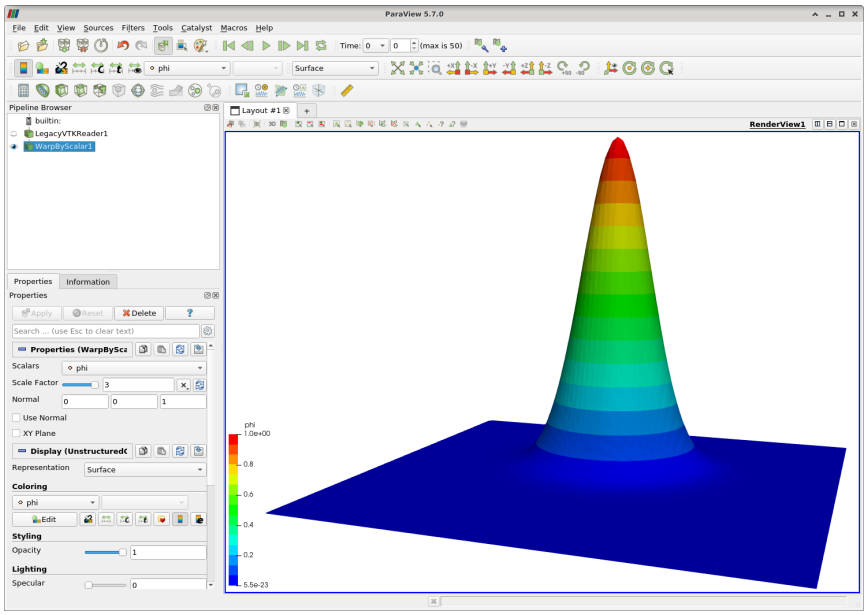


Figure 2.17: Animation of the solution of the rotating hill problem.

File 2.20: `convect_error.cc`

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

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

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

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

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

The instantaneous $L^2(\Omega)$ norm is printed at each time step and the total error in $L^2(]0,T[;L^2(\Omega))$ is finally printed at the end of the stream. A `P2` approximation can be used as well:

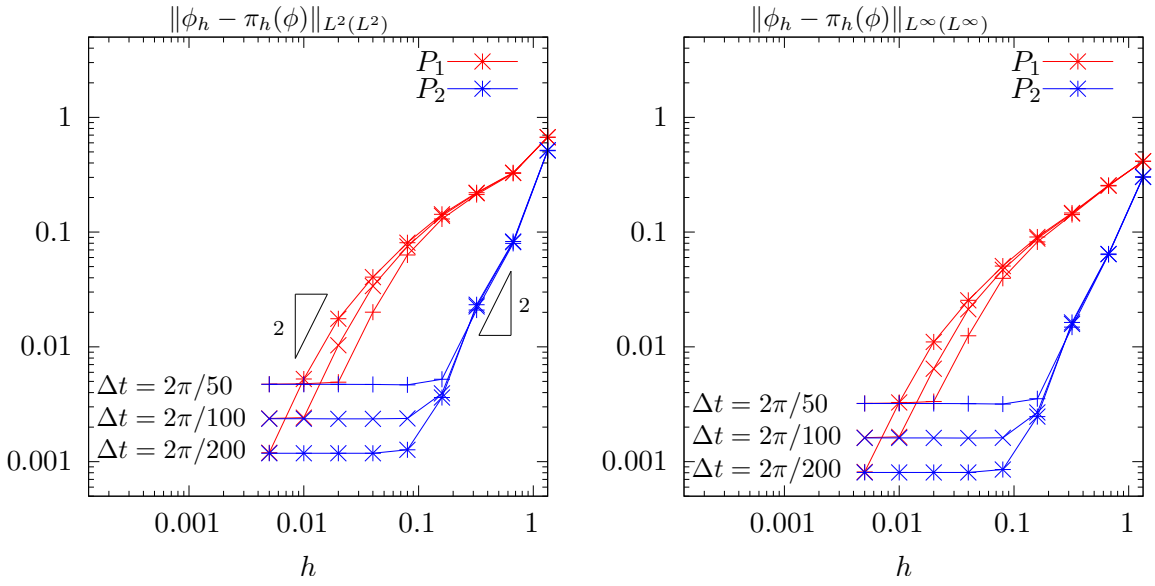


Figure 2.18: Diffusion-convection when $d = 1$ and $\nu = 10^{-2}$: convergence versus h and Δt for P_1 and P_2 elements: (left) in $L^2(L^2)$ norm; (right) in $L^\infty(L^\infty)$ norm.

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

On Fig. 2.18.left we observe the $L^2(L^2)$ convergence versus h for the P_1 and P_2 elements when $d = 1$: the errors reaches a plateau that decreases versus Δt. On Fig. 2.18.right the $L^\infty(L^\infty)$ norm of the error presents a similar behavior. Since the plateau are equispaced, the convergence versus Δt is of first order.

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

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

Let us turn to the two-dimensional test case:

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

The visualization and animation are similar to those of the head problem previously presented in paragraph 2.4.1. Go to the `WrapByScalar` entry in pipeline brower and adjust eventually the `scale factor`, e.g. to 3. Then, play the animation and observe the rotating hill. The result is shown on Fig. 2.17. The error analysis writes:

```
./convect_error < square2.branch > square2-cmp.branch
branch square2-cmp.branch -nofill -bw -elevation
```

From the `paraview` menu, you can visualize simultaneously both the approximate solution and the Lagrange interpolate of the exact one. First, in the pipeline brower go to the `WrapByScalar` entry and adjust the `scale factor`, e.g. to 3 and click on `Apply`. Next, go first to the `File->Open` menu and select in the `/tmp` the `square2-cmp-..vtk` and click on `Apply`. In the `Filter->Alphabetical` menu, select `WrapByScalar`. In the `Properties` panel, go to the `Scalars` entry and select `pi_h_phi`, to the `coloring` entry and select also `pi_h_phi`, adjust the `scale factor` to 3. and click on `Apply`. Next, in the same panel, in the `Representation` entry, choose `wireframe`. You are ready to click on the video play button, at the top of the window. Observe the difference between the solution and its approximation. See also the `paraview` documentation for more. For serious problem, the characteristic method has been superseded by the discontinuous Galerkin one, that will be presented in chapter 4, page 147. You are strongly encouraged to definitively turn to discontinuous Galerkin method for convection dominant and pure transport problems.
Finally, the three-dimensional case:

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

The visualization is similar to the two-dimensional case:

```
branch cube2.branch
branch cube2.branch -volume
```

2.5 The Navier-Stokes equations

Formulation

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

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

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

where $D(\mathbf{u}) = (\boldsymbol{\nabla}\mathbf{u} + \boldsymbol{\nabla}\mathbf{u}^T)/2$. This nonlinear problem is the natural extension of the linear Stokes problem, as presented in paragraph 2.5, page 84. The boundaries are represented on Fig. 2.1, page 44.

Time approximation

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

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

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

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

where, following [Boukir et al., 1997, Fourestey and Piperno, 2004]:

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

It is a second order extension of the method previously introduced in paragraph 2.4.2 page 79. The scheme defines a second order recurrence for the sequence $(\mathbf{u}^n)_{n\geqslant -1}$, that starts with $\mathbf{u}^{-1}=\mathbf{u}^0=0$.

Variational formulation

The variational formulation of this problem expresses:

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

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

where

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

and

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

and $b(.,.)$ and $\mathbf{V}(\alpha)$ was already introduced in paragraph 2.1.4, page 53, while studying the Stokes problem.

Space approximation

The Hood and Taylor [1973] finite element approximation of this generalized Stokes problem was also considered in paragraph 2.1.4, page 53. We introduce a mesh $\mathcal{T}_h$ of Ω and the finite dimensional spaces $\mathbf{X}_h$, $\mathbf{V}_h(\alpha)$ and Q_h. The approximate problem writes:

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

$$\begin{array}{lcccll} a(\mathbf{u}_h^{n+1},\mathbf{v}) & + & b(\mathbf{v},p_h^{n+1}) & = & m(\mathbf{f}_h^n,\mathbf{v}), & \forall \mathbf{v}\in\mathbf{V}_h(0), \\ b(\mathbf{u}_h^{n+1},q) & & & = & 0, & \forall q\in Q_h. \end{array} \tag{2.7}$$

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 2.21: navier_stokes_solve.icc

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

Comments

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

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

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

File 2.22: navier_stokes_cavity.cc

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

File 2.23: navier_stokes_criterion.icc

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

Comments

The code performs a computation by using adaptive mesh refinement, in order to capture recirculation zones. The `adapt_option` declaration is used by `rheolef` to send options to the mesh generator. The code reuse the file '`cavity.h`' introduced page 54. This file contains two functions that defines boundary conditions associated to the cavity driven problem.

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

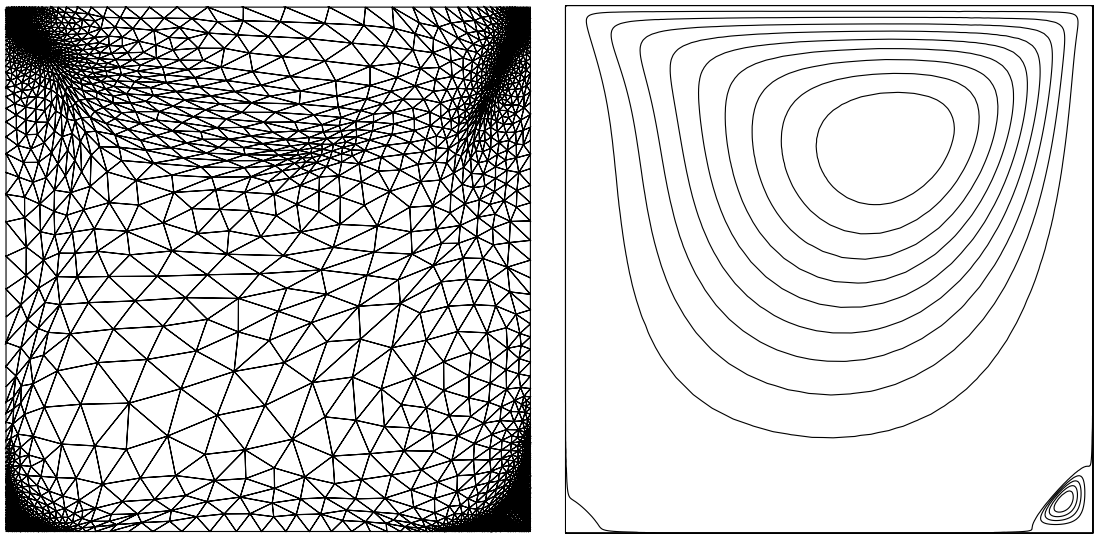
$$c_h = (Re|\mathbf{u}_h|^2 + 2|D(\mathbf{u}_h)|^2)^{1/2}$$

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

How to run the program

$Re = 100$: 4804 elements, 2552 vertices

$\psi_{max} = 9.5 \times 10^{-6}, \psi_{min} = -0.103$



$Re = 400$: 5233 elements, 2768 vertices

$\psi_{max} = 6.4 \times 10^{-4}, \psi_{min} = -0.111$

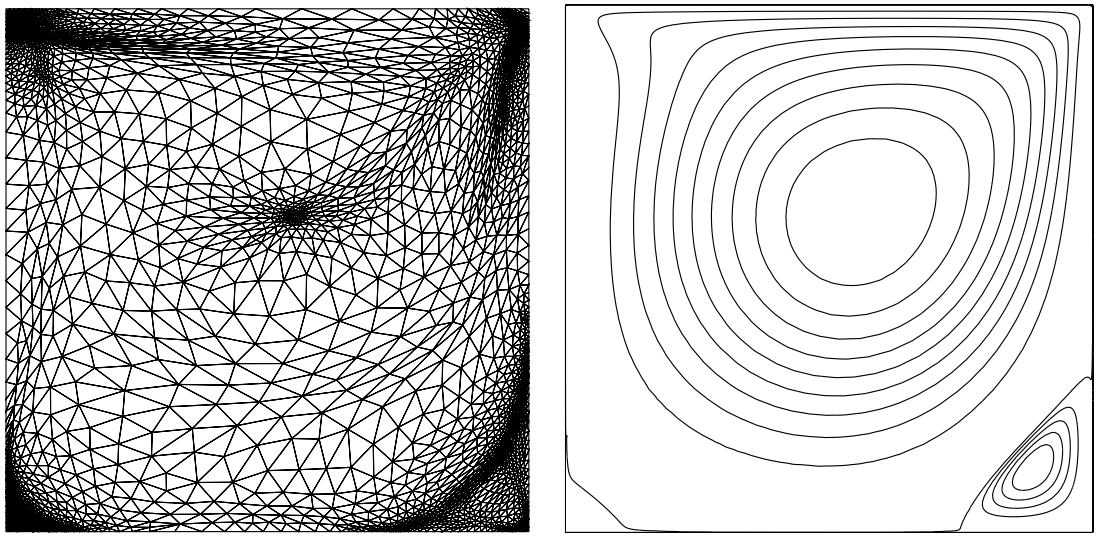


Figure 2.19: Meshes and stream functions associated to the solution of the Navier-Stokes equations for $Re = 100$ (top) and $Re = 400$ (bottom).

The mesh loop adaptation is initiated from a `bamg` mesh (see also appendix A.2.1).

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

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

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

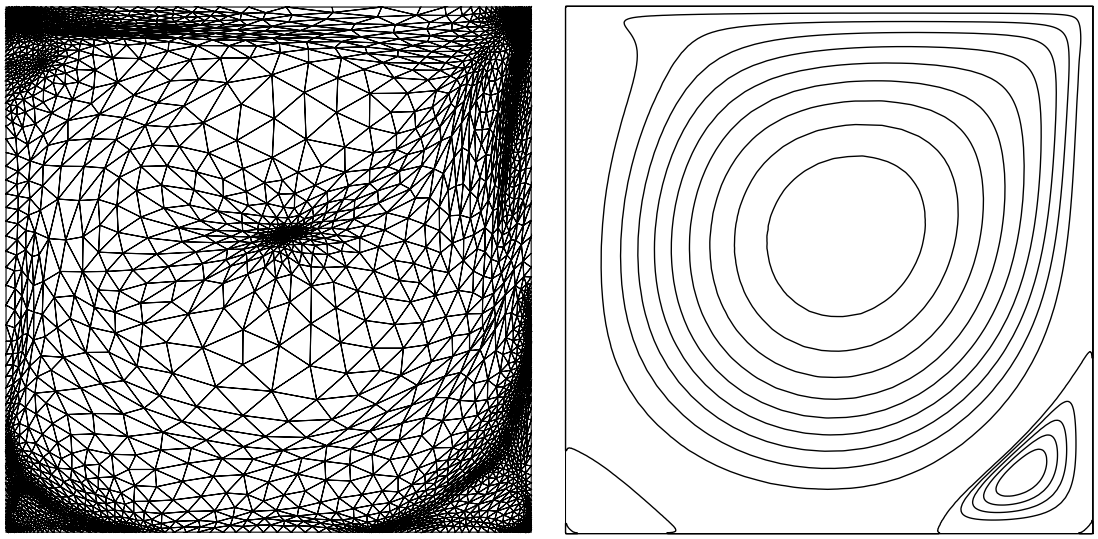


Figure 2.20: Meshes and stream functions associated to the solution of the Navier-Stokes equations for $Re = 1000$.

```
make navier_stokes_cavity
time mpirun -np 8 ./navier_stokes_cavity square.geo 100
```

The program performs a computation with $Re = 100$. By default the time step is $\Delta t = 0.05$ and the computation loops for five mesh adaptations. At each time step, the program prints an approximation of the time derivative, and stops when a stationary solution is reached. The `mpirun -np 8` prefix allows a parallel and distributed run while the `time` one returns the `real` and the `user` times used by the computation. The speedup could be estimated here by the ratio `user`/`real`: it is ideally close to the number of processors. These prefixes are optional and you can omit the `mpirun` one if you are running with a sequential installation of **Rheolef**. Then, we visualize the '`square-005.geo`' adapted mesh and its associated solution:

```
geo   square-005.geo
field square-005.field.gz -velocity
```

The representation of the stream function writes:

```
make streamf_cavity
zcat square-005.field.gz | ./streamf_cavity | field -bw -n-iso-negative 10 -
```

The programs '`streamf_cavity.cc`', already introduced page 59, is here reused. The last options of the `field` program draws isocontours of the stream function using lines, as shown on Fig. 2.19. 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
```

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

```
field square-005.field.gz -comp 0 -cut -normal -1 0 -origin 0.5 0   -gnuplot
field square-005.field.gz -comp 1 -cut -normal  0 1 -origin 0   0.5 -gnuplot
```

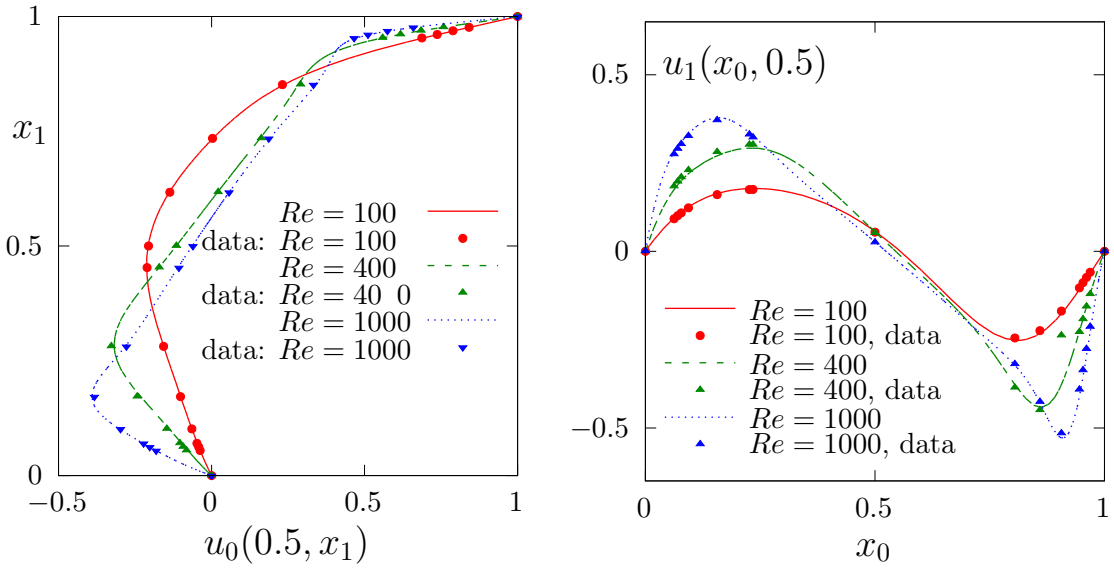


Figure 2.21: Navier-Stokes: velocity profiles along lines passing thought the center of the cavity, compared with data from Ghia et al. [1982]: (a) u_0 along the vertical line; (b) u_1 along the horizontal line line.

Fig. 2.21 compare the cuts with data from Ghia et al. [1982], table 1 and 2 (see also Gupta and Kalita, 2005). Observe that the solution is in good agreement with these previous computations.

Re		x_c	y_c	$-\psi_{\min}$	$\psi_{\max}$
100	present	0.613	0.738	0.103	9.5×10^{-6}
	Labeur and Wells [2007]	0.608	0.737	0.104	-
	Donea and Huerta [2003]	0.62	0.74	0.103	-
400	present	0.554	0.607	0.111	5.6×10^{-4}
	Labeur and Wells [2007]	0.557	0.611	0.115	-
	Donea and Huerta [2003]	0.568	0.606	0.110	-
1000	present	0.532	0.569	0.117	1.6×10^{-3}
	Labeur and Wells [2007]	0.524	0.560	0.121	-
	Donea and Huerta [2003]	0.540	0.573	0.110	-

Figure 2.22: Cavity flow: primary vortex position and stream function value.

Finally, table 2.22 compares the primary vortex position and its associated stream function value. Note also the good agreement with previous simulations. The stream function extreme values are obtained by:

```
zcat square-005.field.gz | ./streamf_cavity | field -min -
zcat square-005.field.gz | ./streamf_cavity | field -max -
```

The maximal value has not yet been communicated to our knowledge and is provided in table 2.22 for cross validation purpose. The small program that computes the primary vortex position is showed below.

```
make vortex_position
zcat square-005.field.gz | ./streamf_cavity | ./vortex_position
```

File 2.24: vortex_position.cc

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

For higher Reynolds number, Shen [1991] showed in 1991 that the flow converges to a stationary state for Reynolds numbers up to 10 000; for Reynolds numbers larger than a critical value $10\,000 < Re_1 < 10\,500$ and less than another critical value $15\,000 < Re_2 < 16\,000$, these authors founded that the flow becomes periodic in time which indicates a Hopf bifurcation; the flow loses time periodicity for $Re \geq Re_2$. Ould Salihi [1998] founded a loss of stationnarity between 10 000 and 20 000. Auteri, Parolini, and Quartapelle [2002] estimated the critical value for the apparition of the first instability to $Re_1 \approx 8018$. Erturk, Corke, and Gökçol [2005] computed steady driven cavity solutions up to $Re \leqslant 21\,000$. This result was infirmed by Gelhard, Lube, Olshanskii, and Starcke [2005]: these authors estimated Re_1 close to 8000, in agreement with Auteri, Parolini, and Quartapelle [2002]. The 3D driven cavity has been investigated in Minev and Ethier [1998] by the method of characteristic (see also Melchior, Legat, Van Dooren, and Wathen, 2012 for 3D driven cavity computations). In conclusion, the exploration of the driven cavity at large Reynolds number is a fundamental challenge in computational fluid dynamics.

Note that, instead of using a time-dependent scheme, that requires many time steps, it is possible to directly compute the stationary solution of the Navier-Stokes problem, thanks to a nonlinear solver. This alternative approach is presented in section 4.5, page 185, based on the discontinuous Galerkin method. The discontinuous Galerkin method is much more robust and accurate than the method of characteristics and is recomanded for serious problems.

Chapter 3

Advanced and highly nonlinear problems

3.1 Equation defined on a surface

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

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

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

$$u - \Delta_s u = f \text{ on } \Gamma \tag{3.1}$$

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

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

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

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

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

We also consider the following variant of this problem:

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

$$-\Delta_s u = f \text{ on } \Gamma \tag{3.2}$$

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

3.1.1 Approximation on an explicit surface mesh

The Helmholtz-Beltrami problem

Tanks to the surface Green formula (see appendix A.1.3), the variational formulation of problem $(P1)$ writes:
$(VF1)$: *find* $u \in H^1(\Gamma)$ *such that:*

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

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

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

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

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

The approximate problem writes:
$(VF1)_h$: *find* $u_h \in W_h$ *such that:*

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

File 3.1: `helmholtz_s.cc`

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

Comments

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

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

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

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

The right-hand-side does not involve any boundary term, since the surface Γ is closed: the boundary domain $\partial\Gamma = \emptyset$. As test problem, the surface Γ is the unit circle when $d = 2$ and the unit

sphere when $d = 3$. The data f has been chosen as in Deckelnick et al. [2009, p. 17]. This choice is convenient since the exact solution is known. Recall that the spherical coordinates (ρ, θ, ϕ) are defined from the Cartesian ones (x_0, x_1, x_2) by:

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

File 3.2: `sphere.icc`

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

How to run the program

The program compile as usual:

```
make helmholtz_s
```

A mesh of a circle is generated by:

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

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

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

The tridimensional case is similar:

```
mkgeo_ball -s -t 10 > sphere.geo
geo sphere.geo -stereo
```

```
./helmholtz_s sphere.geo P1 > sphere.field
field sphere.field
field sphere.field -stereo
```

The solution is represented on Fig .3.1.left.

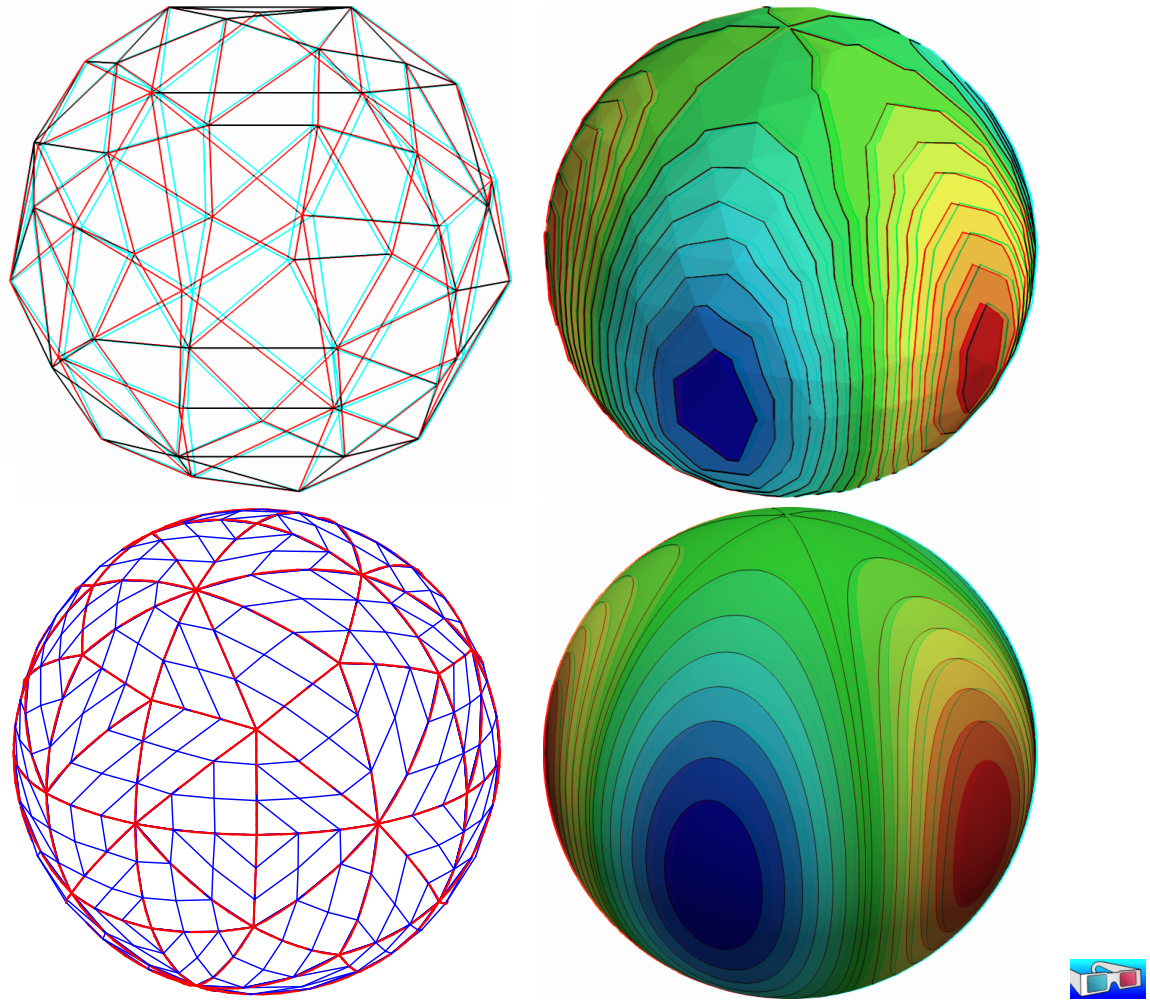


Figure 3.1: Helmholtz-Beltrami problem: high-order curved surface mesh and its corresponding isoparametric solution: (top) $order = 1$; (bottom) $order = 3$.

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

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

Observe the curved edges (see Fig .3.1). The `-subdivide` option allows a graphical representation of the curved edges by subdividing each edge in ten linear parts, since graphical softwares are not yet able to represent curved elements. The computation with the P_3 isoparametric approximation writes:

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

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

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

The solution is represented on Fig .3.1).right-bottom. The graphical representation is not yet able to represent the high-order approximation: each elements is subdivided and a piecewise linear representation is used in each sub-elements.

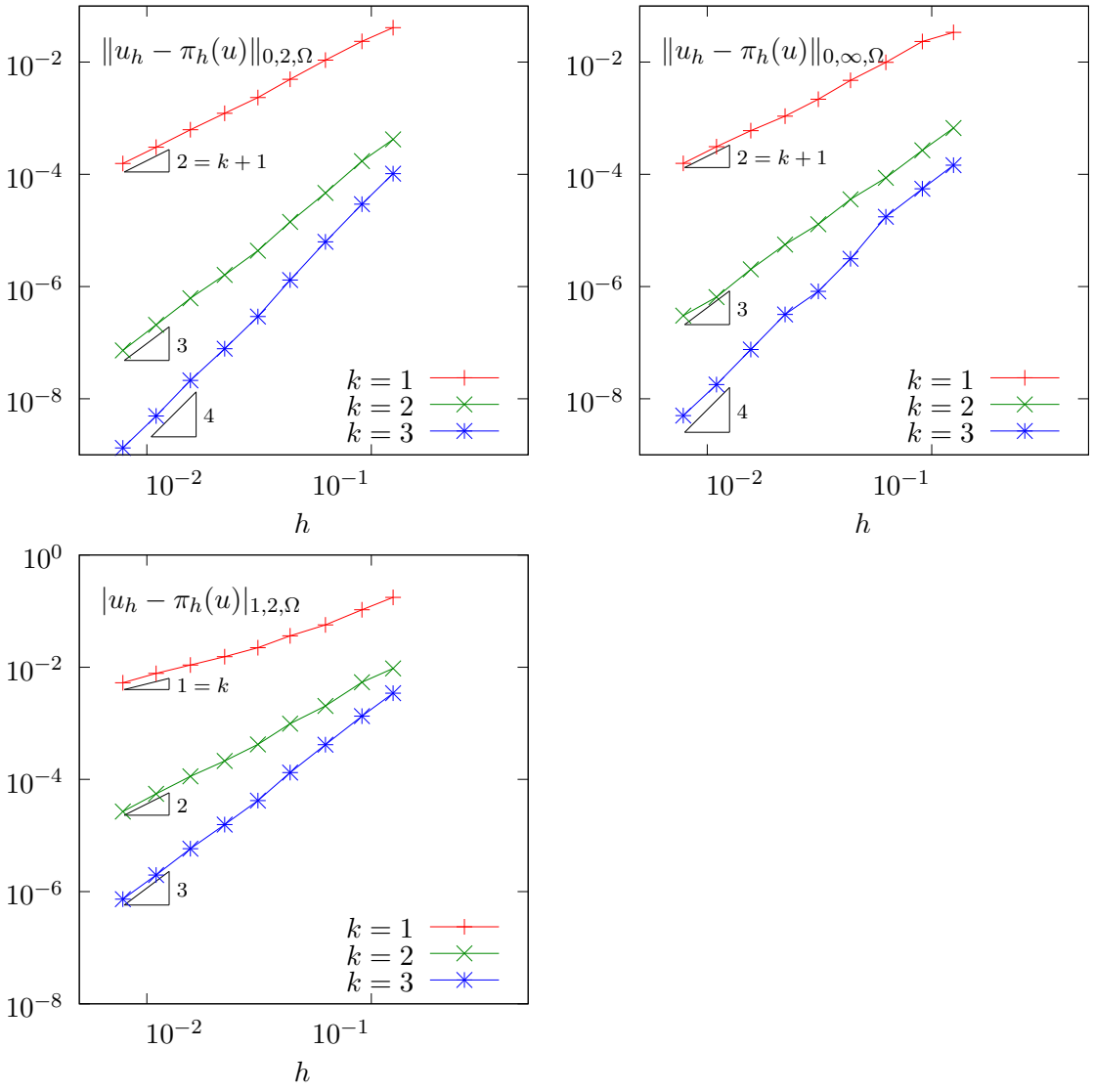


Figure 3.2: Curved non-polynomial surface: error analysis in L^2, L^∞ and H^1 norms.

Since the exact solution is known, the error can be computed: this is done by the program `helmholtz_s_error.cc`. This file is not presented here, as it is similar to some others examples, but can be founded in the **Rheolef** example directory. Figure 3.2 plots the error in various norms versus element size for different isoparametric approximations.

The Laplace-Beltrami problem

This problem has been introduced in (3.2), page 93. While the treatment of the Helmholtz-Beltrami problem was similar to the Helmholtz problem with Neumann boundary conditions, here, the treatment of the Laplace-Beltrami problem is similar to the Laplace problem with Neumann boundary conditions: see section 1.4, page 35. Note that for both problems, the solution is defined up to a constant. Thus, the linear problem has a singular matrix. The '`laplace_s.cc`' code is similar to the '`neumann-laplace.cc`' one, as presented in section 1.4. The only change lies one the definition of the right-hand side.

File 3.3: `laplace_s.cc`

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
#include "torus.icc"
int main (int argc, char**argv) {
  environment rheolef (argc, argv);
  geo gamma (argv[1]);
  size_t d = gamma.dimension();
  space Wh (gamma, argv[2]);
  trial u (Wh); test v (Wh);
  form  a = integrate (dot(grad_s(u),grad_s(v)));
  field b = integrate(v);
  field lh = integrate (f(d)*v);
  form  A = {{    a,     b },
             { trans(b), 0 }};
  field Bh = {   lh,     0 };
  field Uh (Bh.get_space(), 0);
  A.set_symmetry(true);
  problem pa (A);
  pa.solve (Bh, Uh);
  dout << Uh[0];
}
```

File 3.4: torus.icc

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

As test problem, the surface Γ is the a torus when $d = 3$. The data f has been chosen as in Olshanskii et al. [2009, p. 3355]. This choice is convenient since the exact solution is known. Let R and r denotes the large and small torus radii, respectively. The torus coordinates (ρ, θ, ϕ) are defined linked to the Cartesian ones by:

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

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

How to run the program ?

The surface mesh of the torus is generated by:

```
gmsh -2 torus.mshcad -format msh2 -o torus.msh
msh2geo torus.msh > torus.geo
geo torus.geo -stereo
```

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

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

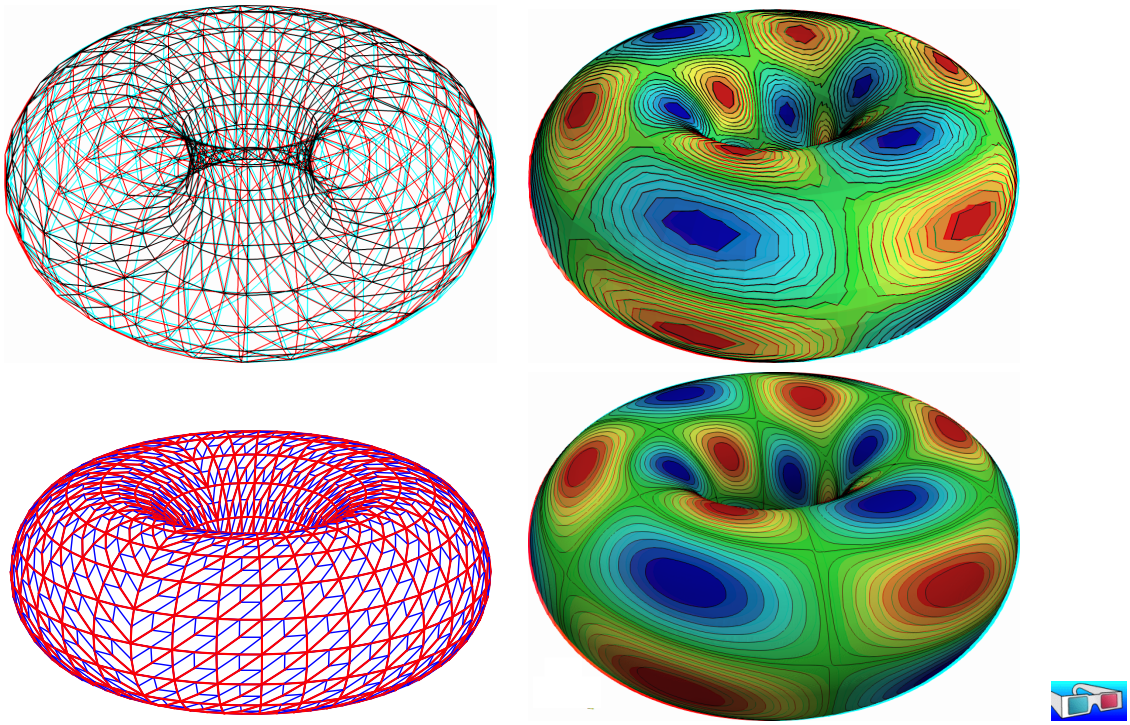


Figure 3.3: Laplace-Beltrami problem on a torus: high-order curved surface mesh and its corresponding isoparametric solution: (top) *order* = 1; (bottom) *order* = 2.

For a higher-order approximation:

```
gmsh -2 -order 2 torus.mshcad -format msh2 -o torus-P2.msh
msh2geo torus-P2.msh > torus-P2.geo
geo torus-P2.geo -gnuplot
./laplace_s torus-P2.geo P2 > torus-P2.field
field torus-P2.field -stereo
```

The solution is represented on Fig. 3.3. By editing '`torus.mshcad`' and changing the density of discretization, we can improve the approximate solution and converge to the exact solution. Due to a bug [Saramito, 2012b] in the current gmsh version 2.5.1 the convergence is not optimal $\mathcal{O}(h^k)$ for higher values of k.

3.1.2 Building a surface mesh from a level set function

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

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

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

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

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

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

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

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

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

File 3.5: `level_set_sphere.cc`

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

Comments

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

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

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

How to run the program ?

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

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

The computation of the previous paragraph can be reused:

```
./helmholtz_s circle.geo P1 | field -paraview -
```

Note that, while the bounding box mesh was uniform, the intersected mesh could present arbitrarily small edge length (see also Fig. 3.4):

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

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

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

While the bounding box mesh was uniform, the triangular elements obtained by intersecting the 3D bounding box mesh with the level set function can present arbitrarily irregular sizes and shapes (see also Fig. 3.4):

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

Nevertheless, and surprisingly, Olshanskii et al. [2012] recently showed that the finite element method converges on these irregular intersected families of meshes.

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

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

Note that the intersected mesh is also irregular:

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

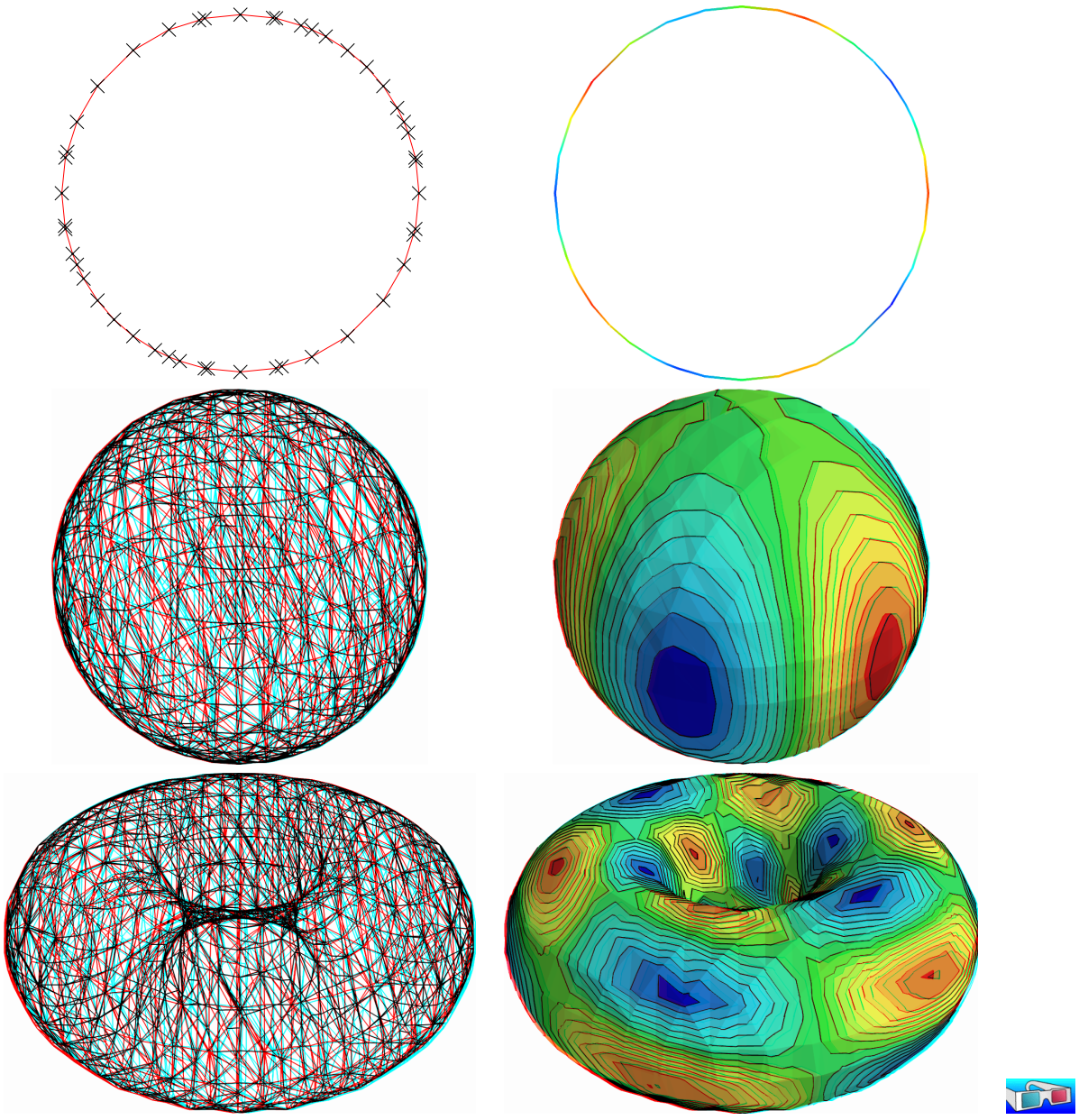


Figure 3.4: Building an explicit surface mesh from level set: (top) circle; (center) sphere; (bottom) torus.

3.1.3 The banded level set method

The banded level set method presents the advantages of the two previous methods without their drawback: it applies to very general geometries, as described by a level set funtion, and stronger convergence properties, as usual finite element methods. The previous intersection mesh can be circumvented by enlarging the surface Γ_h to a band β_h containing all the intersected elements of $\mathcal{T}_h$ (see Olshanskii et al., 2009, Abouorm, 2010, Dicko, 2011):

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

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

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

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

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

where, for all $u, v \in B_h$,

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

for all $u_h, v_h \in B_h$. Note that while u_h and v_h are defined over β_h, the summations in the variational formulations are restricted only to $\Gamma_h \subset \beta_h$.

File 3.6: `helmholtz_band_iterative.cc`

```
#include "rheolef.h"
using namespace std;
using namespace rheolef;
#include "sphere.icc"
int main (int argc, char**argv) {
  environment rheolef(argc, argv);
  geo lambda (argv[1]);
  size_t d = lambda.dimension();
  space Xh (lambda, "P1");
  field phi_h = lazy_interpolate(Xh, phi);
  band gamma_h (phi_h);
  space Bh (gamma_h.band(), "P1");
  trial u (Bh); test v (Bh);
  form  a  = lazy_integrate (gamma_h, u*v + dot(grad_s(u),grad_s(v)));
  field lh = lazy_integrate (gamma_h, f(d)*v);
  field uh (Bh,0);
  solver_option sopt;
  sopt.max_iter = 10000;
  sopt.tol = 1e-10;
  minres (a.uu(), uh.set_u(), lh.u(), eye(), sopt);
  dout << catchmark("phi") << phi_h
       << catchmark("u")   << uh;
}
```

Comments

The band is build directly from the level set function as:

```
band gamma_h (phi_h);
```

The band structure is a small class that groups the surface mesh Γ_h, available as `gamma_h.level_set()`, and the β_h mesh, available as `gamma_h.band()`. It also manages some correspondence between both meshes. Then, the space of piecewise affine functions over the band is introduced:

```
space Bh (gamma_h.band(), "P1");
```

Next, the bilinear form is computed by using the `integrate` function, with the band `gamma_h` as a domain-like argument:

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

The right-hand side also admits the `gamma_h` argument:

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

Recall that summations for both forms and right-hand side will be performed on Γ_h, represented by `gamma_h.level_set()`, while the approximate functional space is B_h. Due to this summation on Γ_h instead of β_h, the matrix of the system is singular [Olshanskii et al., 2009, Olshanskii and Reusken, 2010, Abouorm, 2010] and the MINRES algorithm has been chosen to solve the linear system:

```
minres (a.uu(), uh.set_u(), lh.u(), eye(), sopt);
```

The `eye()` argument represents here the identity preconditioner, i.e. no preconditioner at all. It has few influence of the convergence properties of the matrix and could be replaced by another simple one: the diagonal of the matrix `diag(a.uu())` without sensible gain of performance:

```
minres (a.uu(), uh.set_u(), lh.u(), diag(a.uu()), sopt);
```

See the reference manual for more about `minres`, e.g. on the **Rheolef** web site or as unix manual

```
man minres
```

How to run the program

The compilation and run writes:

```
make helmholtz_band_iterative
mkgeo_grid -T 20 -a -2 -b 2 -c -2 -d 2 -f -2 -g 2 > cube-20.geo
./helmholtz_band_iterative cube-20.geo > sphere-band.field
```

The run generates also two meshes (see Fig. 3.5): the intersection mesh and the band around it. The solution is here defined on this band: this extension has no interpretation in terms of the initial problem and can be restricted to the intersection mesh for visualization purpose:

```
make proj_band
./proj_band < sphere-band.field | field -
```

The '`proj_band.cc`' is presented below. The run generates also the Γ_h mesh (see Fig. 3.5), required for the visualization. The two-dimensional case is obtained simply by replacing the 3D bounding box by a 2D one:

```
mkgeo_grid -t 20 -a -2 -b 2 -c -2 -d 2 > square-20.geo
./helmholtz_band_iterative square-20.geo > circle-band.field
./proj_band < circle-band.field | field -paraview -
./proj_band < circle-band.field | field -elevation -bw -stereo -
```

File 3.7: proj_band.cc

```
#include "rheolef.h"
using namespace std;
using namespace rheolef;
int main (int argc, char**argv) {
  environment rheolef (argc, argv);
  field phi_h;
  din >> catchmark("phi") >> phi_h;
  const space& Xh = phi_h.get_space();
  band gamma_h (phi_h);
  space Bh (gamma_h.band(), "P1");
  field uh(Bh);
  din >> catchmark("u") >> uh;
  space Wh (gamma_h.level_set(), "P1");
  gamma_h.level_set().save();
  dout << lazy_interpolate (Wh, uh);
}
```

3.1.4 Improving the banded level set method with a direct solver

The iterative algorithm previously used for solving the linear system is not optimal: for 3D problems on a surface, the bidimensionnal connectivity of the sparse matrix suggests that a direct sparse factorization would be much more efficient.

Recall that $\phi_h = 0$ on Γ_h. Thus, if $u_h \in B_h$ is solution of the problem, then $u_h + \alpha\phi_{h|\beta_h} \in B_h$ is also solution for any $\alpha \in \mathbb{R}$, where $\phi_{h|\beta_h} \in B_h$ denotes the restriction of the level set function $\phi_h \in X_h$ on the band β_h. Thus there is multiplicity of solutions and the matrix of the problem is singular. The direct resolution is still possible on a modified linear system with additional constraints in order to recover the unicity of the solution. We impose the constraint that the solution u_h should be othogonal to $\phi_{h|\beta_h} \in B_h$. In some special cases, the band is composed of several connected components (see Fig. 3.6): this appends when a vertex of the bounding box mesh belongs to Γ_h. In that case, the constraint should be expressed on each connected component. Fig. 3.6 shows also the case when a full side of an element is included in Γ_h: such an element of the band is called *isolated*.

File 3.8: helmholtz_band.cc

```
#include "rheolef.h"
using namespace std;
using namespace rheolef;
#include "sphere.icc"
int main (int argc, char**argv) {
  environment rheolef(argc, argv);
  geo lambda (argv[1]);
  size_t d = lambda.dimension();
  space Xh (lambda, "P1");
  field phi_h = lazy_interpolate(Xh, phi);
  band gamma_h (phi_h);
  field phi_h_band = phi_h [gamma_h.band()];
  space Bh (gamma_h.band(), "P1");
  Bh.block ("isolated");
  Bh.unblock ("zero");
  trial u (Bh); test v (Bh);
  form  a  = lazy_integrate (gamma_h, u*v + dot(grad_s(u),grad_s(v)));
  field lh = lazy_integrate (gamma_h, f(d)*v);
  vector<field> b (gamma_h.n_connected_component());
  vector<Float> z (gamma_h.n_connected_component(), 0);
  for (size_t i = 0; i < b.size(); i++) {
    const domain& cci = gamma_h.band() ["cc"+to_string(i)];
    field phi_h_cci (Bh, 0);
    phi_h_cci [cci] = phi_h_band [cci];
    b[i] = phi_h_cci;
  }
  form  A = {{ a, trans(b) },
             { b,    0     }};
  field Fh = { lh,   z     };
  A.set_symmetry(true);
  problem pa (A);
  field Uh (Fh.get_space(), 0);
  pa.solve (Fh, Uh);
  dout << catchmark("phi") << phi_h
       << catchmark("u")   << Uh[0];
}
```

Comments

The management of the special sides and vertices that are fully included in Γ_h is perfomed by:

```
  Bh.block ("isolated");
  Bh.unblock ("zero");
```

The addition of linear constraints is similar to the 'neumann-laplace.cc' code, as presented in section 1.4:

```
  form A = {{ a, trans(b)},
            { b,    0    }};
```

Here `b` is a `vector<field>`, i.e. a vector of linear constraints, one per connected component of the band β_h.

How to run the program

The commands are similar to the previous iterative implementation, just replacing `helmholtz_band_iterative` by `helmholtz_band`.

This approach could be also adapted to the Laplace-Beltrami problem on the torus.

File 3.9: laplace_band.cc

```
#include "rheolef.h"
using namespace std;
using namespace rheolef;
#include "torus.icc"
int main (int argc, char**argv) {
  environment rheolef(argc, argv);
  geo lambda (argv[1]);
  size_t d = lambda.dimension();
  space Xh (lambda, "P1");
  field phi_h = lazy_interpolate(Xh, phi);
  band gamma_h (phi_h);
  field phi_h_band = phi_h [gamma_h.band()];
  space Bh (gamma_h.band(), "P1");
  Bh.block ("isolated");
  Bh.unblock ("zero");
  trial u (Bh); test v (Bh);
  form  a  = lazy_integrate (gamma_h, dot(grad_s(u),grad_s(v)));
  field c  = lazy_integrate (gamma_h, v);
  field lh = lazy_integrate (gamma_h, f(d)*v);
  vector<field> b (gamma_h.n_connected_component());
  vector<Float> z (gamma_h.n_connected_component(), 0);
  for (size_t i = 0; i < b.size(); i++) {
    const domain& cci = gamma_h.band() ["cc"+to_string(i)];
    field phi_h_cci (Bh, 0);
    phi_h_cci [cci] = phi_h_band [cci];
    b[i] = phi_h_cci;
  }
  form  A = {{    a,  trans(b), c },
             {    b,      0,    0 },
             { trans(c), 0,    0 }};
  field Fh = {    lh,    z,     0 };
  field Uh (Fh.get_space(), 0);
  A.set_symmetry(true);
  problem pa (A);
  pa.solve (Fh, Uh);
  dout << catchmark("phi") << phi_h
       << catchmark("u")   << Uh[0];
}
```

Comments

The code is similar to the previous one `helmholtz_band.cc`. Since the solution is defined up to a constant, an additional linear constraint has to be inserted:

$$\int_{\Gamma_h} u_h \, \mathrm{d}x = 0$$

This writes:

```
  field c  = integrate (gamma_h, v);
  form  A = {{    a,  trans(b), c },
             {    b,      0,    0 },
             { trans(c), 0,    0 }};
```

How to run the program

```
make laplace_band
mkgeo_grid -T 20 -a -2 -b 2 -c -2 -d 2 -f -2 -g 2 > cube-20.geo
./laplace_band cube-20.geo > torus-band.field
./proj_band < torus-band.field | field -stereo -
```

The solution is represented on Fig. 3.5.bottom.

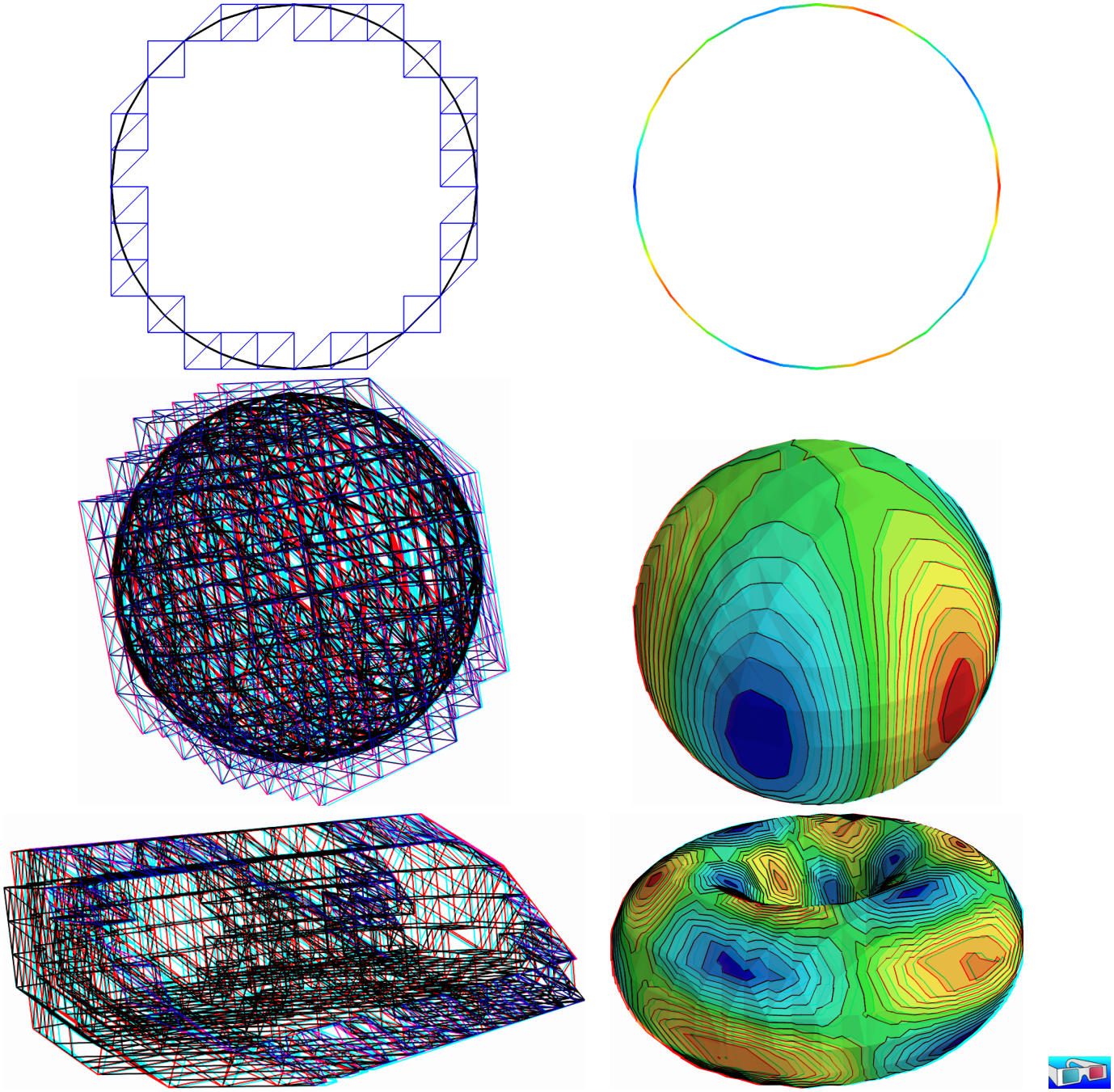


Figure 3.5: The banded level set method: (top) circle; (center) sphere; (bottom) torus.

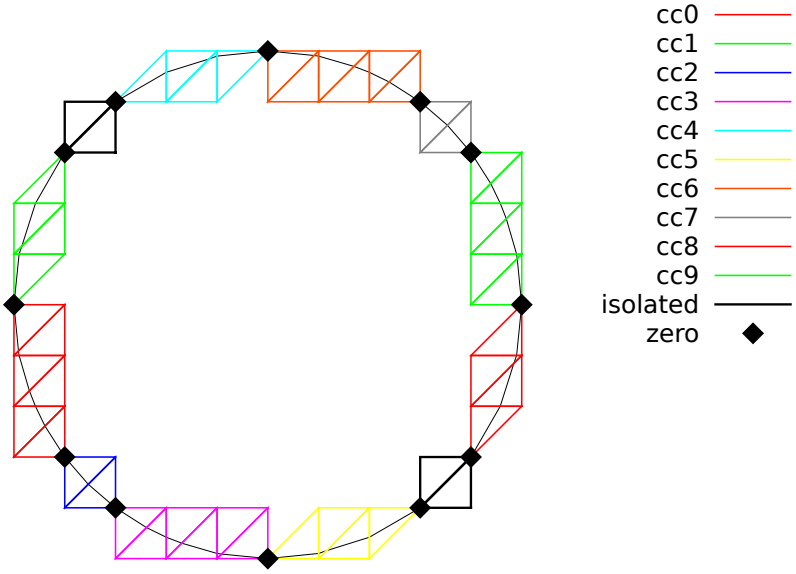


Figure 3.6: The banded level set method: the band is composed of several connected components.

3.2 The highly nonlinear p-laplacian problem

3.2.1 Problem statement

Let us consider the classical p-Laplacian problem with homogeneous Dirichlet boundary conditions in a domain bounded $\Omega \subset \mathbb{R}^d$, $d = 1, 2, 3$:

(P): find u, defined in Ω such that:

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

where $\eta : z \in \mathbb{R}^+ \longmapsto z^{\frac{p-2}{2}} \in \mathbb{R}^+$. Several variants of the η can be considered: see Saramito [2016b] for practical and useful examples: this problem represents a pipe flow of a non-Newtonian power-law fluid. Here $p \in]1, +\infty[$ and f are known. For the computational examples, we choose $f = 1$. When $p = 2$, this problem reduces to the linear Poisson problem with homogeneous Dirichlet boundary conditions. Otherwise, for any $p > 1$, the nonlinear problem is equivalent to the following minimization problem:

(MP): find $u \in W_0^{1,p}(\Omega)$ such that:

$$u = \underset{v \in W_0^{1,p}(\Omega)}{\arg\min} \ \frac{1}{2}\int_\Omega H\left(|\nabla v|^2\right) \mathrm{d}x - \int_\Omega f\, v \,\mathrm{d}x,$$

where H denotes the primitive of η:

$$H(z) = \int_0^z \eta(z)\,\mathrm{d}z = \frac{2z^p}{p}$$

Here $W_0^{1,p}(\Omega)$ denotes the usual Sobolev spaces of functions in $W^{1,p}(\Omega)$ We also assume that $f \in W^{-1,p}(\Omega)$, where $W_0^{-1,p}(\Omega)$ denotes the dual space of $W_0^{1,p}(\Omega)$ that vanishes on the boundary [Brezis, 1983, p. 118]. The variational formulation of this problem expresses:

(VF): find $u \in W_0^{1,p}(\Omega)$ such that:

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

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

$$a(u_0; u, v) = \int_\Omega \eta\left(|\nabla u_0|^2\right) \nabla u.\nabla v \,\mathrm{d}x, \quad \forall u, v \in W_0^{1,p}(\Omega) \tag{3.3}$$

$$l(v) = \int_\Omega f\, v \,\mathrm{d}x, \quad \forall u, v \in L^2(\Omega) \tag{3.4}$$

The quantity $a(u; u, u)^{1/p} = \|\boldsymbol{\nabla} u\|_{0,p,\Omega}$ induces a norm in $W_0^{1,p}$, equivalent to the standard norm. The form $a(.; ., .)$ is bilinear with respect to the two last variable and is related to the *energy* form.

3.2.2 The fixed-point algorithm

Principe of the algorithm

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

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

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

and then set

$$u^{(n+1)} = \omega u^* + (1-\omega) * u^{(n)}$$

Here $\omega > 0$ is the relaxation parameter: when $\omega = 1$ we obtain the usual un-relaxed fixed point algorithm. For stiff nonlinear problems, we will consider the under-relaxed case $0 < \omega < 1$. Let $u^{(n+1)} = G\left(u^{(n)}\right)$ denotes the operator that solve the previous linear subproblem for a given $u^{(n)}$. Since the solution u satisfies $u = G(u)$, it is a fixed-point of G.

Let us introduce a mesh $\mathcal{T}_h$ of Ω and the finite dimensional space X_h of continuous piecewise polynomial functions and V_h, the subspace of X_h containing elements that vanishes on the boundary of Ω:

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

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

$$a\left(u_h^{(n)}; u_h^*, v_h\right) = l(v_h),\ \forall v_h \in V_h$$

By developing u_h^* on a basis of V_h, this problem reduces to a linear system.

File 3.10: p_laplacian_fixed_point.cc

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
#include "eta.h"
#include "dirichlet.icc"
int main(int argc, char**argv) {
  environment rheolef (argc,argv);
  geo omega (argv[1]);
  Float  eps    = std::numeric_limits<Float>::epsilon();
  string approx = (argc > 2) ?      argv[2]  : "P1";
  Float  p      = (argc > 3) ? atof(argv[3]) : 1.5;
  Float  w      = (argc > 4) ? (is_float(argv[4]) ? atof(argv[4]) :2/p) :1;
  Float  tol    = (argc > 5) ? atof(argv[5]) : 1e5*eps;
  size_t max_it = (argc > 6) ? atoi(argv[6]) : 500;
  derr << "# P-Laplacian problem by fixed-point:" << endl
       << "# geo    = " << omega.name() << endl
       << "# approx = " << approx << endl
       << "# p      = " << p << endl
       << "# w      = " << w << endl
       << "# tol    = " << tol << endl;
  space Xh (omega, approx);
  Xh.block ("boundary");
  trial u (Xh); test v (Xh);
  form m = integrate (u*v);
  problem pm (m);
  field uh (Xh), uh_star (Xh, 0.);
  uh["boundary"] = uh_star["boundary"] = 0;
  field lh = integrate (v);
  dirichlet (lh, uh);
  derr << "# n r v" << endl;
  Float r = 1, r0 = 1;
  size_t n = 0;
  do {
    form a = integrate(compose(eta(p),norm2(grad(uh)))*dot(grad(u),grad(v)));
    field mrh = a*uh - lh;
    field rh (Xh, 0);
    pm.solve (mrh, rh);
    r = rh.max_abs();
    if (n == 0) { r0 = r; }
    Float v = (n == 0) ? 0 : log10(r0/r)/n;
    derr << n << " " << r << " " << v << endl;
    if (r <= tol || n++ >= max_it) break;
    problem p (a);
    p.solve (lh, uh_star);
    uh = w*uh_star + (1-w)*uh;
  } while (true);
  dout << catchmark("p") << p << endl
       << catchmark("u") << uh;
  return (r <= tol) ? 0 : 1;
}
```

Comments

The implementation with **Rheolef** involves a weighted forms: the tensor-valued weight $\eta\left(\left|\nabla u_h^{(n)}\right|^2\right)$ is inserted in the variational expression passed to the `integrate` function. The construction of the weighted form $a(.;.,.)$ writes:

```
form a = integrate(compose(eta(p),norm2(grad(uh)))*dot(grad(u),grad(v)));
```

Remarks the usage of the `compose`, `norm2` and `grad` library functions. The weight $\eta\left(\left|\nabla u_h^{(n)}\right|^2\right)$ is represented by the `compose(eta(p),norm2(grad(uh)))` sub-expression. This weight is evaluated on the fly at the quadrature nodes during the assembly process implemented by the `integrate` function. Also, notice the distinction between u_h, that represents the value of the solution at step

n, and the trial u and test v functions, that represents any elements of the function space X_h. These functions appear in the `dot(grad(u),grad(v))` sub-expression.

File 3.11: `eta.h`

```
struct eta {
  Float operator() (const Float& z) const {
   check_macro(z != 0 || p > 2, "eta: division by zero (HINT: check mesh)");
   return pow(z, (p-2)/2);
  }
  Float derivative (const Float& z) const {
   check_macro(z != 0 || p > 4, "eta': division by zero (HINT: check mesh)");
   return 0.5*(p-2)*pow(z, (p-4)/2);
  }
  eta (const Float& q) : p(q) {}
  Float p;
};
```

The η function is implemented separately, in file named `eta.h` in order to easily change its definition. The `derivative` member function is not yet used here: it is implemented for a forthcoming application (the Newton method). Note the guards that check for division by zero and send a message related to the mesh: this will be commentated in the next paragraph. Finally, the fixed-point algorithm is initiated with $u^{(0)}$ as the solution of the linear problem associated to $p = 2$, i.e. the standard Poisson problem with Dirichlet boundary conditions.

File 3.12: `dirichlet.icc`

```
void dirichlet (const field& lh, field& uh) {
  const space& Xh = lh.get_space();
  trial u (Xh); test v (Xh);
  form a = integrate (dot(grad(u),grad(v)));
  problem p (a);
  p.solve (lh, uh);
}
```

Running the program

Compile the program, as usual:

```
make p_laplacian_fixed_point
```

and enter the commands:

```
mkgeo_ugrid -t 50 > square.geo
geo square.geo
```

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

```
./p_laplacian_fixed_point square.geo P1 1.5 > square.field
```

Run the field visualization:

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

The first command shows an elevation view of the solution (see Fig. 3.7) while the second one shows a cut along the first bisector $x_0 = x_1$. Observe that the solution becomes flat at the center when p decreases. The $p = 2$ case, corresponding to the linear case, is showed for the purpose of comparison.

There is a technical issue concerning the mesh: the computation could failed on some mesh that presents at least one triangle with two edges on the boundary:

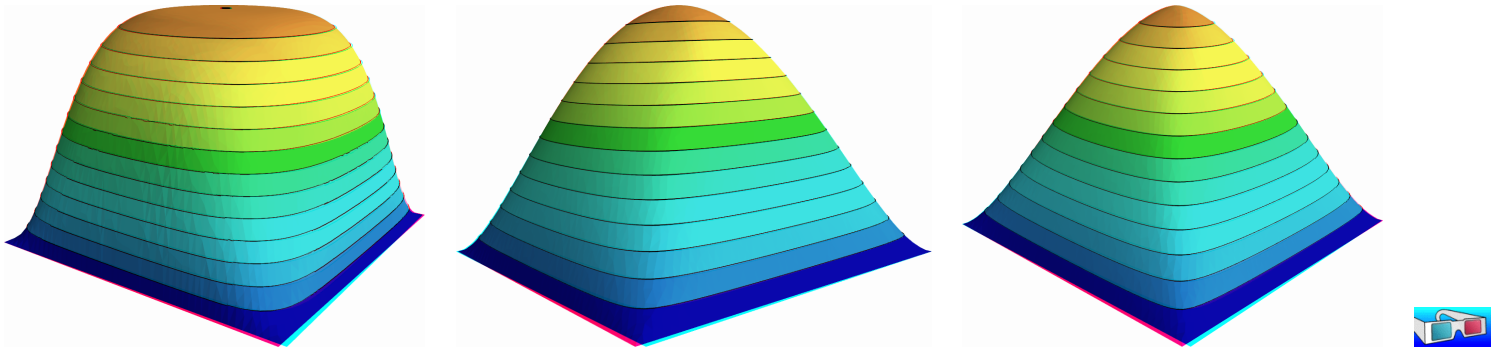


Figure 3.7: The p-Laplacian for $d = 2$: elevation view for $p = 1.25$ (left), $p = 2$ (center) and $p = 2.5$ (right).

```
mkgeo_grid -t 50 > square-bedge.geo
geo square-bedge.geo
./p_laplacian_fixed_point square-bedge.geo P1 1.5 > square-bedge.field
```

The computation stops and claims a division by zero: the three nodes of such a triangle, the three nodes are on the boundary, where $u_h = 0$ is prescribed: thus $\boldsymbol{\nabla} u_h = 0$ uniformly inside this element. Note that this failure occurs only for linear approximations: the computation works well on such meshes for P_k approximations with $k \geqslant 2$. While the `mkgeo_grid` generates uniform meshes that have such triangles, the `mkgeo_ugrid` calls the `gmsh` generator that automatically splits the triangles with two boundary edges. When using `bamg`, you should consider the `-splitpbedge` option.

Convergence properties of the fixed-point algorithm

The fixed-point algorithm prints also r_n, the norm of the residual term, at each iteration n, and the convergence rate $v_n = \log_{10}(r_n/r_0)/n$. The residual term of the non-linear variational formulation is defined by:

$$r_h^{(n)} \in V_h \quad \text{and} \quad m\left(r_h^{(n)}, v_h\right) = a\left(u_h^{(n)}; u_h^{(n)}, v_h\right) - l(v_h), \quad \forall v_h \in V_h$$

where $m(.,.)$ denotes the L^2 scalar product. Clearly, $u_h^{(n)}$ is a solution if and only if $r_h^{(n)} = 0$.
For clarity, let us drop temporarily the n index of the current iteration. The field $r_h \in V_h$ can be extended as a field $r_h \in X_h$ with vanishing components on the boundary. The previous relation writes, after expansion of the bilinear forms and fields on the unknown and blocked parts (see page 16 for the notations):

```
m.uu*rh.u = a.uu*uh.u + a.ub*ub.b - lh.u
     rh.b = 0
```

This relation expresses that the residual term r_h is obtained by solving a linear system involving the mass matrix.
It remains to choose a good norm for estimating this residual term. For the corresponding continuous formulation, we have:

$$r = -\text{div}\left(\eta\left(|\boldsymbol{\nabla} u|^2\right)\boldsymbol{\nabla} u\right) - f \ \in W^{-1,p}(\Omega)$$

Thus, for the continuous formulation, the residual term may be measured with the $W^{-1,p}(\Omega)$

norm. It is defined, for all $\varphi \in W^{-1,p}(\Omega)$, by duality:

$$\|\varphi\|_{-1,p,\Omega} = \sup_{\substack{\varphi \in W_0^{1,p}(\Omega) \\ v \neq 0}} \frac{\langle \varphi, v \rangle}{\|v\|_{1,p,\Omega}} = \sup_{\substack{v \in W_0^{1,p}(\Omega) \\ \|v\|_{1,p,\Omega}=1}} \langle \varphi, v \rangle$$

where $\langle .,. \rangle$ denotes the duality bracked between $W_0^{1,p}(\Omega)$ and $W^{-1,p}(\Omega)$.
By analogy, let us introduce the discrete $W^{-1,p}(\Omega)$ norm, denoted as $\|.\|_{-1,h}$, defined by duality for all $\varphi_h \in V_h$ by:

$$\|\varphi_h\|_{-1,h} = \sup_{\substack{v_h \in V_h \\ \|v_h\|_{1,p,\Omega}=1}} \langle \varphi_h, v_h \rangle$$

The dual of space of the finite element space V_h is identified to V_h and the duality bracket is the Euclidean scalar product of $\mathbb{R}^{\dim(V_h)}$. Then, $\|\varphi_h\|_{-1,h}$ is the largest absolute value of components of φ_h considered as a vector of $\mathbb{R}^{\dim(V_h)}$. With the notations of the **Rheolef** library, it simply writes:

```
Float r = rh.u().max_abs()
```

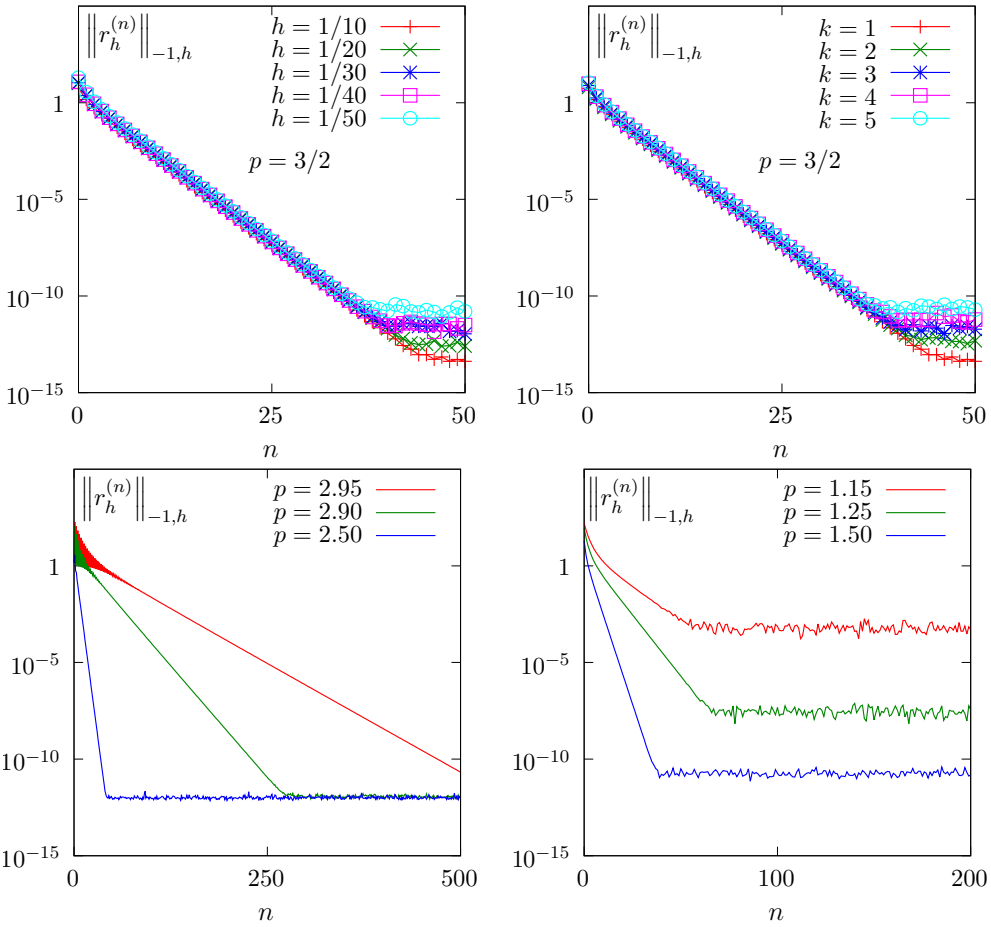


Figure 3.8: The fixed-point algorithm on the p-Laplacian for $d = 2$: when $p = 3/2$, independence of the convergence properties of the residue (top-left) with mesh refinement; (top-right) with polynomial order P_k; when $h = 1/50$ and $k = 1$, convergence (bottom-left) for $p > 2$ and (bottom-right) for $p < 2$.

Fig 3.8.top-left shows that the residual term decreases exponentially versus n, since the slope of the plot in semi-log scale tends to be strait. Moreover, observe that the slope is independent of the mesh size h. Also, by virtue of the previous careful definition of the residual term and its corresponding norm, all the slopes falls into a master curve.

These invariance properties applies also to the polynomial approximation P_k : Fig 3.8.top-right shows that all the curves tends to collapse when k increases. Thus, the convergence properties of the algorithm are now investigated on a fixed mesh $h = 1/50$ and for a fixed polynomial approximation $k = 1$.

Fig 3.8.bottom-left and 3.8.bottom-right show the convergence versus the power-law index p: observe that the convergence becomes easier when p approaches $p = 2$, where the problem is linear. In that case, the convergence occurs in one iteration. Nevertheless, it appears two limitations. From one hand, when $p \to 3$ the convergence starts to slow down and $p \geqslant 3$ cannot be solved by this algorithm (it will be solved later in this chapter). From other hand, when $p \to 1$, the convergence slows down too and numerical rounding effets limits the convergence: the machine precision canot be reached. Let us introduce the convergence rate $v_n = \log_{10}(r_n/r_0)/n$ it tends to

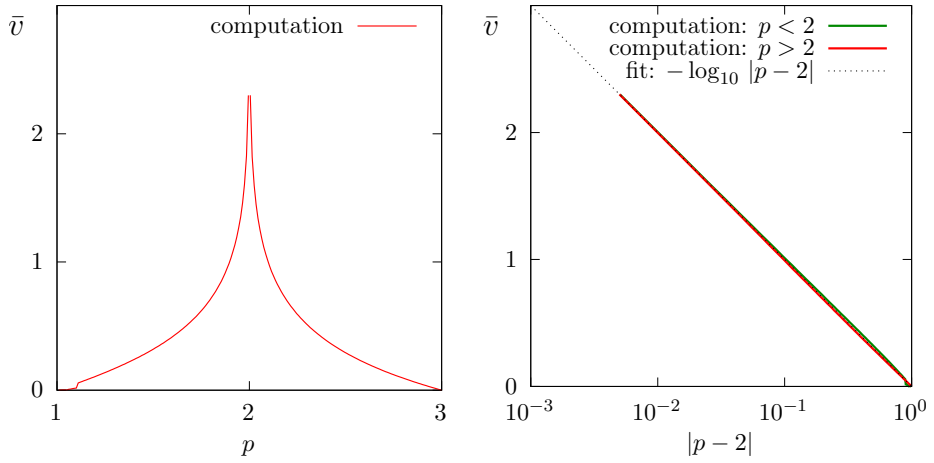


Figure 3.9: The fixed-point algorithm on the p-Laplacian for $d = 2$: (left) convergence rate versus p; (right) convergence rate versus p in semi-log scale.

a constant, denoted as $\bar{v}$ and: $r_n \approx r_0 \times 10^{-\bar{v}\,n}$. Observe on Fig 3.9.left that $\bar{v}$ tends to $+\infty$ when $p = 2$, since the system becomes linear and the algorithm converge in one iteration. Observe also that $\bar{v}$ tends to zero for $p = 1$ and $p = 3$ since the algorithm diverges. Fig 3.9.right shows the same plot in semi-log scale and shows that $\bar{v}$ behaves as: $\bar{v} \approx -\log_{10}|p-2|$. This study shows that the residual term of the fixed point algorithm behaves as:

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

Improvement by relaxation

The relaxation parameter can improve the fixed-point algorithm: for instance, for $p = 3$ and $\omega = 0.5$ we get a convergent sequence:

```
./p_laplacian_fixed_point square.geo P1 3 0.5 > square.field
```

Observe on Fig. 3.10 the effect on the relaxation parameter ω upon the convergence rate $\bar{v}$: for $p < 2$ it can improve it and for $p > 2$, it can converge when $p > 3$. For each p, there is clearly an optimal relaxation parameter, denoted by ω_{opt}. A simple fit shows that (see Fig. 3.10.bottom-left):

$$\omega_{\text{opt}} = 2/p$$

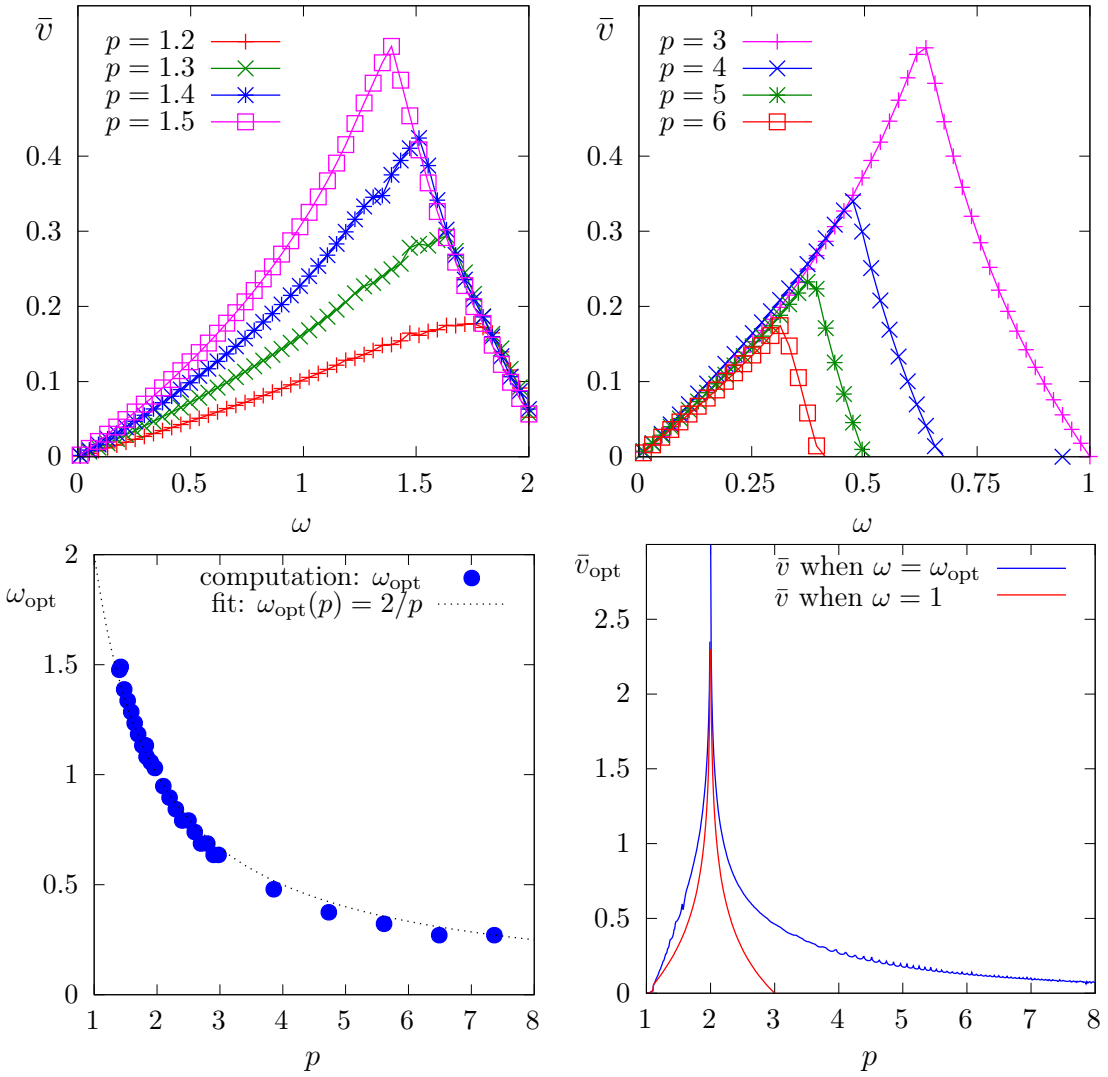


Figure 3.10: The fixed-point algorithm on the p-Laplacian for $d = 2$: effect of the relaxation parameter ω (top-left) when $p < 2$; (top-right) when $p > 2$; (bottom-left) optimal $\omega_{\rm opt}$; (bottom-right) optimal $\bar{v}_{\rm opt}$.

Let us denote $\bar{v}_{\rm opt}$ the corresponding rate of convergence. Fig. 3.10.top-right shows that the convergence is dramatically improved when $p > 2$ while the gain is less pronounced when $p < 2$. Coveniently replacing the extra parameter ω on the command line by - leads to compute automatically $\omega = \omega_{\rm opt}$: the fixed-point algorithm is always convergent with an optimal convergent rate, e.g.:

```
./p_laplacian_fixed_point square.geo P1 4.0 - > square.field
```

There is no way to improve more the fixed point algorithm: the next paragraph shows a different algorithm that dramatically accelerates the computation of the solution.

3.2.3 The Newton algorithm

Principe of the algorithm

An efficient alternative to the fixed-point algorithm is to solve the nonlinear problem (P) by using the Newton algorithm. Let us consider the following operator:

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

The F operator computes simply the residual term and the problem expresses now as: find $u \in W_0^{1,p}(\Omega)$ such that $F(u) = 0$.
The Newton algorithm reduces the nonlinear problem into a sequence of linear subproblems: the sequence $\left(u^{(n)}\right)_{n\geqslant 0}$ is classically defined by recurrence as:

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

$$F'\left(u^{(n)}\right)\ \delta u^{(n)} = -F\left(u^{(n)}\right)$$

  and then compute explicitly:

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

The notation $F'(u)$ stands for the Fréchet derivative of F, as an operator from $W^{-1,p}(\Omega)$ into $W_0^{1,p}(\Omega)$. For any $r \in W^{-1,p}(\Omega)$, the linear tangent problem writes:
find $\delta u \in W_0^{1,p}(\Omega)$ such that:

$$F'(u)\,\delta u = -r$$

After the computation of the Fréchet derivative, we obtain the strong form of this problem:
(LT): find δu, defined in Ω, such that

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

where

$$\eta'(z) = \frac{1}{2}(p-2)z^{\frac{p-4}{2}},\ \forall z > 0$$

This is a Poisson-like problem with homogeneous Dirichlet boundary conditions and a non-constant tensorial coefficient. The variational form of the linear tangent problem writes:
(VLT): find $\delta u \in W_0^{1,p}(\Omega)$ such that

$$a_1(u;\delta u,\delta v) = l_1(v),\quad \forall \delta v \in W_0^{1,p}(\Omega)$$

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

$$\begin{array}{rcl} a_1(u;\delta u,\delta v) & = & \displaystyle\int_\Omega \left(\eta\left(|\boldsymbol{\nabla} u|^2\right)\boldsymbol{\nabla}(\delta u).\boldsymbol{\nabla}(\delta v) + 2\eta'\left(|\boldsymbol{\nabla} u|^2\right)\left\{\boldsymbol{\nabla} u.\boldsymbol{\nabla}(\delta u)\right\}\left\{\boldsymbol{\nabla} u.\boldsymbol{\nabla}(\delta v)\right\}\right)\mathrm{d}x \\ l_1(v) & = & \displaystyle -\int_\Omega r\,v\,\mathrm{d}x \end{array}$$

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

$$\nu(\boldsymbol{\xi}) = \eta\left(|\boldsymbol{\xi}|^2\right)\ I + 2\eta'\left(|\boldsymbol{\xi}|^2\right)\ \boldsymbol{\xi}\otimes\boldsymbol{\xi}$$

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

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

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

File 3.13: `p_laplacian_newton.cc`

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
#include "p_laplacian.h"
int main(int argc, char**argv) {
  environment rheolef (argc, argv);
  geo omega (argv[1]);
  Float  eps      = std::numeric_limits<Float>::epsilon();
  string approx   = (argc > 2) ?      argv[2]  : "P1";
  Float  p        = (argc > 3) ? atof(argv[3]) : 1.5;
  Float  tol      = (argc > 4) ? atof(argv[4]) : 1e5*eps;
  size_t max_iter = (argc > 5) ? atoi(argv[5]) : 500;
  derr << "# P-Laplacian problem by Newton:" << endl
       << "# geo = " << omega.name() << endl
       << "# approx = " << approx << endl
       << "# p = " << p << endl
       << "# tol = " << tol << endl
       << "# max_iter = " << max_iter << endl;
  p_laplacian F (p, omega, approx);
  field uh = F.initial ();
  int status = newton (F, uh, tol, max_iter, &derr);
  dout << setprecision(numeric_limits<Float>::digits10)
       << catchmark("p") << p << endl
       << catchmark("u") << uh;
  return status;
}
```

File 3.14: `p_laplacian.h`

```
class p_laplacian {
public:
  typedef field value_type;
  typedef Float float_type;
  p_laplacian (Float p, const geo& omega, string approx);
  field initial() const;
  field residue          (const field& uh) const;
  void update_derivative (const field& uh) const;
  field derivative_solve      (const field& mrh) const;
  field derivative_trans_mult (const field& mrh) const;
  Float space_norm       (const field& uh) const;
  Float dual_space_norm  (const field& mrh) const;
  Float p;
  space Xh;
  field lh;
  form  m;
  problem pm;
  mutable form a1;
  mutable problem pa1;
};
#include "p_laplacian1.icc"
#include "p_laplacian2.icc"
```

Comments

The Newton algorithm is implemented in a generic way, for any F function, by the `newton` function of the **Rheolef** library. The reference manual for the `newton` generic function is available online:

```
man newton
```

The function F and its derivative F' are provided by a template class argument. Here, the `p_laplacian` class describes our F function, i.e. our problem to solve: its interface is defined in the file '`p_laplacian.h`' and its implementation in '`p_laplacian1.icc`' and '`p_laplacian2.icc`'. The introduction of the class `p_laplacian` will allow an easy exploration of some variants of the Newton algorithm for this problem, as we will see in the next section.

File 3.15: `p_laplacian1.icc`

```
#include "eta.h"
#include "nu.h"
#include "dirichlet.icc"
p_laplacian::p_laplacian (Float p1, const geo& omega, string approx)
 : p(p1), Xh(), lh(), m(), pm(), a1(), pa1() {
  Xh = space (omega, approx);
  Xh.block ("boundary");
  trial u (Xh); test v (Xh);
  lh = integrate (v);
  m = integrate (u*v);
  pm = problem (m);
}
field p_laplacian::initial() const {
  field uh (Xh, 0);
  dirichlet (lh, uh);
  return uh;
}
field p_laplacian::residue (const field& uh) const {
  trial u (Xh); test v (Xh);
  form a = integrate (compose(eta(p), norm2(grad(uh)))*dot(grad(u),grad(v)));
  field mrh = a*uh - lh;
  mrh.set_b() = 0;
  return mrh;
}
void p_laplacian::update_derivative (const field& uh) const {
  size_t d = Xh.get_geo().dimension();
  trial u (Xh); test v (Xh);
  a1 = integrate (dot(compose(nu<eta>(eta(p),d), grad(uh))*grad(u),grad(v)));
  pa1 = problem (a1);
}
field p_laplacian::derivative_solve (const field& rh) const {
  field delta_uh (Xh,0);
  pa1.solve (rh, delta_uh);
  return delta_uh;
}
```

The residual term $F(u_h)$ is computed by the member function `residual` while the resolution of $F'(u_h)\delta u_h = Mr_h$ is performed by the function `derivative_solve`. The derivative $F'(u_h)$ is computed separately by the function `update_derivative`:

```
a1 = integrate(dot(compose(nu<eta>(eta(p),d),grad(uh))*grad(u),grad(v)));
```

Note that the $a_1(u;.,.)$ bilinear form is a tensorial weighted form, where $\nu = \nu(\boldsymbol{\nabla} u)$ is the weight tensor. The tensorial weight ν is inserted as $(\nu\nabla u).\nabla v$ in the variational expression for the `integrate` function. As the tensor ν is symmetric, the bilinear form $a_1(.,.)$ is also symmetric.

The linear system involving the derivative $F'(u_h)$ is solved by the `p_laplacian` member function `derivative_solve`. Finally, applying the generic Newton method requires a stopping criteria on the residual term: this is the aim of the member function `dual_space_norm`. The three last member functions are not used by the Newton algorithm, but by its extension, the damped Newton method, that will be presented later.

File 3.16: p_laplacian2.icc

```
field p_laplacian::derivative_trans_mult (const field& mrh) const {
  field rh (Xh, 0);
  pm.solve (mrh, rh);
  field mgh = a1*rh;
  mgh.set_b() = 0;
  return mgh;
}
Float p_laplacian::space_norm (const field& uh) const {
  return sqrt (m(uh,uh));
}
Float p_laplacian::dual_space_norm (const field& mrh) const {
  field rh (Xh, 0);
  pm.solve (mrh, rh);
  return sqrt (dual(mrh, rh));
}
```

The ν function is implemented for a generic η function, as a class-function that accept as template agument another class-function.

File 3.17: nu.h

```
template<class Function>
struct nu {
  tensor operator() (const point& grad_u) const {
    Float x2 = norm2 (grad_u);
    Float a = f(x2);
    Float b = 2*f.derivative(x2);
    tensor value;
    for (size_t i = 0; i < d; i++) {
      value(i,i)   = a + b*grad_u[i]*grad_u[i];
      for (size_t j = 0; j < i; j++)
        value(j,i) = value(i,j) = b*grad_u[i]*grad_u[j];
    }
    return value;
  }
  nu (const Function& f1, size_t d1) : f(f1), d(d1) {}
  Function f;
  size_t   d;
};
```

Running the program

Enter:

```
make p_laplacian_newton
mkgeo_ugrid -t 50 > square.geo
./p_laplacian_newton square.geo P1 3 > square.field
field square.field -elevation -stereo
```

The program prints at each iteration n, the residual term r_n in discrete $L^2(\Omega)$ norm. Convergence occurs in less than ten iterations: it dramatically improves the previous algorithm (see Fig. 3.11). Observe that the slope is no more constant in semi-log scale: the convergence rate accelerates and the slope tends to be vertical, the so-called super-linear convergence. This is the major advantage of the Newton method. Figs. 3.12.top-left and. 3.12.top-bottom shows that the algorithm converge when $p \geqslant 3$ and that the convergence properties are independent of the mesh size h and the polynomial order k. There are still two limitations of the method. From one hand, the Newton algorithm is no more independent of h and k when $p \leqslant 3/2$ and to tends to diverges in that case when h tends to zero (see Fig. 3.12.bottom-left). From other hand, when p becomes large (see Fig. 3.12.bottom-right), an overshoot in the convergence tends to increase and destroy the convergence, due to rounding problems. In order to circumvent these limitations, another strategy is considered in the next section: the damped Newton algorithm.

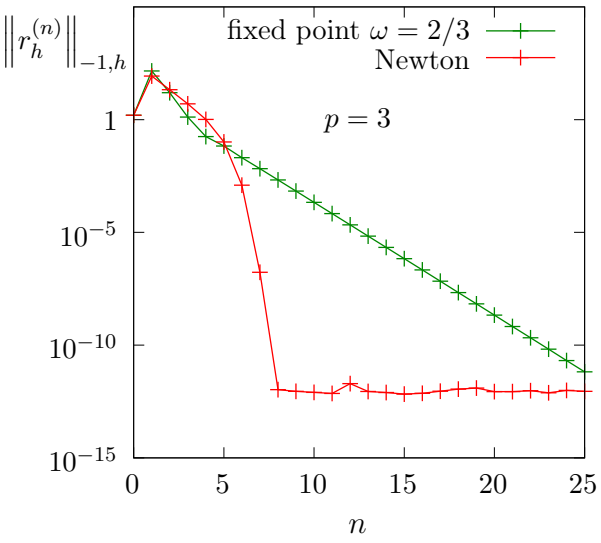


Figure 3.11: The Newton algorithm on the p-laplacian for $d = 2$: comparison with the fixed-point algorithm.

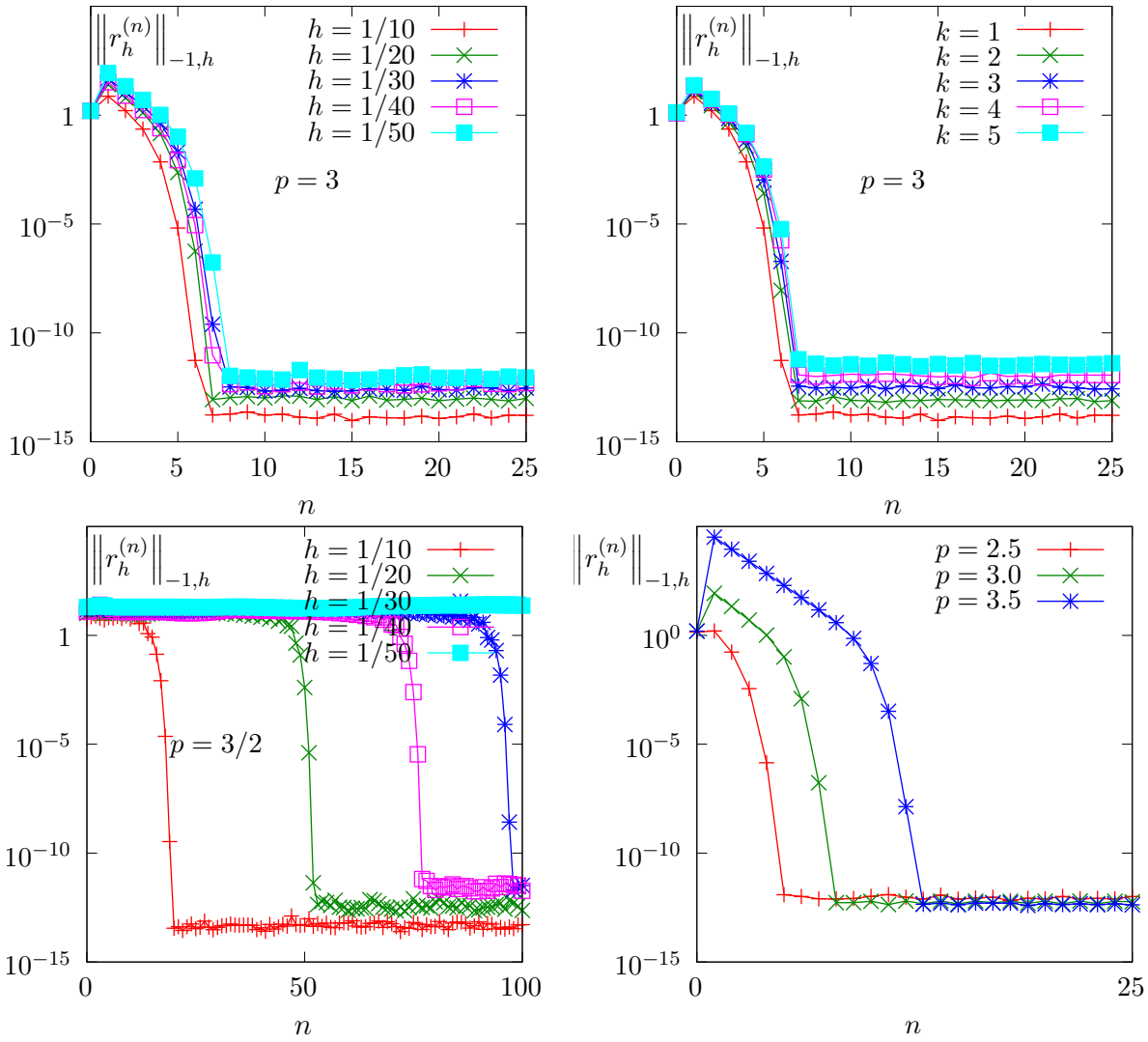


Figure 3.12: The Newton algorithm on the p-Laplacian for $d = 2$: (top-left) comparison with the fixed-point algorithm; when $p = 3$, independence of the convergence properties of the residue (top-left) with mesh refinement; (top-right) with polynomial order P_k; (bottom-left) mesh-dependence convergence when $p < 2$; (bottom-right) overshoot when $p > 2$.

3.2.4 The damped Newton algorithm

Principe of the algorithm

The Newton algorithm diverges when the initial $u^{(0)}$ is too far from a solution, e.g. when p is not at the vicinity of 2. Our aim is to modify the Newton algorithm and to obtain a *globally convergent algorithm*, i.e to converge to a solution for any initial $u^{(0)}$. By this way, the algorithm should converge for any value of $p \in]1, +\infty[$. The basic idea is to decrease the step length while maintaining the direction of the original Newton algorithm:

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

where $\lambda^{(n)} \in]0, 1]$ and $\delta u^{(n)}$ is the direction from the Newton algorithm, given by:

$$F'\left(u^{(n)}\right) \; \delta u^{(n)} = -F\left(u^{(n)}\right)$$

Let V a Banach space and let $T : V \to \mathbb{R}$ defined for any $v \in V$ by:

$$T(v) = \frac{1}{2} \| C^{-1} F(v) \|_V^2,$$

where C is some non-singular operator, easy to invert, used as a non-linear preconditioner. The simplest case, without preconditioner, is $C = I$. The T function furnishes a measure of the residual term in L^2 norm. The convergence is global when for any initial $u^{(0)}$, we have for any $n \geqslant 0$:

$$T\left(u^{(n+1)}\right) \leqslant T\left(u^{(n)}\right) + \alpha \left\langle T'\left(u^{(n)}\right), \; u^{(n+1)} - u^{(n)} \right\rangle_{V',V} \tag{3.5}$$

where $\langle ., . \rangle_{V',V}$ is the duality product between V and its dual V', and $\alpha \in]0, 1[$ is a small parameter. Note that

$$T'(u) \quad = \quad \{C^{-1} F'(u)\}^* C^{-1} F(u)$$

where the superscript * denotes the adjoint operator, i.e. the transpose matrix the in finite dimensional case. In practice we consider $\alpha = 10^{-4}$ and we also use a minimal step length $\lambda_{\min} = 1/10$ in order to avoid too small steps. Let us consider a fixed step $n \geqslant 0$: for convenience the n superscript is dropped in $u^{(n)}$ and $\delta u^{(n)}$. Let $g : \mathbb{R} \to \mathbb{R}$ defined for any $\lambda \in \mathbb{R}$ by:

$$g(\lambda) = T\left(u + \lambda \delta u\right)$$

Then :

$$\begin{aligned} g'(\lambda) &= \langle T'(u + \lambda \delta u), \, \delta u \rangle_{V',V} \\ &= \langle C^{-1} F(u + \lambda \delta u), \; F'(u + \lambda \delta u) C^{-1} \delta u \rangle_{V,V'} \end{aligned}$$

where the superscript * denotes the adjoint operator, i.e. the transpose matrix the in finite dimensional case. The practical algorithm for obtaining λ was introduced first in [J. E. Dennis and Schnablel, 1983] and is also presented in [Press et al., 1997, p. 385]. The step length λ that satisfy (3.5) is computed by using a finite sequence λ_k, $k = 0, 1 \ldots$ with a second order recurrence:

- $k = 0$: initialization $\lambda_0 = 1$. If (3.5) is satisfied with $u + \lambda_0 \, d$ then let $\lambda := \lambda_0$ and the sequence stop here.
- $k = 1$: first order recursion. The quantities $g(0) = f(u)$ et $g'(0) = \langle f'(u), \, d \rangle$ are already computed at initialization. Also, we already have computed $g(1) = f(u + d)$ when verifying whether (3.5) was satisfied. Thus, we consider the following approximation of $g(\lambda)$ by a second order polynomial:

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

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

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

Since the initialization at $k = 0$ does not satisfy (3.5), it is possible to show that, when α is small enough, we have $\tilde{\lambda}_1 \leqslant 1/2$ and $\tilde{\lambda}_1 \approx 1/2$. Let $\lambda_1 := \max(\lambda_{\min}, \tilde{\lambda}_1)$. If (3.5) is satisfied with $u + \lambda_1 d$ then let $\lambda := \lambda_1$ and the sequence stop here.

- $k \geqslant 2$: second order recurrence. The quantities $g(0) = f(u)$ et $g'(0) = \langle f'(u), d \rangle$ are available, together with λ_{k-1}, $g(\lambda_{k-1})$, λ_{k-2} and $g(\lambda_{k-2})$. Then, $g(\lambda)$ is approximated by the following third order polynomial:

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

where a et b are expressed by:

$$\begin{pmatrix} a \\ b \end{pmatrix} = \frac{1}{\lambda_{k-1} - \lambda_{k-2}} \begin{pmatrix} \frac{1}{\lambda_{k-1}^2} & -\frac{1}{\lambda_{k-2}^2} \\ -\frac{\lambda_{k-2}}{\lambda_{k-1}^2} & \frac{\lambda_{k-1}}{\lambda_{k-2}^2} \end{pmatrix} \begin{pmatrix} g(\lambda_{k-1}) - g'(0)\lambda_{k-1} - g(0) \\ g(\lambda_{k-2}) - g'(0)\lambda_{k-2} - g(0) \end{pmatrix}$$

The minimum of $\tilde{g}_k(\lambda)$ is

$$\tilde{\lambda}_k = \frac{-b + \sqrt{b^2 - 3ag'(0)}}{3a}$$

Let $\lambda_k = \min(1/2\,\lambda_k, \max(\tilde{\lambda}_k/10, \tilde{\lambda}_{k+1})$ in order for λ_k to be at the same order of magnitude as λ_{k-1}. If (3.5) is satisfied with $u + \lambda_k d$ then let $\lambda := \lambda_k$ and the sequence stop here.

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

File 3.18: `p_laplacian_damped_newton.cc`

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
#include "p_laplacian.h"
int main(int argc, char**argv) {
  environment rheolef (argc,argv);
  geo omega (argv[1]);
  Float eps = numeric_limits<Float>::epsilon();
  string approx   = (argc > 2) ?      argv[2]  : "P1";
  Float  p        = (argc > 3) ? atof(argv[3]) : 1.5;
  Float  tol      = (argc > 4) ? atof(argv[4]) : eps;
  size_t max_iter = (argc > 5) ? atoi(argv[5]) : 500;
  derr << "# P-Laplacian problem by damped Newton:" << endl
       << "# geo = " << omega.name() << endl
       << "# approx = " << approx << endl
       << "# p = " << p << endl;
  p_laplacian F (p, omega, approx);
  field uh = F.initial ();
  int status = damped_newton (F, uh, tol, max_iter, &derr);
  dout << catchmark("p") << p << endl
       << catchmark("u") << uh;
  return status;
}
```

Comments

The `damped_newton` function implements the damped Newton algorithm for a generic $T(u)$ function, i.e. a generic nonlinear preconditioner. This algorithms use a backtrack strategy implemented

in the file 'newton-backtrack.h' of the **Rheolef** library. The simplest choice of the identity preconditioner $C = I$ i.e. $T(u) = \|F(u)\|_{V'}^2/2$ 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:

$$\begin{aligned} g'(0) &= \langle T'(u),\, \delta u \rangle_{V',V} \\ &= \langle F(u),\; F'(u)\delta u \rangle_{V',V'} \\ &= -\|F(u)\|_{V'}^2 \end{aligned}$$

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.

Running the program

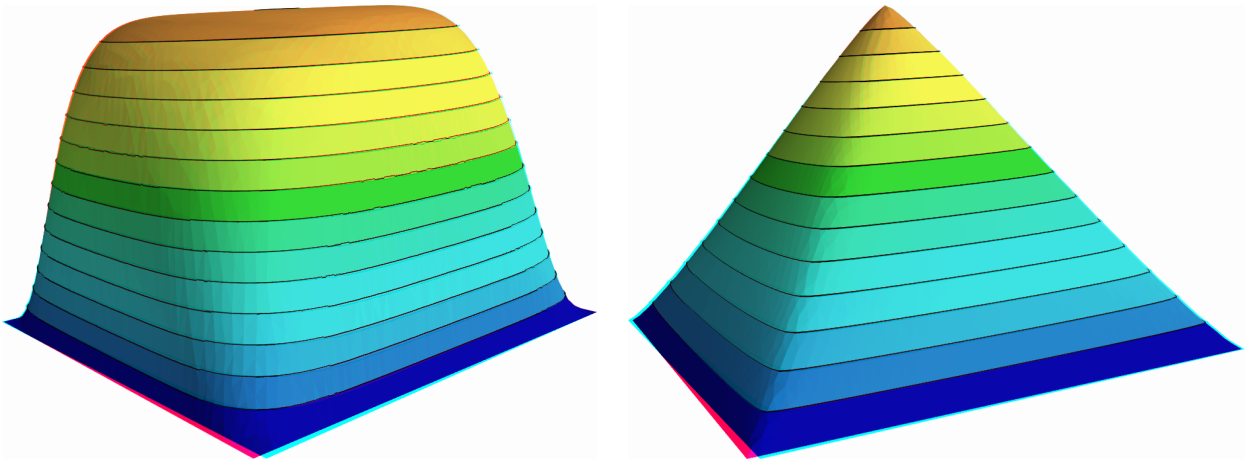


Figure 3.13: The p-Laplacian for $d = 2$: elevation view for $p = 1.15$ (left) and $p = 7$ (right).

As usual, enter:

```
make p_laplacian_damped_newton
mkgeo_ugrid -t 50 > square.geo
./p_laplacian_damped_newton square.geo P1 1.15 | field -stereo -elevation -
./p_laplacian_damped_newton square.geo P1 7    | field -stereo -elevation -
```

See Fig. 3.13 for the elevation view of the solution. The algorithm is now quite robust: the convergence occurs for quite large range of $p > 1$ values and extends the range previously presented on Fig. 3.7. The only limitation is now due to machine roundoff on some architectures.

Figs. 3.14.top shows that the convergence properties seems to slightly depend on the mesh refinement. Nevertheless, there are quite good and support both mesh refinement and high order polynomial degree. When p is far from $p = 2$, i.e. either close to one or large, Figs. 3.14.bottom shows that the convergence becomes slower and that the first linear regime, corresponding to the line search, becomes longer. This first regime finishes by a brutal super-linear regime, where the residual terms fall in few iterations to the machine precision.

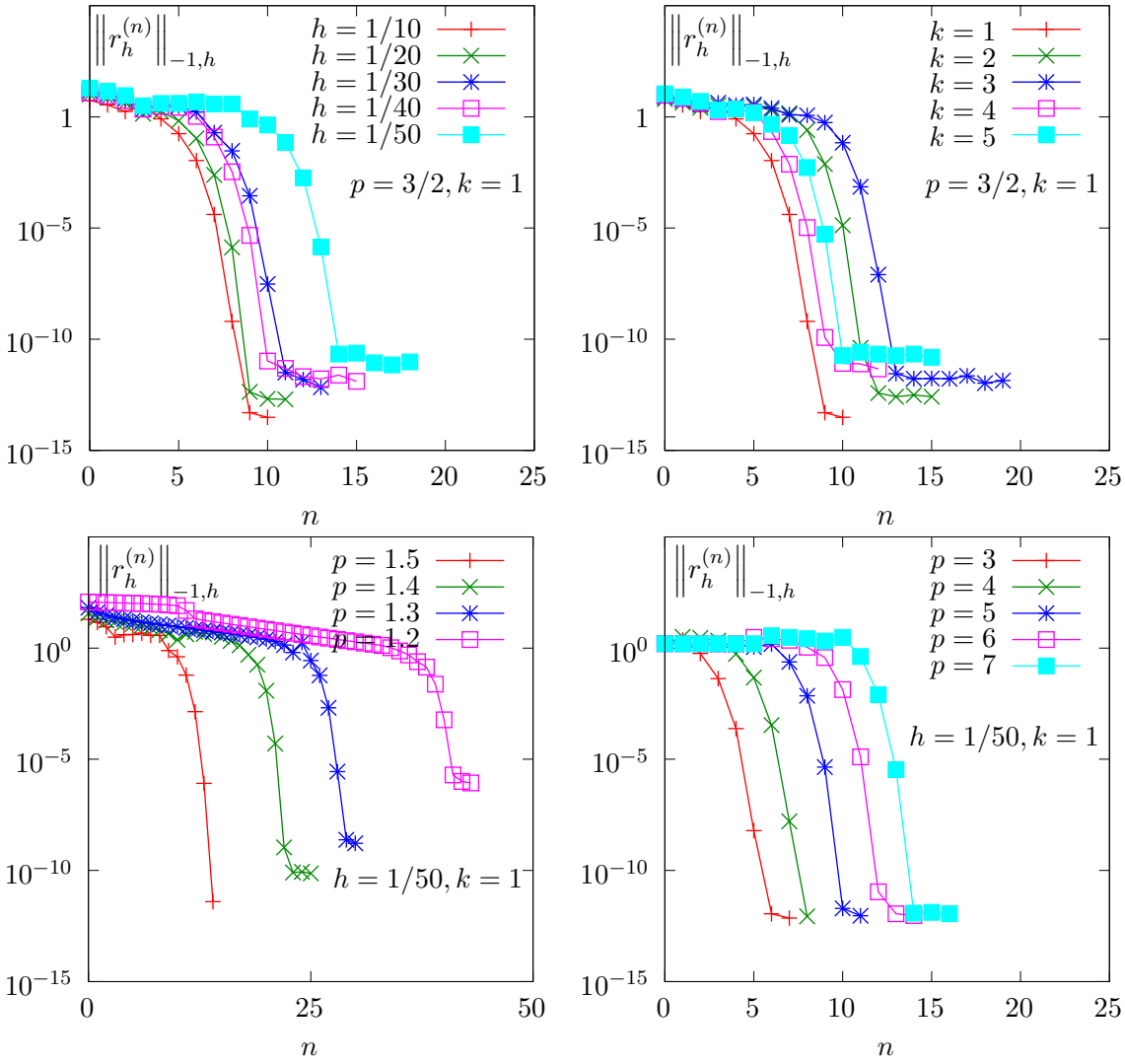


Figure 3.14: The damped Newton algorithm on the p-Laplacian for $d = 2$: when $p = 1.5$ and $h = 1/50$, convergence properties of the residue (top-left) with mesh refinement; (top-right) with polynomial order P_k; (bottom-left) convergence when $p < 2$; (bottom-right) when $p > 2$.

3.2.5 Error analysis

While there is no simple explicit expression for the exact solution in the square $\Omega =]0,1[^2$, there is one when considering Ω as the unit circle:

$$u(x) = \frac{(p-1)\ 2^{-\frac{1}{p-1}}}{p} \left(1 - \left(x_0^2 + x_1^2\right)^{\frac{p}{p-1)}}\right)$$

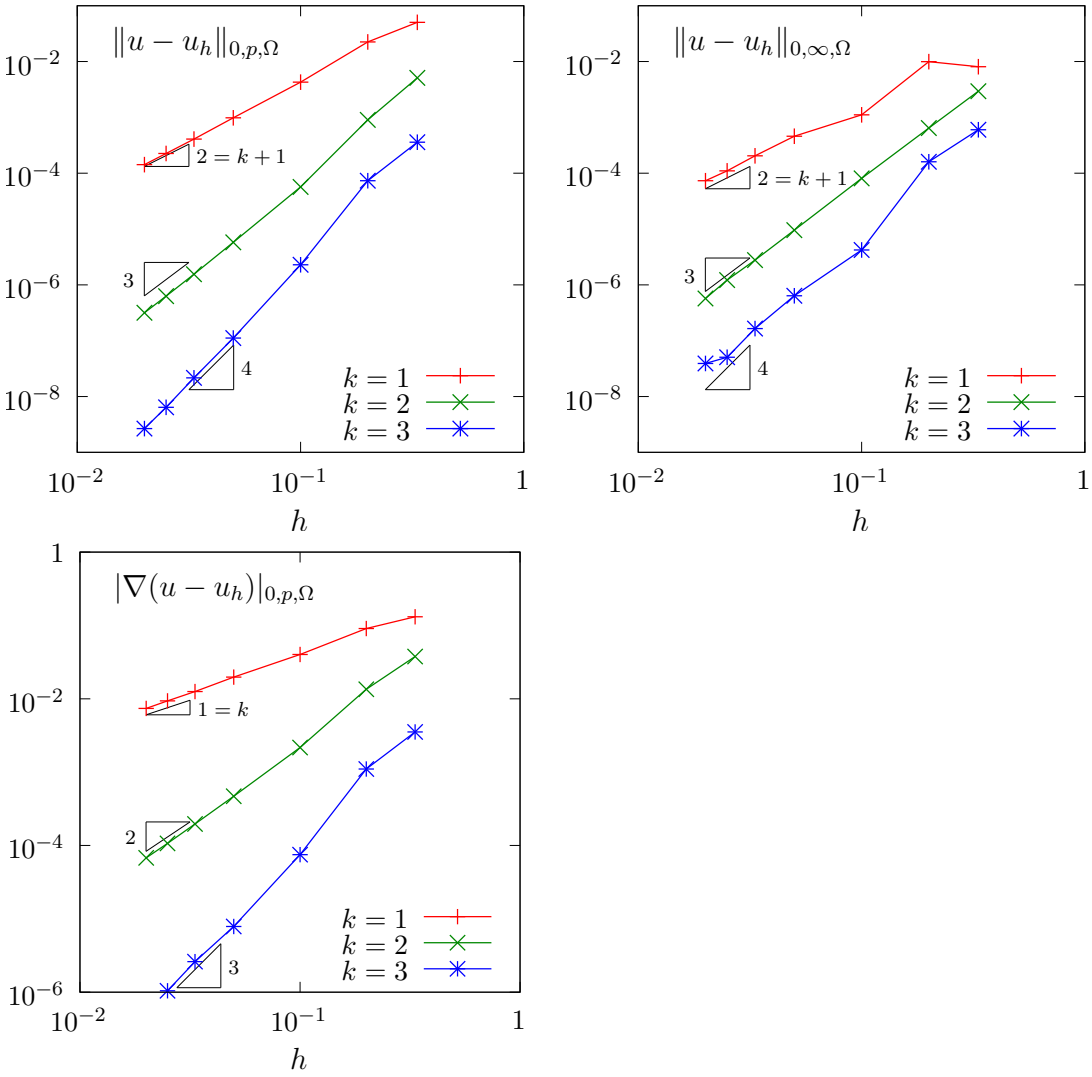


Figure 3.15: The p-Laplacian for $d = 2$: error analysis.

File 3.19: `p_laplacian_circle.h`

```
struct u_exact {
  Float operator() (const point& x) const {
    return (1 - pow(norm2(x), p/(2*p-2)))/((p/(p-1))*pow(2.,1/(p-1)));
  }
  u_exact (Float q) : p(q) {}
  protected: Float p;
};
struct grad_u {
  point operator() (const point& x) const {
    return - (pow(norm2(x), p/(2*p-2) - 1)/pow(2.,1/(p-1)))*x;
  }
  grad_u (Float q) : p(q) {}
  protected: Float p;
};
```

File 3.20: p_laplacian_error.cc

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
#include "p_laplacian_circle.h"
int main(int argc, char**argv) {
  environment rheolef (argc,argv);
  Float tol = (argc > 1) ? atof(argv[1]) : 1e-15;
  Float p;
  field uh;
  din >> catchmark("p") >> p
      >> catchmark("u") >> uh;
  const geo& omega = uh.get_geo();
  const space& Xh = uh.get_space();
  field pi_h_u = lazy_interpolate (Xh, u_exact(p));
  field eh = pi_h_u - uh;
  integrate_option iopt;
  iopt.set_family(integrate_option::gauss);
  iopt.set_order(2*Xh.degree());
  Float err_lp  = pow(integrate (omega,
    pow(fabs(uh - u_exact(p)), p), iopt), 1./p);
  Float err_w1p = pow(integrate (omega,
    pow(norm(grad(uh) - grad_u(p)), p), iopt), 1./p);
  Float err_linf = eh.max_abs();
  dout << "err_linf = " << err_linf << endl
       << "err_lp   = " << err_lp << endl
       << "err_w1p  = " << err_w1p << endl;
  return (err_linf < tol) ? 0 : 1;
}
```

Note, in the file 'p_laplacian_error.cc', the usage of the `integrate` function, together with a quadrature formula specification, for computing the errors in L^p norm and $W^{1,p}$ semi-norm. Note also the flexibility of expressions, mixing together `fields` as `uh` and functors, as `u_exact`. The whole expression is evaluated by the `integrate` function at quadrature points inside each element of the mesh.

By this way, the error analysis investigation becomes easy:

```
make p_laplacian_error
mkgeo_ball -t 10 -order 2 > circle-10-P2.geo
./p_laplacian_damped_newton circle-10-P2.geo P2 1.5 | ./p_laplacian_error
```

We can vary both the mesh size and the polynomial order and the error plots are showed on Fig. 3.15 for both the L^2, L^∞ norms and the $W^{1,p}$ semi-norm. Observe the optimal error behavior: the slopes in the log-log scale are the same as those obtained by a direct Lagrange interpolation of the exact solution.

3.3 Continuation and bifurcation methods

This chapter is an introduction to continuation and bifurcation methods with **Rheolef**. We consider a model nonlinear problem that depends upon a parameter. This problem is inspired from application to combustion. Solutions exists for a limited range of this parameter and there is a limit point: beyond this limit point there is no more solution. Our first aim is to compute the branch of solutions until this limit point, thanks to the continuation algorithm. Moreover, the limit point is a turning point for the branch of solutions: there exists a second branch of solutions, continuing the first one. Our second aim is to compute the second branch of solutions after this limit point with the Keller continuation algorithm. For simplicity in this presentation, the discretization is in one dimension: the extension to high order space dimension is immediate.

3.3.1 Problem statement and the Newton method

Let us consider the following model problem (see Paumier, 1997, p. 59 or Crouzeix and Rappaz, 1990, p. 2), defined for all $\lambda \in \mathbb{R}$:

(P): *find u, defined in Ω such that*

$$\begin{aligned} -\Delta u + \lambda \exp(u) &= 0 \text{ in } \Omega \\ u &= 0 \text{ on } \partial\Omega \end{aligned}$$

In order to apply a Newton method to the whole problem, let us introduce:

$$F(\lambda, u) = -\Delta u + \lambda \exp(u)$$

Then, the Gâteau derivative at any $(\lambda, u) \in \mathbb{R} \times H_0^1(\Omega)$ is given by:

$$\frac{\partial F}{\partial u}(\lambda, u).(v) = -\Delta v + \lambda \exp(u)v, \ \forall v \in H_0^1(\Omega)$$

File 3.21: `combustion.h`

```
struct combustion {
  typedef Float float_type;
  typedef field value_type;
  combustion(const geo& omega=geo(), string approx="");
  void reset(const geo& omega, string approx);
  field initial (std::string restart="");
  idiststream& get (idiststream& is, field& uh);
  odiststream& put (odiststream& os, const field& uh) const;
  string parameter_name() const { return "lambda"; }
  float_type parameter() const { return lambda; }
  void set_parameter(float_type lambda1) { lambda = lambda1; }
  bool stop (const field& xh) const { return xh.max_abs() > 10; }
  field residue (const field& uh) const;
  form derivative (const field& uh) const;
  field derivative_versus_parameter (const field& uh) const;
  problem::determinant_type update_derivative (const field& uh) const;
  field derivative_solve (const field& mrh) const;
  field derivative_trans_mult (const field& mrh) const;
  field massify   (const field& uh) const { return m*uh; }
  field unmassify (const field& uh) const;
  float_type space_dot  (const field& xh, const field& yh) const;
  float_type dual_space_dot  (const field& mrh, const field& msh) const;
protected:
  float_type lambda;
  space Xh;
  form m;
  problem pm;
  mutable form a1;
  mutable problem pa1;
  mutable branch event;
};
#include "combustion1.icc"
#include "combustion2.icc"
```

File 3.22: combustion1.icc

```
combustion::combustion (const geo& omega, string approx)
 : lambda(0), Xh(), m(), pm(), a1(), pa1(), event("lambda","u") {
  if (approx != "") reset (omega, approx);
}
void combustion::reset (const geo& omega, string approx) {
  Xh = space (omega, approx);
  Xh.block ("boundary");
  m = form (Xh, Xh, "mass");
  pm = problem (m);
}
field combustion::initial (std::string restart) {
  if (restart == "") return field (Xh, 0);
  idiststream in (restart);
  field xh0;
  get (in, xh0);
  derr << "# restart from lambda=" << lambda << endl;
  return xh0;
}
odiststream& combustion::put (odiststream& os, const field& uh) const {
  return os << event(lambda,uh);
}
idiststream& combustion::get (idiststream& is, field& uh) {
  is >> event(lambda,uh);
  if (!is) return is;
  if (Xh.name() == "") reset (uh.get_geo(), uh.get_approx());
  if (uh.b().dis_size() == 0) {
    // re-allocate the field with right blocked/unblocked sizes
    field tmp = field(Xh, 0);
    std::copy (uh.begin_dof(), uh.end_dof(), tmp.begin_dof());
    uh = tmp;
  }
  return is;
}
```

File 3.23: combustion2.icc

```
field combustion::residue (const field& uh) const {
  test v(Xh);
  field mrh = integrate(dot(grad(uh),grad(v)) - lambda*exp(uh)*v);
  mrh.set_b() = 0;
  return mrh;
}
form combustion::derivative (const field& uh) const {
  trial du(Xh); test v(Xh);
  return integrate(dot(grad(du),grad(v)) - lambda*exp(uh)*du*v);
}
problem::determinant_type
combustion::update_derivative (const field& uh) const {
  a1 = derivative (uh);
  solver_option sopt;
  sopt.compute_determinant = true;
  pa1 = problem (a1, sopt);
  return pa1.det();
}
field combustion::derivative_versus_parameter (const field& uh) const {
  test v(Xh);
  return - integrate(exp(uh)*v);
}
field combustion::derivative_solve (const field& rh) const {
  field delta_uh (Xh,0);
  pa1.solve (rh, delta_uh);
  return delta_uh;
}
field combustion::derivative_trans_mult (const field& mrh) const {
  field rh = unmassify(mrh);
  field mgh = a1*rh;
  mgh["boundary"] = 0;
  return mgh;
}
field combustion::unmassify (const field& mrh) const {
  field rh (Xh, 0);
  pm.solve (mrh, rh);
  return rh;
}
Float combustion::space_dot (const field& xh, const field& yh) const {
  return m(xh,yh); }
Float combustion::dual_space_dot (const field& mrh, const field& msh) const{
  return dual(unmassify(mrh), msh); }
```

File 3.24: combustion_newton.cc

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
#include "combustion.h"
int main(int argc, char**argv) {
  environment rheolef (argc,argv);
  geo omega (argv[1]);
  Float eps = numeric_limits<Float>::epsilon();
  string approx   = (argc > 2) ?       argv[2]  : "P1";
  Float  lambda   = (argc > 3) ? atof(argv[3]) : 0.1;
  Float  tol      = (argc > 4) ? atof(argv[4]) : eps;
  size_t max_iter = (argc > 5) ? atoi(argv[5]) : 100;
  combustion F (omega, approx);
  F.set_parameter (lambda);
  field uh = F.initial ();
  Float residue = tol;
  size_t n_iter = max_iter;
  damped_newton (F, uh, residue, n_iter, &derr);
  F.put (dout, uh);
  return (residue <= sqrt(tol)) ? 0 : 1;
}
```

Let us choose $\alpha = 1/2$ and $\lambda = 8(\alpha/\cosh(\alpha))^2 \approx 1.57289546593186$. Compilation and run are:

```
make combustion_newton
mkgeo_grid -e 10 > line-10.geo
./combustion_newton line-10 P1 1.57289546593186 > line-10.field
field line-10.field
```

3.3.2 Error analysis and multiplicity of solutions

In one dimension, when $\Omega =]0,1[$, the problem can be solved explicitly [Paumier, 1997, p. 59]:

- when $\lambda \leqslant 0$ the solution is parameterized by $\alpha \in]0, \pi/2[$ and:

$$\begin{aligned} \lambda &= -\frac{8\alpha^2}{\cos^2(\alpha)} \\ u(x) &= 2\log\left(\frac{\cos(\alpha)}{\cos(\alpha(1-2x))}\right),\ x \in]0,1[\end{aligned}$$

- when $0 \leqslant \lambda \leqslant \lambda_c$ there is two solutions. The smallest one is parameterized by $\alpha \in]0, \alpha_c]$ and: and the largest by $\alpha \in]\alpha_c, +\infty[$ with:

$$\begin{aligned} \lambda &= \frac{8\alpha^2}{\cosh^2(\alpha)} \\ u(x) &= 2\log\left(\frac{\cosh(\alpha)}{\cosh(\alpha(1-2x))}\right),\ x \in]0,1[\end{aligned}$$

- when $\lambda > \lambda_c$ there is no more solution.

The critical parameter value $\lambda_c = 8\alpha_c^2/\cosh^2(\alpha_c)$ where α_c is the unique positive solution to $\tanh(\alpha) = 1/\alpha$. The following code compute α_c and λ_c by using a Newton method.

File 3.25: `lambda_c.h`

```
struct alpha_c_fun {
  typedef Float value_type;
  typedef Float float_type;
  alpha_c_fun() : _f1(0) {}
  Float residue (const Float& a) const { return tanh(a) - 1/a; }
  void update_derivative (const Float& a) const {
                                _f1 = 1/sqr(cosh(a)) + 1/sqr(a); }
  Float derivative_solve (const Float& r) const { return r/_f1; }
  Float dual_space_norm  (const Float& r) const { return abs(r); }
  mutable Float _f1;
};
Float alpha_c() {
  Float tol = numeric_limits<Float>::epsilon();
  size_t max_iter = 100;
  alpha_c_fun f;
  Float ac = 1;
  newton (alpha_c_fun(), ac, tol, max_iter);
  return ac;
}
Float lambda_c() {
  Float ac = alpha_c();
  return 8*sqr(ac/cosh(ac));
}
```

File 3.26: `lambda_c.cc`

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
#include "lambda_c.h"
int main(int argc, char**argv) {
  environment rheolef (argc,argv);
  dout << setprecision(numeric_limits<Float>::digits10)
       << "alpha_c  = " << alpha_c()  << endl
       << "lambda_c = " << lambda_c() << endl;
}
```

Compilation and run write:

```
make lambda_c
./lambda_c
```

and then $\alpha_c \approx 1.19967864025773$ and $\lambda_c \approx 3.51383071912516$. The exact solution and its gradient at the limit point are computed by the following functions:

File 3.27: `combustion_exact.icc`

```
#include "lambda2alpha.h"
struct u_exact {
  Float operator() (const point& x) const {
    return 2*log(cosh(a)/cosh(a*(1-2*x[0]))); }
  u_exact (Float lambda, bool is_upper)
   : a(lambda2alpha(lambda,is_upper)) {}
  u_exact (Float a1) : a(a1) {}
  Float a;
};
struct grad_u {
  point operator() (const point& x) const {
    return point(4*a*tanh(a*(1-2*x[0]))); }
  grad_u (Float lambda, bool is_upper)
   : a(lambda2alpha(lambda,is_upper)) {}
  grad_u (Float a1) : a(a1) {}
  Float a;
};
```

The `lambda2alpha` function converts λ into α. When $0 < \lambda < \lambda_c$, there is two solutions to the equation $8\,(\alpha/\cosh(\alpha))^2 = \lambda$ and thus we specify with the Boolean `is_upper` which one is expected. Then α is computed by a dichotomy algorithm.

File 3.28: `lambda2alpha.h`

```
#include "lambda_c.h"
Float lambda2alpha (Float lambda, bool up = false) {
  static const Float ac = alpha_c();
  Float tol = 1e2*numeric_limits<Float>::epsilon();
  size_t max_iter = 1000;
  Float a_min = up ?  ac : 0;
  Float a_max = up ? 100 : ac;
  for (size_t k = 0; abs(a_max - a_min) > tol; ++k) {
    Float a1      = (a_max + a_min)/2;
    Float lambda1 = 8*sqr(a1/cosh(a1));
    if ((up && lambda > lambda1) || (!up && lambda < lambda1))
          { a_max = a1; }
    else  { a_min = a1; }
    check_macro (k < max_iter, "lambda2alpha: max_iter=" << k
                        << " reached and err=" << a_max - a_min);
  }
  return(a_max + a_min)/2;
};
```

Finally, the errors in various norms are available:

File 3.29: combustion_error.cc

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
#include "combustion_exact.icc"
int main(int argc, char**argv) {
  environment rheolef (argc,argv);
  bool is_upper = (argc > 1) && (argv[1][0] == '1');
  bool is_crit  = (argc > 1) && (argv[1][0] == 'c');
  Float tol     = (argc > 2) ? atof(argv[2]) : 1e-15;
  Float lambda_h;
  field uh;
  din >> catchmark("lambda") >> lambda_h
      >> catchmark("u")      >> uh;
  Float lambda = (is_crit ? lambda_c() : lambda_h);
  const geo& omega = uh.get_geo();
  const space& Xh = uh.get_space();
  field pi_h_u = lazy_interpolate (Xh, u_exact(lambda,is_upper));
  field eh = pi_h_u - uh;
  integrate_option iopt;
  iopt.set_family(integrate_option::gauss);
  iopt.set_order(2*Xh.degree()+1);
  Float err_l2
   = sqrt(integrate(omega, norm2(uh - u_exact(lambda,is_upper)), iopt));
  Float err_h1
   = sqrt(integrate(omega, norm2(grad(uh)-grad_u(lambda,is_upper)), iopt));
  Float err_linf = eh.max_abs();
  dout << "err_linf = " << err_linf << endl
       << "err_l2   = " << err_l2   << endl
       << "err_h1   = " << err_h1   << endl;
  return (err_h1 < tol) ? 0 : 1;
}
```

The computation of the error writes:

```
make combustion_error
./combustion_error < line-10.field
```

The solution is represented on Fig. 3.16. Then we consider the vicinity of $\lambda_c \approx 3.51383071912516$.

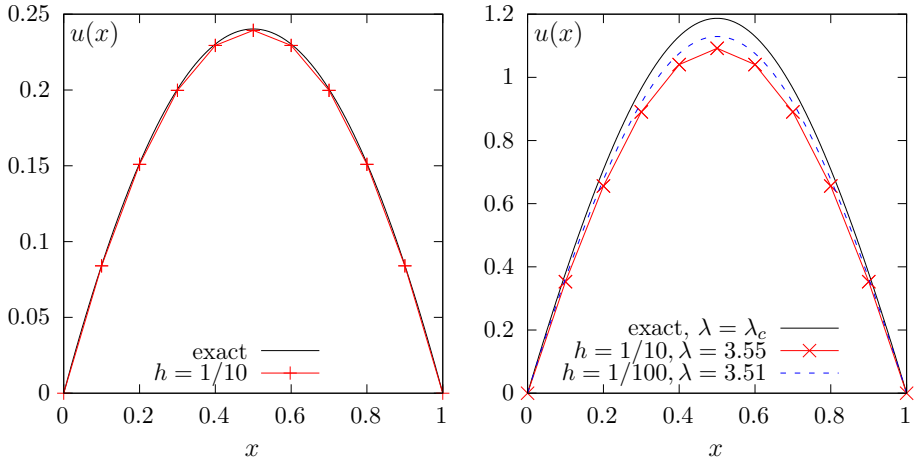


Figure 3.16: Combustion problem: (left) $\alpha = 1/2$. (right) near α_c.

```
./combustion_newton line-10  P1 3.55 > line-10.field
field line-10.field
mkgeo_grid -e 100 > line-100.geo
./combustion_newton line-100 P1 3.51 > line-100.field
```

The Newton method fails when the parameter λ is greater to some critical value $\lambda_{c,h}$ that depends upon the mesh size. Moreover, while the approximate solution is close to the exact one for moderate $\lambda = 1.57289546593186$, the convergence seems to be slower at the vicinity of λ_c.

In the next section, we compute accurately $\lambda_{c,h}$ for each mesh and observe the convergence of $\lambda_{c,h}$ to λ_c and the convergence of the associated approximate solution to the exact one.

3.3.3 The Euler-Newton continuation algorithm

The Euler-Newton continuation algorithm writes [Paumier, 1997, p. 176]:

algorithm 1 (*continuation*)

- $n = 0$: Let (λ_0, u_0) be given. Compute

$$\dot{u}_0 = -\left(\frac{\partial F}{\partial u}(\lambda_0, u_0)\right)^{-1} \frac{\partial F}{\partial \lambda}(\lambda_0, u_0)$$

- $n \geqslant 0$: Let (λ_n, u_n) and $\dot{u}_n$ being known.

  1) First choose a step $\Delta\lambda_n$ and set $\lambda_{n+1} = \lambda_n + \Delta\lambda_n$.

  2) Then, perform a prediction by computing

$$w_0 = u_n - \Delta\lambda_n \left(\frac{\partial F}{\partial u}(\lambda_n, u_n)\right)^{-1} \frac{\partial F}{\partial \lambda}(\lambda_n, u_n)$$

  3) Then, perform a correction step: for all $k \geqslant 0$, with w_k being known, compute

$$w_{k+1} = w_k - \left(\frac{\partial F}{\partial u}(\lambda_{n+1}, w_k)\right)^{-1} F(\lambda_{n+1}, w_k)$$

  At convergence of the correction loop, set $u_{n+1} = w_\infty$.

  4) Finally, compute

$$\dot{u}_{n+1} = -\left(\frac{\partial F}{\partial u}(\lambda_{n+1}, u_{n+1})\right)^{-1} \frac{\partial F}{\partial \lambda}(\lambda_{n+1}, u_{n+1})$$

The step $\Delta\lambda_n$ can be chosen from a guest $\Delta\lambda_* = \Delta\lambda_{n-1}$ by adjusting the contraction ratio $\kappa(\Delta\lambda_*)$ of the Newton method. Computing the two first iterates $w_{0,*}$ and $w_{1,*}$ with the guest step $\Delta\lambda_*$ and $\lambda_* = \lambda_n + \Delta\lambda_*$ we have:

$$\kappa(\Delta\lambda_*) = \frac{\left\| \left(\frac{\partial F}{\partial u}(\lambda_*, w_{1,*})\right)^{-1} F(\lambda_*, w_{1,*}) \right\|}{\left\| \left(\frac{\partial F}{\partial u}(\lambda_*, w_{0,*})\right)^{-1} F(\lambda_*, w_{0,*}) \right\|}$$

As the Newton method is expected to converge quadratically for small enough step, we get a practical expression for $\Delta\lambda_n$ [Paumier, 1997, p. 185]:

$$\frac{\kappa_0}{\Delta\lambda_n} \approx \frac{\kappa(\Delta\lambda_*)}{\Delta\lambda_*^2}$$

where $\kappa_0 \in]0, 1[$ is the chosen reference for the contraction ratio, for instance $\kappa_0 = 1/2$.

File 3.30: `combustion_continuation.cc`

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
#include "combustion.h"
int main(int argc, char**argv) {
  environment rheolef (argc, argv);
  cin >> noverbose;
  geo omega (argv[1]);
  string approx = (argc > 2) ? argv[2] : "P1";
  Float eps = numeric_limits<Float>::epsilon();
  continuation_option opts;
  opts.ini_delta_parameter = 0.1;
  opts.max_delta_parameter = 1;
  opts.min_delta_parameter = 1e-7;
  opts.tol                 = eps;
  derr << setprecision(numeric_limits<Float>::digits10)
       << "# continuation in lambda:" << endl
       << "# geo           = " << omega.name() << endl
       << "# approx        = " << approx << endl
       << "# dlambda_ini   = " << opts.ini_delta_parameter << endl
       << "# dlambda_min   = " << opts.min_delta_parameter << endl
       << "# dlambda_max   = " << opts.max_delta_parameter << endl
       << "# tol           = " << opts.tol << endl;
  combustion F (omega, approx);
  field uh = F.initial();
  F.put (dout, uh);
  continuation (F, uh, &dout, &derr, opts);
}
```

Then, the program is compiled and run as:

```
make combustion_continuation
mkgeo_grid -e 10 > line-10.geo
./combustion_continuation line-10 > line-10.branch
branch line-10.branch -toc
```

The last command lists all the computations preformed by the continuation algorithm. The last

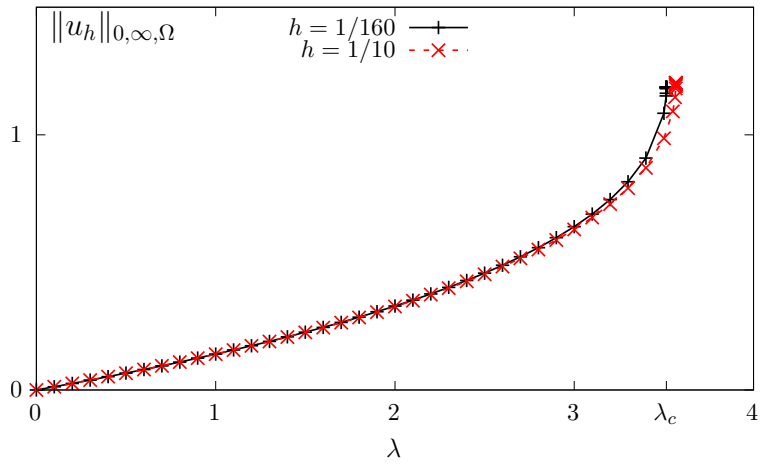


Figure 3.17: Combustion problem: $\|u_h\|_{0,\infty,\Omega}$ vs λ when $h = 1/10$ and $1/160$.

recorded computation is associated to the *limit point* denoted and denoted as $\lambda_{c,h}$, says at index 21: Let us visualize the solution u_h at the limit point and compute its maximum $\|u_h\|_{0,\infty,\Omega} = u_h(1/2)$:

```
branch line-10.branch -extract 21 -branch | field -
branch line-10.branch -extract 21 -branch | field -max -
```

Fig. 3.17 plots $\|u_h\|_{0,\infty,\Omega}$ versus λ for various meshes.

Fig. 3.18 plots its convergence to λ_c and also the convergence of the corresponding appoximate solution u_h to the exact one u, associated to λ_c. Observe that $|\lambda_{c,h} - \lambda_c|$ converges to zero

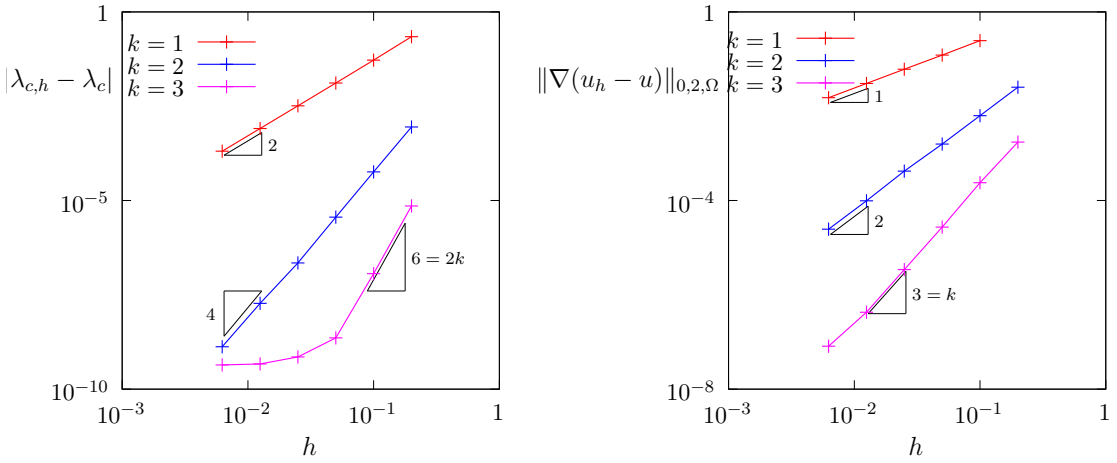


Figure 3.18: Combustion problem: (left) convergence of $|\lambda_{c,h} - \lambda_c|$ vs h ; (right) convergence of $|u_h - u|$ vs h at the limit point.

as $\mathcal{O}(h^{2k})$ while $|u_h - u| = \mathcal{O}(h^k)$. When $k = 3$, the convergence of $\lambda_{c,h}$ slows down around 10^{-9}: this is due to the stopping criterion of the Newton method that detects automatically the propagation of rounding effects in the resolution of linear systems. Observe on Fig. 3.19 the plot

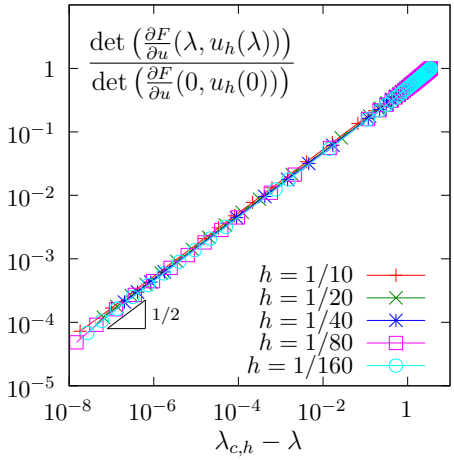


Figure 3.19: Combustion problem: $\det\left(\dfrac{\partial F}{\partial u}(\lambda, u_h)\right)$ versus $\lambda_{c,h} - \lambda$ for various h and $k = 1$.

of the determinant of the Jacobean $\dfrac{\partial F}{\partial u}(\lambda, u_h(\lambda))$ versus $\lambda_{c,h} - \lambda$: it tends to zero at the vicinity of $\lambda = \lambda_c$ and thus, the Newton method reaches increasing difficulties to solve the problem. Note that the determinant has been normalized by its value when $\lambda = 0$, i.e. $\det\left(\dfrac{\partial F}{\partial u}(0, u_h(0))\right)$: all the curves tend to superpose on a unique master curve and the asymptotic behavior is independent

of h. More precisely, Fig. 3.19 suggests a $1/2$ slope in logarithmic scale and thus

$$\frac{\det\left(\dfrac{\partial F}{\partial u}(\lambda, u_h(\lambda))\right)}{\det\left(\dfrac{\partial F}{\partial u}(0, u_h(0))\right)} \approx C\,(\lambda_{c,h} - \lambda)^{1/2}$$

where $C \approx 0.52$ when $k = 1$. This behavior explains that the Newton method is unable to reach the limit point λ_c.

3.3.4 Beyond the limit point : the Keller algorithm

Note that the continuation method stops to the limit point (see Fig. 3.17) while the the branch continues: the limit point is a turning point for the branch of solutions. Especially, for each $\lambda \in]0, \lambda_c[$ there are two solutions and only one has been computed. Keller proposed a method to follow the branch beyond a turning point and this method is presented here. The main idea is to parameterize the branch $(\lambda, u(\lambda))$ by a curvilinear abscissa s as $(\lambda(s), u(s))$. In order to have the count of unknown and equations we add a normalization equation:

$$\begin{aligned} \mathcal{N}(s, \lambda(s), u(s)) &= 0 \\ F(\lambda(s), u(s)) &= 0 \end{aligned}$$

where $\mathcal{N}$ is a given normalization function. For the normalization function, Keller proposed to choose, when (s, λ, u) is at the vicinity of $(s_n, \lambda(s_n), u(s_n))$ the one following orthogonal norms:

- The **orthogonal** norm:

$$\begin{aligned} \mathcal{N}_n(s,\, \chi{=}(\lambda, u)) &= (\chi'(s_n),\, \chi - \chi(s_n)) - (s - s_n) \\ &= \lambda'(s_n)\,(\lambda - \lambda(s_n)) + (u'(s_n),\, u - u(s_n))_V - (s - s_n) \end{aligned} \tag{3.6a}$$

- The **spherical** norm:

$$\begin{aligned} \widetilde{\mathcal{N}}_n(s,\, \chi{=}(\lambda, u)) &= \|\chi - \chi_n\|^2 - |s - s_n|^2 \\ &= |\lambda - \lambda_n|^2 + \|u - u_n\|_V^2 - |s - s_n|^2 \end{aligned} \tag{3.6b}$$

The orthogonal norm induces a pseudo curvilinear arc-length s, measured on the tangent at $s = s_n$. The spherical norm measures is simply a distance between (s, χ) and (s_n, χ_n) (see also Paumier, 1997, pp. 179-180 and the corresponding Fig. 5.2). We add the subscript n to $\mathcal{N}$ in order to emphasize that $\mathcal{N}$ depends upon both $(\lambda(s_n), u(s_n))$ and $(\lambda'(s_n), u'(s_n))$ For any $s \in \mathbb{R}$ and $\chi = (\lambda, u) \in \mathbb{R} \times V$ we introduce:

$$\mathcal{F}_n(s, \chi) = \begin{pmatrix} \mathcal{N}_n(s, \chi) \\ F(\chi) \end{pmatrix} \quad \text{and} \quad \widetilde{\mathcal{F}}_n(s, \chi) = \begin{pmatrix} \widetilde{\mathcal{N}}_n(s, \chi) \\ F(\chi) \end{pmatrix}$$

Then, the Keller problem with the orthogonal norm reduces to find, for any $s \in \mathbb{R}$, $\chi(s) \in \mathbb{R} \times V$ such that

$$\mathcal{F}_n(s, \chi(s)) = 0 \tag{3.7a}$$

Conversely, the Keller problem with the spherical norm reduces to find, for any $s \in \mathbb{R}$, $\chi(s) \in \mathbb{R} \times V$ such that

$$\widetilde{\mathcal{F}}_n(s, \chi(s)) = 0 \tag{3.7b}$$

Both problems falls into the framework of the previous paragraph, when F is replaced by either $\mathcal{F}_n$ or $\widetilde{\mathcal{F}}_n$. Then, for any s and $\chi = (\lambda, u)$, the partial derivatives are:

$$\begin{aligned}
\frac{\partial \mathcal{F}_n}{\partial s}(s,\chi) &= \begin{pmatrix} -1 \\ 0 \end{pmatrix} \\
\frac{\partial \mathcal{F}_n}{\partial \chi}(s,\chi) &= \begin{pmatrix} \dfrac{\partial \mathcal{N}_n}{\partial \chi}(s,\chi) \\ \\ F'(s,\chi) \end{pmatrix} = \begin{pmatrix} \lambda'(s_n) & u'(s_n) \\ \\ \dfrac{\partial F}{\partial \lambda}(\lambda,u) & \dfrac{\partial F}{\partial u}(\lambda,u) \end{pmatrix}
\end{aligned}$$

and

$$\begin{aligned}
\frac{\partial \widetilde{\mathcal{F}}_n}{\partial s}(s,\chi) &= \begin{pmatrix} -2(s-s_n) \\ 0 \end{pmatrix} \\
\frac{\partial \widetilde{\mathcal{F}}_n}{\partial \chi}(s,\chi) &= \begin{pmatrix} 2(\chi-\chi_n)^T \\ \\ F'(\chi) \end{pmatrix} = \begin{pmatrix} 2(\lambda-\lambda_n) & 2(u-u_n)^T \\ \\ \dfrac{\partial F}{\partial \lambda}(\lambda,u) & \dfrac{\partial F}{\partial u}(\lambda,u) \end{pmatrix}
\end{aligned}$$

Let us focus on the orthogonal norm case, as the spherical one is similar. The continuation algorithm of the previous paragraph is able to follows the branch of solution beyond the limit point and explore the second part of the branch. Let us compute $\lambda'(s_n)$ and $u'(s_n)$. By differentiating (3.7) with respect to s, we get:

$$\frac{\partial \mathcal{F}_n}{\partial s}(s,\chi(s)) + \frac{\partial \mathcal{F}_n}{\partial \chi}(s,\chi(s)).(\chi'(s)) = 0$$

that writes equivalently

$$\begin{aligned}
\frac{\partial \mathcal{N}_n}{\partial s}(s,\chi(s)) + \frac{\partial \mathcal{N}_n}{\partial \chi}(s,\chi(s)).(\chi'(s)) &= 0 \\
F'(s,\chi(s)).(\chi'(s)) &= 0
\end{aligned}$$

Using the expression 3.6 for $\mathcal{N}_n$ we obtain:

$$-1 + \lambda'(s_n)\lambda'(s) + (u'(s_n), u'(s)) = 0 \tag{3.8}$$

$$\frac{\partial F}{\partial \lambda}(\lambda(s),u(s))\,\lambda'(s) + \frac{\partial F}{\partial u}(\lambda(s),u(s)).(u'(s)) = 0 \tag{3.9}$$

Here $(.,.)$ denotes the scalar product of the V space for u. Let us choose $s = s_n$, for any $n \geqslant 0$: we obtain

$$\begin{aligned}
|\lambda'(s_n)|^2 + \|u'(s_n)\|^2 &= 1 \\
\frac{\partial F}{\partial \lambda}(\lambda_n,u_n)\,\lambda'(s_n) + \frac{\partial F}{\partial u}(\lambda_n,u_n).(u'(s_n)) &= 0
\end{aligned}$$

where we use the notations $\lambda_n = \lambda(s_n)$ and $u_n = u(s_n)$, and where $\|.\|$ denotes the norm of the V space. Thus

$$\begin{aligned}
\chi'(s_n) &= \begin{pmatrix} \lambda'(s_n) \\ u'(s_n) \end{pmatrix} \\
&= \frac{1}{\left(1 + \left\| \left(\dfrac{\partial F}{\partial u}(\lambda_n,u_n)\right)^{-1} \dfrac{\partial F}{\partial \lambda}(\lambda_n,u_n) \right\|^2\right)^{1/2}} \begin{pmatrix} 1 \\ -\left(\dfrac{\partial F}{\partial u}(\lambda_n,u_n)\right)^{-1} \dfrac{\partial F}{\partial \lambda}(\lambda_n,u_n) \end{pmatrix}
\end{aligned}$$

The previous relation requires $\dfrac{\partial F}{\partial u}(\lambda_n, u_n)$ to be nonsingular, e.g. the computation is not possible at a singular point (λ_n, u_n).

For a singular point, suppose that $n \geqslant 1$ and that both (λ_n, u_n), (λ_{n-1}, u_{n-1}) and $(\dot\lambda_{n-1}, \dot u_{n-1})$ are known. By differentiating (3.7) at step $n-1$ we get the equivalent of (3.8)-(3.9) at step $n-1$ that is then evaluated for $s = s_n$. We get:

$$\begin{aligned} \lambda'(s_{n-1})\lambda'(s_n) + (u'(s_{n-1}), u'(s_n)) &= 1 \\ \frac{\partial F}{\partial \lambda}(\lambda_n, u_n)\,\lambda'(s_n) + \frac{\partial F}{\partial u}(\lambda_n, u_n).(u'(s_n)) &= 0 \end{aligned}$$

that writes equivalently

$$\begin{pmatrix} \lambda'(s_{n-1}) & u'(s_{n-1}) \\ \dfrac{\partial F}{\partial \lambda}(\lambda_n, u_n) & \dfrac{\partial F}{\partial u}(\lambda_n, u_n) \end{pmatrix} \begin{pmatrix} \lambda'(s_n) \\ u'(s_n) \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \tag{3.10}$$

The matrix involved in the left hand side is exactly the Jacobean of $\mathcal{F}_{n-1}$ evaluated at point $\chi_n = (\lambda_n, u_n)$. This Jacobean is expected to be nonsingular at a simple limit point.

Thus, at the first step, we suppose that the initial point (λ_0, u_0) is non-singular and we compute $(\dot\lambda_0, \dot u_0)$. Then, at the begining of the n-th step, $n \geqslant 1$, of the Keller continuation algorithm, we suppose that both (λ_{n-1}, u_{n-1}) and $(\dot\lambda_{n-1}, \dot u_{n-1})$ are known. We consider the problem $\mathcal{F}_{n-1}(s, \chi(s)) = 0$. Here, $\mathcal{F}_{n-1}(s, \chi)$ is completely defined at the vicinity of $(s_{n-1}, \lambda_{n-1}, u_{n-1})$. The step control procedure furnishes as usual a parameter step Δs_{n-1} and we set $s_n = s_{n-1} + \Delta s_{n-1}$. The Newton method is performed and we get (λ_n, u_n). Finaly, we compute $\dot\lambda_n$ and $\dot u_n$ from (3.10).

Recall that the function $\mathcal{F}_{n-1}$ depends upon n and should be *refreshed* at the begining of each iteration by using the values $(\dot\lambda_{n-1}, \dot u_{n-1})$.

The Keller continuation algorithm writes:

algorithm 2 (*Keller continuation*)

- $n = 0$: Let $(s_0,\ \chi_0 = (\lambda_0, u_0))$ be given. The recurrence requires also $\dot\chi_0$ and its orientation $\varepsilon_0 \in \{-1, +1\}$: they could either be given or computed. When computing $(\dot\chi_0, \varepsilon_0)$, the present algorithm supposes that (λ_0, u_0) is a regular point: on a singular point, e.g. a bifurcation one, there a several possible directions, and one should be chosen. Then, choose $\varepsilon = \pm 1$ and compute $\dot\chi_0$ in three steps:

$$\begin{aligned} \frac{\mathrm{d}u}{\mathrm{d}\lambda}(\lambda_0) &= -\left(\frac{\partial F}{\partial u}(\lambda_0, u_0)\right)^{-1} \frac{\partial F}{\partial \lambda}(\lambda_0, u_0) \\ c &= \left(1 + \left\| \frac{\mathrm{d}u}{\mathrm{d}\lambda}(\lambda_0) \right\|\right)^{-1/2} \\ \dot\chi_0 &\overset{def}{=} (\dot\lambda_0,\ \dot u_0)^T = c \left(1,\ \frac{\mathrm{d}u}{\mathrm{d}\lambda}(\lambda_0)\right)^T \end{aligned}$$

- $n \geqslant 0$: Let $(s_n,\ \chi_n = (\lambda_n, u_n))$, $\dot\chi_n = (\dot\lambda_n, \dot u_n)$ and ε_n being known.

  1) First choose a step Δs_n and set $s_{n+1} = s_n + \Delta s_n$, as in the classical continuation algorithm

  2) Then, perform a prediction, as usual:

$$y_0 = \chi_n + \varepsilon_n \Delta s_n \dot\chi_n$$

  3) Also as usual, do a correction loop: for all $k \geqslant 0$, y_k being known, compute

$$y_{k+1} = y_k - \left(\frac{\partial \mathcal{F}_n}{\partial \chi}(s_{n+1}, y_k)\right)^{-1} \mathcal{F}_n(s_{n+1}, y_k)$$

At convergence of the correction loop, set $\chi_{n+1} = y_\infty$. This Newton correction loop can be replaced by a damped Newton one.

4) Check : if $n \geqslant 1$, compute the following angle cosinus:

$$\begin{aligned} c_1 &= (\chi_{n+1} - \chi_n,\, \chi_n - \chi_{n-1}) \\ &= (\lambda_{n+1} - \lambda_n)(\lambda_n - \lambda_{n-1}) + (u_{n+1} - u_n,\, u_n - u_{n-1})_V \\ c_2 &= (\dot\chi_n,\, \chi_{n+1} - \chi_n) \\ &= (\dot\lambda_n,\, \lambda_{n+1} - \lambda_n) + (\dot u_n,\, u_{n+1} - u_n)_V \end{aligned}$$

When either $c_1 \leqslant 0$ or $c_2 \leqslant 0$, then decreases Δs_n and go back at step 1. Otherwise the computation of χ_{n+1} is validated.

5) Finally, compute $\dot\chi_{n+1} = (\dot\lambda_{n+1}, \dot u_{n+1})$ as:

$$\dot\chi_{n+1} = -\left(\frac{\partial \mathcal{F}_n}{\partial \chi}(s_{n+1}, \chi_{n+1})\right)^{-1} \frac{\partial \mathcal{F}_n}{\partial s}(s_{n+1}, \chi_{n+1})$$

If $\varepsilon_n\ (\dot\chi_{n+1},\, \dot\chi_n) \geqslant 0$ then set $\varepsilon_{n+1} = \varepsilon_n$ else $\varepsilon_{n+1} = -\varepsilon_n$.

The Keller algorithm with the spherical norm is simply obtained by replacing $\mathcal{F}_n$ by $\widetilde{\mathcal{F}}_n$ Both algorithm variants still require to save $\dot\chi_n$ and ε_n at each step for restarting nicely with the same sequence of computation. The only drawback is that, at a restart of the algorithm, we skip the first Check step, and its possible to go back at the first iterate if Δs_0 is too large. A possible remedy is, when restarting, to furnish two previous iterates, namely χ_{-1} and χ_0, together with $\dot\chi_0$: χ_{-1} is used only for the Check of a possible change of direction.

Howell [2009] suggested that the Keller algorithm with spherical norm is more robust that its variant with orthogonal one. Howell [2009] reported that

> *[...] the spherical constraint N_2 is much more efficient than the orthogonal constraint N_1, as N_2 required only one step of length 0.01 to exceed the target value, while N_1 required 25, with 8 additional convergence failures.*

> *[...] spherical constraint was seen to be more efficient than an orthogonal constraint in a region of high curvature of the solution manifold, while both constraints performed similarly in regions of low or moderate curvature.*

File 3.31: `combustion_keller.cc`

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
#include "combustion.h"
int main(int argc, char**argv) {
  environment rheolef (argc, argv);
  din >> noverbose;
  geo omega (argv[1]);
  string approx = (argc > 2) ? argv[2] : "P1";
  string metric = (argc > 3) ? argv[3] : "orthogonal";
  Float eps = numeric_limits<Float>::epsilon();
  continuation_option opts;
  opts.ini_delta_parameter = 0.1;
  opts.max_delta_parameter = 0.5;
  opts.min_delta_parameter = 1e-10;
  opts.tol                 = eps;
  derr << setprecision(numeric_limits<Float>::digits10)
       << "# continuation in s:" << endl
       << "# geo       = " << omega.name() << endl
       << "# approx    = " << approx << endl
       << "# metric    = " << metric << endl
       << "# ds_init   = " << opts.ini_delta_parameter << endl
       << "# ds_min    = " << opts.min_delta_parameter << endl
       << "# ds_max    = " << opts.max_delta_parameter << endl
       << "# tol       = " << opts.tol << endl;
  dout << catchmark("metric") << metric << endl;
  keller<combustion> F (combustion(omega,approx), metric);
  keller<combustion>::value_type xh = F.initial();
  F.put (dout, xh);
  continuation (F, xh, &dout, &derr, opts);
}
```

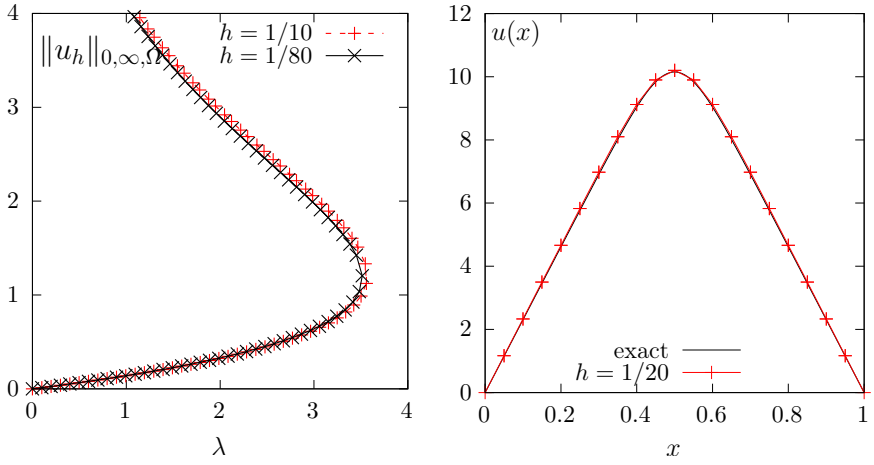


Figure 3.20: Combustion problem: (left) $\|u_h\|_{0,\infty,\Omega}$ vs λ when $h = 1/10$ and $1/80$. The full branch of solutions when $\lambda \geqslant 0$, with the limit point and the upper part of the branch. (right) solution of the upper branch when $\lambda = 10^{-2}$.

How to run the program

Enter the following unix commands:

```
make combustion_keller combustion_keller_post combustion_error
mkgeo_grid -e 20 > line-20.geo
```

```
./combustion_keller line-20 > line-20.branch
./combustion_keller_post < line-20.branch
```

The last command scans the file `line-20.branch` containing the branch of solutions $(\lambda(s), u(s))$ and compute some useful informations for graphical representations. Observe on Fig. 3.20.left the full branch of solutions when $\lambda \geqslant 0$, with the limit point and the upper part of the branch. Compare it with Fig. 3.17, where the continuation algorithm was limited to the lower part part of the branch. Next, in order to vizualise the last computed solution, enter:

```
branch line-20.branch -toc
branch line-20.branch -extract 41 -branch | ./combustion_error 1
```

Please, replace 41 by the last computed index on the branch. Fig. 3.20.right represents this solution and compares it ith the exact one. Observe the excellent agreement.

File 3.32: `combustion_keller_post.cc`

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
#include "combustion.h"
int main(int argc, char**argv) {
  environment rheolef (argc,argv);
  string metric;
  din >> catchmark("metric") >> metric;
  keller<combustion> F (combustion(), metric);
  keller<combustion>::value_type xh, dot_xh;
  dout << noverbose
       << setprecision(numeric_limits<Float>::digits10)
       << "# metric " << metric << endl
       << "# s lambda umax det(mantissa,base,exp) |u| |grad(u)| |residue|"
       << endl;
  for (size_t n = 0; F.get(din,xh); ++n) {
    problem::determinant_type det;
    if (n > 0 || metric != "spherical") det = F.update_derivative(xh);
    const space& Xh = xh.second.get_space();
    trial u (Xh); test v (Xh);
    form a = integrate(dot(grad(u),grad(v))),
         m = integrate(u*v);
    const combustion& F0 = F.get_problem();
    field mrh = F0.residue(xh.second);
    dout << F.parameter() << " " << xh.first
         << " " << xh.second.max_abs()
         << " " << det.mantissa
         << " " << det.base
         << " " << det.exponant
         << " " << sqrt(m(xh.second,xh.second))
         << " " << sqrt(a(xh.second,xh.second))
         << " " << sqrt(F0.dual_space_dot (mrh,mrh))
         << endl;
    dot_xh = F.direction (xh);
    F.refresh (F.parameter(), xh, dot_xh);
  }
}
```

Chapter 4

Discontinuous Galerkin methods

4.1 Linear first-order problems

The aim of this chapter is to introduce to discontinuous Galerkin methods within the **Rheolef** environment. For some recent presentations of discontinuous Galerkin methods, see di Pietro and Ern [2012] for theoretical aspects and Hesthaven and Warburton [2008] for algorithmic and implementation.

4.1.1 The stationary transport equation

The steady scalar transport problem writes:

(P): *find ϕ, defined in Ω, such that*

$$\begin{aligned} \mathbf{u}.\nabla\phi + \sigma\phi &= f \quad \text{in } \Omega \\ \phi &= \phi_\Gamma \quad \text{on } \partial\Omega_- \end{aligned}$$

where $\mathbf{u}$, $\sigma > 0$, f and ϕ_Γ being known. Note that this is the steady version of the unsteady diffusion-convection problem previously introduced in section 2.4.2, page 79 and when the diffusion coefficient ν vanishes. Here, the $\partial\Omega_-$ notation is the *upstream* boundary part, defined by

$$\partial\Omega_- = \{x \in \partial\Omega;\ \mathbf{u}(x).\mathbf{n}(x) < 0\}$$

Let us suppose that $\mathbf{u} \in W^{1,\infty}(\Omega)^d$ and introduce the space:

$$X = \{\varphi \in L^2(\Omega);\ (\mathbf{u}.\nabla)\varphi \in L^2(\Omega)^d\}$$

and, for all $\phi, \varphi \in X$

$$\begin{aligned} a(\phi,\varphi) &= \int_\Omega (\mathbf{u}.\nabla\phi\,\varphi + \sigma\,\phi\,\varphi)\ \mathrm{d}x + \int_{\partial\Omega} \max(0, -\mathbf{u}.\mathbf{n})\,\phi\,\varphi\,\mathrm{d}s \\ l(\varphi) &= \int_\Omega f\,\varphi\,\mathrm{d}x + \int_{\partial\Omega} \max(0, -\mathbf{u}.\mathbf{n})\,\phi_\Gamma\,\varphi\,\mathrm{d}s \end{aligned}$$

Then, the variational formulation writes:

(FV): *find $\phi \in X$ such that*

$$a(\phi,\varphi) = l(\varphi),\ \forall\varphi \in X$$

Note that the term $\max(0, -\mathbf{u}.\mathbf{n}) = (|\mathbf{u}.\mathbf{n}| - \mathbf{u}.\mathbf{n})/2$ is positive and vanishes everywhere except on $\partial\Omega_-$. Thus, the boundary condition $\phi = \phi_\Gamma$ is weakly imposed on $\partial\Omega_-$ via the integrals on the boundary. The *discontinuous* finite element space is defined by:

$$X_h = \{\varphi_h \in L^2(\Omega); \varphi_{h|K} \in P_k,\ \forall K \in \mathcal{T}_h\}$$

where $k \geqslant 0$ is the polynomial degree. Note that $X_h \not\subset X$ and that the $\nabla\phi_h$ term has no more sense for discontinuous functions $\phi_h \in X_h$. Following di Pietro and Ern [2012, p. 14], we introduce the *broken gradient* ∇_h as a convenient notation:

$$(\nabla_h \phi_h)_{|K} = \nabla(\phi_{h|K}),\ \forall K \in \mathcal{T}_h$$

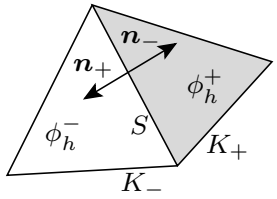
Thus

$$\int_\Omega \mathbf{u}.\nabla_h\phi_h\,\varphi_h\,\mathrm{d}x = \sum_{K\in\mathcal{T}_h} \int_K \mathbf{u}.\nabla\phi_h\,\varphi_h\,\mathrm{d}x,\ \forall \phi_h, \varphi_h \in X_h$$

This leads to a discrete version a_h of the bilinear form a, defined for all $\phi_h, \varphi_h \in X_h$ by (see e.g. di Pietro and Ern, 2012, p. 57, eqn. (2.34)):

$$\begin{aligned} a_h(\phi_h,\varphi_h) &= \int_\Omega (\mathbf{u}.\nabla_h\phi_h\,\varphi_h + \sigma\phi_h\ \varphi_h)\ \mathrm{d}x + \int_{\partial\Omega} \max(0, -\mathbf{u}.\mathbf{n})\,\phi_h\,\varphi_h\,\mathrm{d}s \\ &\quad + \sum_{S\in\mathcal{S}_h^{(i)}} \int_S \Big(-\mathbf{u}.\mathbf{n}\,[\![\phi_h]\!]\,\{\!\{\varphi_h\}\!\} + \frac{\alpha}{2}|\mathbf{u}.\mathbf{n}|\,[\![\phi_h]\!]\,[\![\varphi_h]\!]\Big)\ \mathrm{d}s \end{aligned}$$

The last term involves a sum over $\mathcal{S}_h^{(i)}$, the set of *internal sides* of the mesh $\mathcal{T}_h$. Each internal side



$\boldsymbol{n} = \boldsymbol{n}_- = -\boldsymbol{n}_+$ on S

Figure 4.1: Discontinuous Galerkin method: an internal side, its two neighbor elements and their opposite normals.

$S \in \mathcal{S}_h^{(i)}$ has two possible orientations: one is choosen definitively. In practice, this orientation is defined in the '`.geo`' file containing the mesh, where all sides are listed, together with their orientation. Let $\mathbf{n}$ the normal to the oriented side S: as S is an internal side, there exists two elements K_- and K_+ such that $S = \partial K_- \cap \partial K_+$ and $\mathbf{n}$ is the outward unit normal of K_- on $\partial K_- \cap S$ and the inward unit normal of K_+ on $\partial K_+ \cap S$, as shown on Fig. 4.1. For all $\phi_h \in X_h$, recall that ϕ_h is in general discontinuous across the internal side S. We define on S the *inner value* $\phi_h^- = \phi_{h|K_-}$ of ϕ_h as the restriction $\phi_{h|K_-}$ of ϕ_h in K_- along $\partial K_- \cap S$. Conversely, we define the *outer value* $\phi_h^+ = \phi_{h|K_+}$. We also denote on S the *jump* $[\![\phi_h]\!] = \phi_h^- - \phi_h^+$ and the *average* $\{\!\{\phi_h\}\!\} = (\phi_h^- + \phi_h^+)/2$. The last term in the definition of a_h is pondered by a coefficient $\alpha \geqslant 0$. Choosing $\alpha = 0$ correspond to the so-called *centered flux* approximation, while $\alpha = 1$ is the *upwinding flux* approximation. The case $\alpha = 1$ and $k = 0$ (piecewise constant approximation) leads to the popular upwinding finite volume scheme. Finally, the discrete variational formulation writes:

$(FV)_h$: *find* $\phi_h \in X_h$ *such that*

$$a_h(\phi_h,\varphi_h) = l(\varphi_h),\ \forall \varphi_h \in X_h$$

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

File 4.1: transport_dg.cc

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
int main(int argc, char**argv) {
  environment rheolef (argc, argv);
  geo omega (argv[1]);
  space Xh (omega, argv[2]);
  Float alpha   = (argc > 3) ? atof(argv[3]) : 1;
  Float sigma   = (argc > 4) ? atof(argv[4]) : 3;
  point u (1,0,0);
  trial phi (Xh); test psi (Xh);
  form ah = integrate (dot(u,grad_h(phi))*psi + sigma*phi*psi)
          + integrate ("boundary", max(0., -dot(u,normal()))*phi*psi)
          + integrate ("internal_sides",
                - dot(u,normal())*jump(phi)*average(psi)
                + 0.5*alpha*abs(dot(u,normal()))*jump(phi)*jump(psi));
  field lh = integrate ("boundary", max(0, -dot(u,normal()))*psi);
  field phi_h(Xh);
  problem p (ah);
  p.solve (lh, phi_h);
  dout << catchmark("sigma") << sigma << endl
       << catchmark("phi")   << phi_h;
}
```

Comments

The data are $f = 0$, $\phi_\gamma = 1$ and $\mathbf{u} = (1, 0, 0)$, and then the exact solution is known: $\phi(x) = \exp(-\sigma x_0)$. The numerical tests are running with $\sigma = 3$ by default. The one-dimensional case writes:

```
make transport_dg
mkgeo_grid -e 10 > line.geo
./transport_dg line P0  | field -
./transport_dg line P1d | field -
./transport_dg line P2d | field -
```

Observe the jumps across elements: these jumps decreases with mesh refinement or when polynomial approximation increases. The two-dimensional case writes:

```
mkgeo_grid -t 10 > square.geo
./transport_dg square P0  | field -
./transport_dg square P1d | field -elevation -
./transport_dg square P2d | field -elevation -
```

The elevation view shows details on inter-element jumps. Finaly, the three-dimensional case writes:

```
mkgeo_grid -T 5 > cube.geo
./transport_dg cube P0  | field -
./transport_dg cube P1d | field -
./transport_dg cube P2d | field -
```

Fig. 4.2 plots the solution when $d = 1$ and $k = 0$: observe that the boundary condition $\phi = 1$ at $x_0 = 0$ is only weakly satisfied. It means that the approximation $\phi_h(0)$ tends to 1 when h tends to zero. Fig. 4.3 plots the error $\phi - \phi_h$ in L^2 and L^∞ norms: these errors behave as $\mathcal{O}\left(h^{k+1}\right)$ for all $k \geqslant 0$, which is optimal. A theoretical $\mathcal{O}\left(h^{k+1/2}\right)$ error bound was shown by Johnson and Pitkäranta [1986]. The present numerical results confirm that these theoretical error bounds can be improved for some families of meshes, as pointed out by Richter [1988], that showed a $\mathcal{O}\left(h^{k+1}\right)$ optimal bound for the transport problem. This result was recently extended by Cockburn et al.

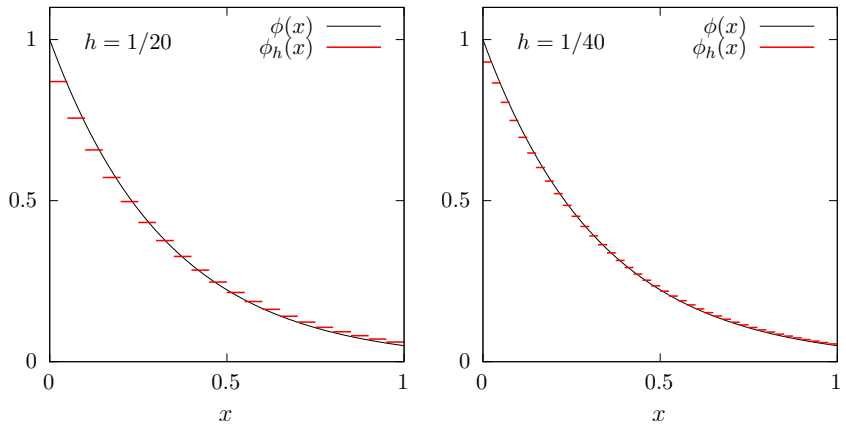


Figure 4.2: The discontinuous Galerkin method for the transport problem when $k = 0$ and $d = 1$.

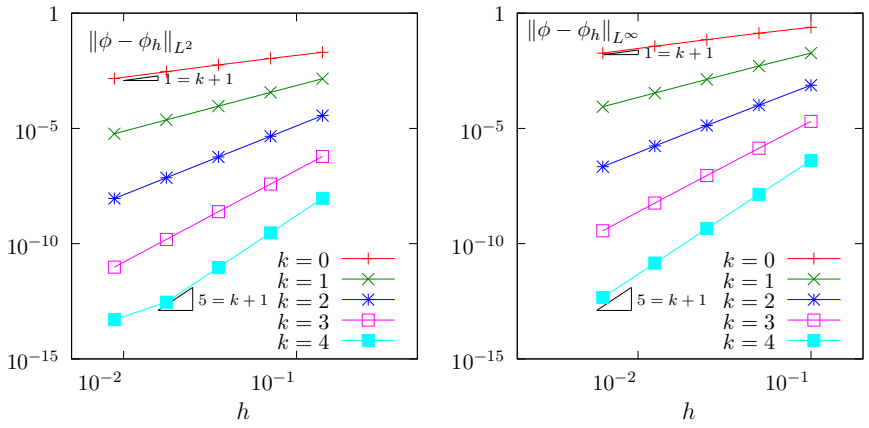


Figure 4.3: The discontinuous Galerkin method for the transport problem: convergence when $d = 2$.

[2010a], while Peterson [1991] showed that the estimate $\mathcal{O}\left(h^{k+1/2}\right)$ is sharp for general families of quasi-uniform meshes.

4.1.2 [New] The time-dependent transport equation

Problem statement

The time-dependent transport equation is involved in many problems of mathematical physics. One of these problem concerns a moving domain, transported by a velocity field. Let us denote, at any time $t \geqslant 0$ by $\omega(t) \subset \mathbb{R}^d$, $d = 2, 3$, a bounded moving domain and by $\Gamma(t) = \partial\omega(t)$ its boundary, called the *interface*. Let $\Omega \subset \mathbb{R}^d$ be the bounded computational domain. We assume that it contains at any time $t \geqslant 0$ the moving domain $\omega(t)$. A function $\phi : [0, \infty[\times\Omega \to \mathbb{R}$ is a *level set* for the moving domain if and only if $\Gamma(t) = \{\boldsymbol{x} \in \Omega \ / \ \phi(t, \boldsymbol{x}) = 0\}$ for any $t \geqslant 0$. For a given velocity field $\boldsymbol{u}$, associated to the motion, the level set function is defined as the solution of the following time dependent transport problem:

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

$$\begin{cases} \dfrac{\partial \phi}{\partial t} + \boldsymbol{u}.\nabla\phi = 0 \quad \text{in }]0,\infty[\times\Omega & (4.1a) \\ \phi(t{=}0) = \phi_0 \quad \text{in } \Omega & (4.1b) \end{cases}$$

where ϕ_0 is given and represents the initial shape $\omega(0)$. Here, we assume a divergence free velocity field, such that the volume of the moving domain is conserved. This is a linear *hyperbolic problem*. The level set method for describing moving interfaces was introduced by Osher and Sethian [1988] (see also Sethian, 1999).

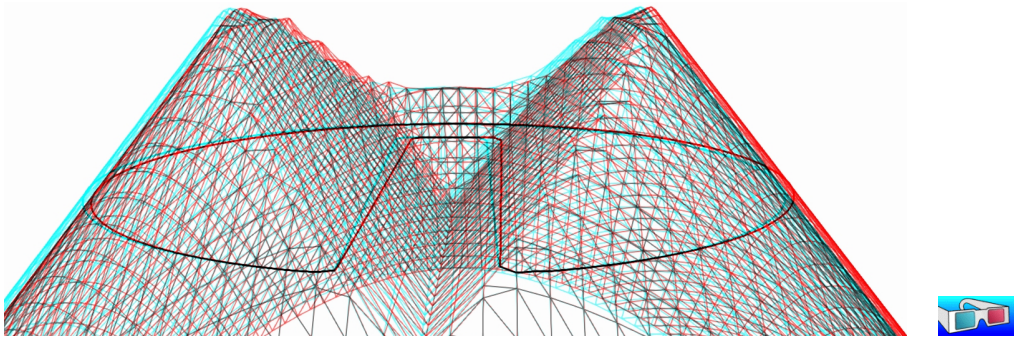


Figure 4.4: Zalesak slotted disk: elevation view of the interpolated level set function (upside-down), together with its zero level set, in black thick line.

Note that several choices are possible for the function ϕ: the only requirement being that a fixed isocontour of ϕ coincides with the front at each time t. A common choice is the signed distance from the front: e.g. $\omega(t)$ is the part where $\phi(t,.)$ is negative and $\phi(t,\boldsymbol{x}) = -\text{dist}(\Gamma(t),\boldsymbol{x})$ for all $\boldsymbol{x} \in \omega(t)$. Fig. 4.4, page 151, represents in elevation such a signed distance level set function: the interface Γ, the thick black line, corresponds to the Zalesak slotted disk, that will be our benchmark in the next section.

Approximation

The evolution transport equation (4.1a) is then discretized with respect to time. di Pietro et al. [2006] and Marchandise et al. [2006] used an *explicit* and *strong stability preserving* Runge-Kutta method [Gottlieb et al., 2001]. Here, we propose to use a high order *implicit* scheme based on the following *backward differentiation formula* (BDF):

$$\frac{\partial \phi}{\partial t}(t) = \frac{1}{\Delta t}\sum_{i=0}^{p} \alpha_{p,i}\ \phi(t{-}i\Delta t) + \mathcal{O}\left(\Delta t^p\right)$$

where $\Delta t > 0$ is the time step, $p \geqslant 1$ is the order of the scheme and $(\alpha_{p,i})_{0\leqslant i\leqslant p}$ are the $p+1$ coefficients of the formula. When $p = 1$, the scheme coincides with the usual implicit Euler method. For $1 \leqslant p \leqslant 2$, the scheme is unconditionally stable, for $3 \leqslant p \leqslant 6$, the scheme is almost unconditionally stable while when $p > 6$, the scheme is unstable and cannot be used (see e.g. Süli and Mayers, 2003, p. 349).

The coefficients when $p \leqslant 6$ are given in table 4.1 and the file '`bdf.icc`' implements this table. This file it is not listed here but is available in the **Rheolef** example directory. When the time step $n < p$, the scheme is started by using the BDF(n) formula instead and we introduce $p_n = \min(p,n)$ for convenience. The discretization with respect to space follows the method developed in the previous section 4.1.1. Let us introduce the following linear and bilinear form,

$p\backslash i$	0	1	2	3	4	5	6
1	1	-1					
2	3/2	-2	1/2				
3	11/6	-3	3/2	$-1/3$			
4	25/12	-4	3	$-4/3$	1/4		
5	137/60	-5	5	$-10/3$	5/4	$-1/5$	
6	147/60	-6	15/2	$-20/3$	15/4	$-6/5$	1/6

Table 4.1: BDF(p) schemes: the $\alpha_{p,i}$ coefficients, $1 \leqslant p \leqslant 6$, $1 \leqslant i \leqslant p$.

defined for all $\varphi, \xi \in X_h$ by

$$a_n(\varphi,\xi) = \int_\Omega \left(\frac{\alpha_{p_n,0}}{\Delta t}\varphi\,\xi + \boldsymbol{u}(t_n).\nabla_h\varphi\,\xi\right)\mathrm{d}x + \int_{\partial\Omega} \max(0,-\boldsymbol{u}(t_n).\boldsymbol{n})\,\varphi\,\xi\,\mathrm{d}s + \sum_{S\in\mathscr{S}_h^{(i)}} \int_S [\![\varphi]\!]\left(\frac{|\boldsymbol{u}(t_n).\boldsymbol{n}|}{2}[\![\xi]\!] - \boldsymbol{u}(t_n).\boldsymbol{n}\,\{\!\{\xi\}\!\}\right)\mathrm{d}s \tag{4.2a}$$

$$l_n(\xi) = -\sum_{i=1}^{p_n} \frac{\alpha_{p_n,i}}{\Delta t}\int_\Omega \phi_h^{(n-i)}\,\xi\mathrm{d}x \tag{4.2b}$$

When $n = 0$ we set $\phi_h^{(0)} = \pi_h(\phi_0)$ and when $n \geqslant 1$, the time approximation $\phi_h^{(n)}$ is defined by a p_n-order recurrence as the solution of the following linear problem:

$(P)_h$: *find* $\phi_h^{(n)} \in X_h$ *such that*

$$a_n\left(\phi_h^{(n)},\xi_h\right) = l_n(\xi_h),, \quad \forall \xi_h \in X_h$$

Finally, at each time step, the problem reduces to a linear system.

4.1.3 [New] Example: the Zalesak slotted disk

The Zalesak [1979, p. 350] slotted disk in rotation is a widely used benchmark for comparing the performances of interface capturing methods. The radius of the disk is 0.15 and its center is $(0.5, 0.75)$ The width of the slot is 0.05 and its length is 0.25. This slotted disk is rotated around the barycenter $(0.5, 0.5)$ of the computational domain $\Omega = [0,1]^2$ at velocity $\boldsymbol{u}(x_0,x_1) = (-(x_1-1/2)/2,\,(x_0-1/2)/2)$, such that the angular velocity $\operatorname{curl}\boldsymbol{u} = 1$ is constant. Note that the slotted disk returns to its initial position after a period $t_f = 4\pi$. The presentation of this section is mainly inspired from Ta et al. [2016], that used **Rheolef**. The following code implement the previous BDF scheme for this benchmark.

File 4.2: zalesak_dg.cc

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
#include "zalesak.h"
#include "bdf.icc"
int main(int argc, char**argv) {
  environment rheolef (argc, argv);
  geo omega (argv[1]);
  space Xh (omega, argv[2]);
  size_t n_max = (argc > 3) ?   atoi(argv[3]) : 1000;
  size_t strip = (argc > 4) ? string(argv[4]) == "true" : false;
  size_t p     = (argc > 5) ?   atoi(argv[5]) : min(Xh.degree()+1,bdf::pmax);
  Float tf = u::period(), delta_t = tf/n_max;
  trial phi (Xh); test xi (Xh);
  form m  = integrate (phi*xi),
       a0 = integrate (dot(u(),grad_h(phi))*xi)
          + integrate ("boundary", max(0, -dot(u(),normal()))*phi*xi)
          + integrate ("internal_sides",
                - dot(u(),normal())*jump(phi)*average(xi)
                + 0.5*abs(dot(u(),normal()))*jump(phi)*jump(xi));
  problem pb;
  branch event ("t","phi");
  vector<field> phi_h (p+1);
  phi_h[0] = phi_h[1] = lazy_interpolate (Xh, phi0());
  dout << event (0, phi_h[0]);
  for (size_t n = 1; n <= n_max; n++) {
    Float t = n*delta_t;
    if (n % 10 == 0) derr << "[" << n << "]";
    size_t pn = min(n,p);
    field rhs(Xh, 0);
    for (size_t i = 1; i <= pn; i++)
      rhs += (bdf::alpha[pn][i]/delta_t)*phi_h[i];
    field lh = integrate(rhs*xi)
             + integrate("boundary", max(0,-dot(u(),normal()))*phi_exact(t)*xi);
    if (pn <= p) {
      form an = a0 + (bdf::alpha[pn][0]/delta_t)*m;
      pb = problem (an);
    }
    pb.solve (lh, phi_h[0]);
    check_macro (phi_h[0].max_abs() < 100, "BDF failed -- HINT: decrease delta_t");
    if (!strip || n == n_max) dout << event (t, phi_h[0]);
    for (size_t i = min(p,pn+1); i >= 1; i--)
      phi_h[i] = phi_h[i-1];
  }
  derr << endl;
}
```

Comments

Numerical computations are performed on the time interval $[0, t_f]$ with a time step $\Delta t = t_f/n_{max}$ where n_{max} is the number of time steps. Since $d = 2$, a direct method is preferable (see section 1.1.10 and Fig. 1.5). Observe that $\boldsymbol{u}$ is here independent upon t. Then a_n, as given by (4.2a), is independent upon n when $n \geqslant p$, and a_n can be factored one time for all $n \geqslant p$. The file '`zalesak.h`' included here implements the `phi0` and `phi_exact` class-functions representing the exact solution $\phi(t)$ of level set function for the slotted disk, together with the `u` class-function for the velocity field. This file it is not listed here but is available in the **Rheolef** example directory. Recall that the BDF(p) scheme is unconditionally stable when $p \in \{1, 2\}$ and *almost* unconditionally stable when $p \in \{3, \ldots, 6\}$ (see e.g. Süli and Mayers, 2003, p. 349). Then, when an instability is detected, an issue for decreasing the time step is generated and the computation is stopped.

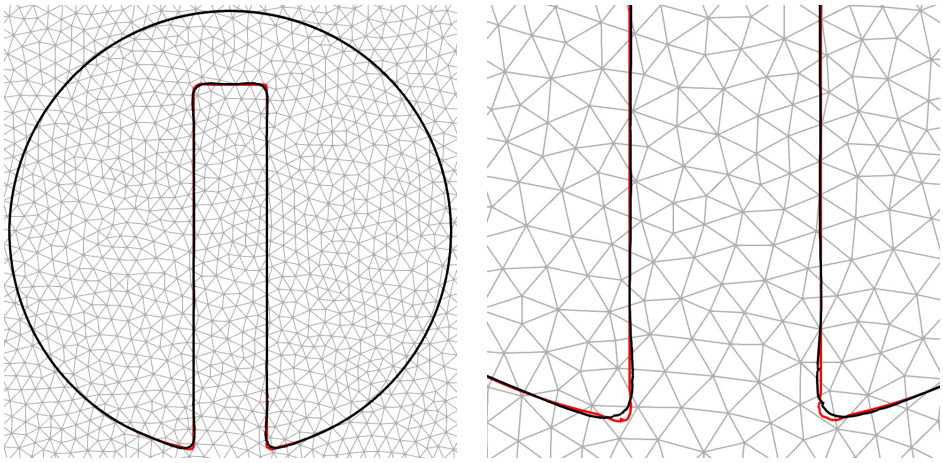


Figure 4.5: Zalesak slotted disk: superposition of the initial disk (black) and after a period (red). Polynomial degree $k = 2$, time step $\Delta t = 4\pi/6000$, implicit BDF(3) scheme with a fixed mesh $h = 1/100$. On the right: zoom.

How to run the program

Let us first generate a mesh for the $\Omega =]0,1[^2$ geometry with $h = 1/100$ and then run the code with a $k = 2$ discontinuous Galerkin method and a $BDF(3)$ scheme:

```
mkgeo_ugrid -t 50 > square.geo
make zalesak_dg
time mpirun -np 8 ./zalesak_dg square P2d > square-P2d.branch
```

The computation could take about ten minutes, depending upon your computer: there are 1000 time steps by default ; a prompt shows the step advancement until 1000. The `time` and `mpirun -np 8` prefixes are optional and you should omit them if you are running a sequential installation of **Rheolef**. Then, the visualization writes:

```
branch square-P2d.branch -iso 0 -bw
```

The slotted disk appears at his initial position. Then, click on the video play button, at the top of the window: the disk starts to rotate. Fig. 4.5 compares the initial disk position with the final one after one period when $h = 1/100$ and 6000 time steps. Observe the dramatic efficiency of the combination of the discontinuous Galerkin method and the high-order time scheme. The corresponding solution could be directly observed as a video from the following link (3 Mo):

http://www-ljk.imag.fr/membres/Pierre.Saramito/rheolef/zalesak-100-P2d-6000.mp4

Moreover, when using higher order polynomials and schemes, e.g. for $k \geqslant 3$, the changes are no more perceptible and the disk after one period fits the initial shape. The error between the two shapes could also be evaluated, and the convergence versus mesh and polynomial degree could be checked: see Ta et al. [2016] for more details.

4.1.4 [New] Example: the Leveque vortex-in-box

The LeVeque [1996, p. 653] vortex-in-box is also another widely used benchmark. This example tests the ability of the scheme to accurately resolve thin filaments on the scale of the mesh which

can occur in swirling and stretching flows. An initial shape is deformed by a non-constant velocity field given by:

$$\boldsymbol{u}(t,\boldsymbol{x}) = \begin{cases} \cos(\pi t/t_f) \begin{pmatrix} \sin^2(\pi x_0)\sin(2\pi x_1) \\ -\sin(2\pi x_0)\sin^2(\pi x_1) \end{pmatrix} & \text{when } d=2 \\ \cos(\pi t/t_f) \begin{pmatrix} 2\sin^2(\pi x_0)\sin(2\pi x_1)\sin(2\pi x_2) \\ -\sin(2\pi x_0)\sin^2(\pi x_1)\sin(2\pi x_2) \\ -\sin^2(\pi x_0)\sin(2\pi x_1)\sin(2\pi x_2) \end{pmatrix} & \text{when } d=3 \end{cases}$$

At the half-period $t = t_f/2$, the initial data is quite deformed and the flow slows down and reverses direction in such a way that the initial data should be recovered at time t_f. This gives a very useful test problem since we then know the true solution at time t_f even though the flow field has a quite complicated structure. Here we use $t_f = 8$ when $d = 2$ and $t_f = 3$ when $d = 3$. The initial shape is a circle of radius 0.15 placed at $(0.5, 0.75)$ when $d = 2$ and a sphere of radius 0.15 placed at $(0.35, 0.35, 0.35)$ when $d = 3$. The tridimensional flow field was first proposed by LeVeque [1996, p. 662] and the corresponding initial shape described by Enright et al. [2002, p. 112]. The computational domain is $\Omega =]0,1[^d$ and note that the flow field $\boldsymbol{u}$ vanishes on $\partial\Omega$.

File 4.3: `leveque_dg.cc`

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
#include "leveque.h"
#include "bdf.icc"
int main(int argc, char**argv) {
  environment rheolef (argc, argv);
  geo omega (argv[1]);
  space Xh (omega, argv[2]);
  size_t n_max = (argc > 3) ?   atoi(argv[3]) : 1000;
  size_t strip = (argc > 4) ? string(argv[4]) == "true" : false;
  size_t p     = (argc > 5) ?   atoi(argv[5]) : min(Xh.degree()+1,bdf::pmax);
  size_t d = omega.dimension();
  Float tf = u::period(), delta_t = tf/n_max;
  trial phi (Xh); test xi (Xh);
  branch event ("t","phi");
  vector<field> phi_h (p+1);
  phi_h[0] = phi_h[1] = lazy_interpolate (Xh, phi0(d));
  dout << event (0, phi_h[0]);
  for (size_t n = 1; n <= n_max; n++) {
    Float t = n*delta_t;
    if (n % 10 == 0) derr << "[" << n << "]";
    size_t pn = min(n,p);
    form an = integrate ((bdf::alpha[pn][0]/delta_t*phi + dot(u(d,t),grad_h(phi)))*xi)
            + integrate ("internal_sides",
                - dot(u(d,t),normal())*jump(phi)*average(xi)
                + 0.5*abs(dot(u(d,t),normal()))*jump(phi)*jump(xi));
    field rhs(Xh, 0);
    for (size_t i = 1; i <= pn; i++)
      rhs += (bdf::alpha[pn][i]/delta_t)*phi_h[i];
    field lh = integrate(rhs*xi);
    problem pb (an);
    pb.solve (lh, phi_h[0]);
    check_macro (phi_h[0].max_abs() < 100, "BDF failed -- HINT: decrease delta_t");
    if (!strip || n == n_max) dout << event (t, phi_h[0]);
    for (size_t i = min(p,pn+1); i >= 1; i--)
      phi_h[i] = phi_h[i-1];
  }
  derr << endl;
}
```

Comments

Numerical computations are performed on the time interval $[0, T]$ with a time step $\Delta t = T/n_{max}$ where n_{max} is the number of time steps. Observe that $\boldsymbol{u}$ is here time-dependent. Then a_n, as given by (4.2a), is also dependent upon n, and a_n can no more be factored one time for all, as it was done for the Zalesak case. The file '`leveque.h`' included here implements both the `u` and `phi0` class-functions for the flow field and the initial data ϕ_0 associated to the initial circular shape, respectively. This file it is not listed here but is available in the **Rheolef** example directory. Recall that $\boldsymbol{u}$ vanishes on $\partial\Omega$. Then, all boundary terms for a_n and ℓ_n, involved in (4.2a)-(4.2b), are here omitted.

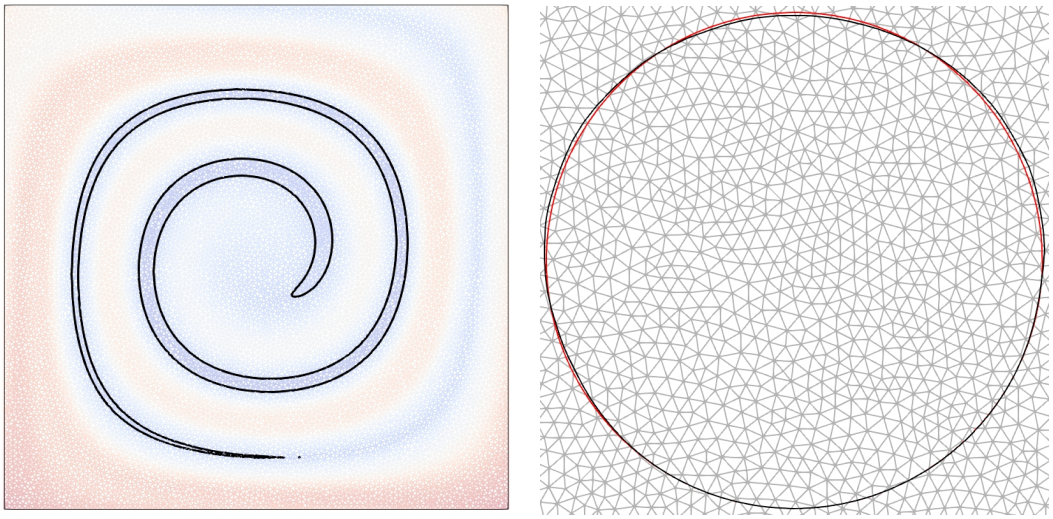


Figure 4.6: Leveque vortex-in-box: approximation with $h = 1/100$, $k = 2$, $p = 3$ and $\Delta t = 4\pi/6000$. (left) solution at half period : the deformation is maximal ; (right) after a full period : comparison with the initial shape, in red.

How to run the program

Compilation and run writes:

```
mkgeo_ugrid -t 100 > square.geo
make leveque_dg
mirun -np 8 ./leveque_dg square P2d 6000 > square-P2d.branch
```

The computation could take about two hours, depending upon your computer: a prompt shows the advancement for the 6000 time steps. The `mpirun -np 8` prefix is optional and you should omit it if you are running a sequential installation of **Rheolef**. Then, the visualization writes:

```
branch square-P2d.branch -iso 0 -bw
```

As for the Zalesak case, the solution is represented by an animation with `paraview`. The video could be directly observed from the link (7 Mo):

http://www-ljk.imag.fr/membres/Pierre.Saramito/rheolef/leveque-100-P2d-6000.mp4

Fig. 4.6.left represents the solution when $h = 1/100$ and 6000 time steps at the half-period $t = t_f/2$, when the deformation is maximal. Conversely, Fig. 4.6.right represents it when $t = t_f$, when the shape recovers its initial position. Observe the good correspondence between the final and the initial shapes, which is represented in red.

4.2 Nonlinear first-order problems

4.2.1 Abstract setting

The aim of this paragraph is to study the discontinuous Galerkin discretization of scalar nonlinear hyperbolic equations. This section presents the general framework and discretization tools while the next section illustrates the method for the Burgers equation.

Problem statement

A time-dependent nonlinear hyperbolic problem writes in general form [di Pietro and Ern, 2012, p. 99]:

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

$$\begin{aligned} \frac{\partial u}{\partial t} + \operatorname{div}\mathbf{f}(u) &= 0 \ \text{ in }]0,T[\times\Omega && \text{(4.3a)}\\ u(t=0) &= u_0 \ \text{ in } \Omega && \text{(4.3b)}\\ \mathbf{f}(u).\mathbf{n} &= \Phi(\mathbf{n};\, u, g) \text{ on }]0,T[\times\partial\Omega && \text{(4.3c)} \end{aligned}$$

where $T>0$, $\Omega\subset\mathbb{R}^d$, $d=1,2,3$ and the initial condition $\mathbf{u}_0$ being known. As usual, $\mathbf{n}$ denotes the outward unit normal on the boundary $\partial\Omega$. The function $\mathbf{f}:\mathbb{R}\longrightarrow\mathbb{R}^d$ is also known and supposed to be continuously differentiable. The initial data u_0, defined in Ω, and the boundary one, g, defined on $\partial\Omega$ are given. The function Φ, called the Godunov flux associated to $\mathbf{f}$, is defined, for all $\boldsymbol{\nu}\in\mathbb{R}^d$ and $a,b\in\mathbb{R}$, by

$$\Phi(\boldsymbol{\nu};\, a, b) = \begin{cases} \min\limits_{v\in[a,b]} \mathbf{f}(v).\boldsymbol{\nu} & \text{when } a\leqslant b \\ \max\limits_{v\in[b,a]} \mathbf{f}(v).\boldsymbol{\nu} & \text{otherwise} \end{cases} \qquad \text{(4.3d)}$$

Note that, with this general formalism, the linear transport problem considered in the previous section 4.1.1 corresponds to (see e.g. di Pietro and Ern, 2012, p. 104:

$$\begin{aligned} \boldsymbol{f}(u) &= \boldsymbol{a}u && \text{(4.4a)}\\ \Phi(\boldsymbol{n};\, u, v) &= \boldsymbol{a}.\boldsymbol{n}\frac{u+v}{2} + \frac{|\boldsymbol{a}.\boldsymbol{n}|}{2}(u-v) && \text{(4.4b)} \end{aligned}$$

Space discretization

In this section, we consider first the semi-discretization with respect to space while the problem remains continuous with respect to time. The semi-discrete problem writes in variational form [di Pietro and Ern, 2012, p. 100]:

$(P)_h$: *find* $u_h\in C^1([0,T],X_h)$ *such that*

$$\begin{aligned} \int_\Omega \frac{\partial u_h}{\partial t} v_h \,\mathrm{d}x - \int_\Omega \mathbf{f}(u_h).\nabla_h v_h \,\mathrm{d}x + \sum_{S\in\mathcal{S}_h^{(i)}} \int_S \Phi(\mathbf{n};\, u_h^-, u_h^+)[\![v_h]\!]\,\mathrm{d}s + \int_{\partial\Omega} \Phi(\mathbf{n};\, u_h, g) v_h \,\mathrm{d}s &= 0, \quad \forall v_h\in X_h \\ u_h(t=0) &= \pi_h(u_0) \end{aligned}$$

where π_h denotes the Lagrange interpolation operator on X_h and others notations has been introduced in the previous section.

For convenience, we introduce the discrete operator G_h, defined for all $u_h, v_h\in X_h$ by

$$\int_\Omega G_h(u_h) v_h \,\mathrm{d}x = -\int_\Omega \mathbf{f}(u_h).\nabla_h v_h \,\mathrm{d}x + \sum_{S\in\mathcal{S}_h^{(i)}} \int_S \Phi(\mathbf{n};\, u_h^-, u_h^+)[\![v_h]\!]\,\mathrm{d}s + \int_{\partial\Omega} \Phi(\mathbf{n};\, u_h, g) v_h \,\mathrm{d}s \qquad \text{(4.5)}$$

For a given $u_h \in X_h$, we also define the linear form g_h as

$$g(v_h) \quad = \quad \int_\Omega G_h(u_h) v_h \, \mathrm{d}x$$

As the matrix M, representing the L^2 scalar product in X_h, is block-diagonal, it can be easily inverted at the element level, and for a given $u_h \in X_h$, we have $G(u_h) = M^{-1} g_h$. Then, the problem writes equivalently as a set of coupled nonlinear ordinary differential equations.

$(P)_h$: *find* $u_h \in C^1([0,T], X_h)$ *such that*

$$\frac{\partial u_h}{\partial t} + G_h(u_h) \quad = \quad 0$$

Time discretization

Let $\Delta t > 0$ be the time step. The previous nonlinear ordinary differential equations are discretized by using a specific explicit Runge-Kutta with intermediates states [Shu and Osher, 1988, Gottlieb and Shu, 1998, Gottlieb et al., 2001]. This specific variant of the usual Runge-Kutta scheme, called *strong stability preserving*, is suitable for avoiding possible spurious oscillations of the approximate solution when the exact solution has a poor regularity. Let u_h^n denotes the approximation of u_h at time $t_n = n\Delta t$, $n \geqslant 0$. Then u_h^{n+1} is defined by recurrence:

$$\begin{aligned} u_h^{n,0} &= u_h^n \\ u_h^{n,i} &= \sum_{j=0}^{i-1} \alpha_{i,j} u_h^{n,j} - \Delta t \, \beta_{i,j} G_h\left(u_h^{n,j}\right), \quad 1 \leqslant i \leqslant p \\ u_h^{n+1} &= u_h^{n,p} \end{aligned}$$

where the coefficients satisfy $\alpha_{i,j} \geqslant 0$ and $\beta_{i,j} \geqslant 0$ for all $1 \leqslant i \leqslant p$ and $0 \leqslant j \leqslant i-1$, and $\sum_{j=0}^{i-1} \alpha_{i,j} = 1$ for all $1 \leqslant i \leqslant p$. Note that when $p = 1$ this scheme coincides with the usual explicit Euler scheme. For a general $p \geqslant 1$ order scheme, there are $p-1$ intermediate states $u_h^{n,i}$, $i = 1 \ldots p-1$. Computation of the coefficients $\alpha_{i,j}$ and $\beta_{i,j}$ can be founded in [Shu and Osher, 1988, Gottlieb and Shu, 1998, Gottlieb et al., 2001] and are grouped in file '`runge_kutta_ssp.icc`' of the examples directory for convenience.

4.2.2 Slope limiters

Slope limiters are required when the solution develops discontinuities: this is a very classical feature of most solutions of nonlinear hyperbolic equations. A preliminary version of the slope limiter proposed by Cockburn [1998, p. 208] is implemented in **Rheolef**: this preliminary version only supports the $d = 1$ dimension and $k = 1$ polynomial degree. Recall that the $k = 0$ case do not need any limiter. More general implementation will support the $d = 2, 3$ and $k \geqslant 2$ cases in the future. The details of the limiter implementation is presented in this section: the impatient reader, who is interested by applications, can jump to the next section, devoted to the Burgers equation.

Fig. 4.7 shows the $d+1$ neighbor elements K_i, $i = 0 \ldots d$ around an element d. Let $S_i = \partial K \cap \partial K_i$, $i = 0 \ldots d$ be the i-th side of K. We denote by $\boldsymbol{x}_K$, $\boldsymbol{x}_{K_i}$ ad $\boldsymbol{x}_{S_i}$ the barycenters of these elements and sides, $i = 0 \ldots d$. When $d = 2$, the barycenter $\boldsymbol{x}_{S_i}$ of the edge belongs to the interior of a triangle $(\boldsymbol{x}_K, \boldsymbol{x}_{K_i}, \boldsymbol{x}_{K_{J_{i,1}}})$ for exactly one of the two possible $J_{i,1} \neq i$ and $0 \leqslant J_{i,1} \leqslant d$. When $d = 3$, the barycenter $\boldsymbol{x}_{S_i}$ of the face belongs to the interior of a tetrahedron $(\boldsymbol{x}_K, \boldsymbol{x}_{K_i}, \boldsymbol{x}_{K_{J_{i,1}}}, \boldsymbol{x}_{K_{J_{i,2}}})$ for exactly one pair $(J_{i,1}, J_{i,2})$, up to a permutation, of the three possible pairs $J_{i,1}, J_{i,2} \neq i$ and $0 \leqslant J_{i,1}, J_{i,2} \leqslant d$. Let us denote $J_{i,0} = i$. Then, the vector $\overrightarrow{\boldsymbol{x}_K \boldsymbol{x}_{S_i}}$ decompose on the basis

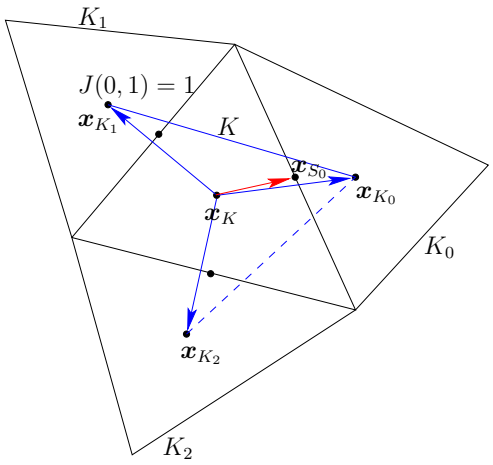


Figure 4.7: Limiter: the neighbors elements and the middle edge points.

$(\overrightarrow{\boldsymbol{x}_K\boldsymbol{x}_{K_{J_{i,k}}}})_{0\leqslant k\leqslant d-1}$ as

$$\overrightarrow{\boldsymbol{x}_K\boldsymbol{x}_{S_i}} = \sum_{k=0}^{d-1} \alpha_{i,k} \overrightarrow{\boldsymbol{x}_K\boldsymbol{x}_{K_{J_{i,k}}}} \tag{4.6}$$

where $\alpha_{i,k} \geqslant 0$, $k = 0 \dots d-1$. Let us consider now the patch ω_K composed of K and its d neighbors:

$$\omega_K = K \cup K_0 \cup \dots \cup K_d$$

For any affine function $\xi \in P_1(\omega_K)$ over this patch, let us denote

$$\begin{aligned} \delta_{K,i}(\xi) &= \sum_{k=0}^{d-1} \alpha_{i,k} \left(\xi(\boldsymbol{x}_{K_{J_{i,k}}}) - \xi(\boldsymbol{x}_K) \right), \quad i = 0 \dots d-1 \\ &= \xi(\boldsymbol{x}_{S_i}) - \xi(\boldsymbol{x}_K) \quad \text{from (4.6)} \end{aligned}$$

In other terms, $\delta_{K,i}(\xi)$ represents the departure of the value of ξ at $\boldsymbol{x}_{S_i}$ from its average $\xi(\boldsymbol{x}_K$ on the element K.

Let now $(\varphi_i)_{0\leqslant i\leqslant d-1}$ denote the Lagrangian basis in K associated to the set of nodes $(\boldsymbol{x}_{S_i})_{0\leqslant i\leqslant d-1}$:

$$\begin{gathered} \varphi_i(\boldsymbol{x}_{S_j}) = \delta_{i,j}, \quad 0 \leqslant i,j \leqslant d-1 \\ \sum_{i=0}^{d-1} \varphi_i(\boldsymbol{x}) = 1, \quad \forall \boldsymbol{x} \in K \end{gathered}$$

The affine function $\xi \in P_1(\omega_K)$ expresses on this basis as

$$\xi(\boldsymbol{x}) = \xi(\boldsymbol{x}_K) + \sum_{i=0}^{d-1} \delta_{K,i}(\xi)\, \varphi_i(\boldsymbol{x}), \quad \forall \boldsymbol{x} \in K$$

Let now $u_h \in \mathbb{P}_{1d}(\mathscr{T}_h)$. On any element $K \in \mathscr{T}_h$, let us introduce its average value:

$$\bar{u}_K = \frac{1}{\text{meas(K)}} \int_K u_h(\boldsymbol{x})\, \mathrm{d}x$$

and its departure from its average value:

$$\tilde{u}_K(\boldsymbol{x}) = u_{h|K}(\boldsymbol{x}) - \bar{u}_K, \quad \forall \boldsymbol{x} \in K$$

Note that $u_h \notin P_1(\omega_K)$. Let us extends $\delta_{K,i}$ to u_h as

$$\delta_{K,i}(u_h) \quad = \quad \sum_{k=0}^{d-1} \alpha_{i,k} \left(\bar{u}_{K_{J_{i,k}}} - \bar{u}_K \right), \quad i = 0 \dots d-1$$

Since $u_h \notin P_1(\omega_K)$, we have $\tilde{u}_K(\boldsymbol{x}_{K_{J_{i,k}}}) \neq \delta_{K,i}(u_h)$ in general. The idea is then to capture oscillations by controlling the departure of the values $\tilde{u}_K(\boldsymbol{x}_{K_{J_{i,k}}})$ from the values $\delta_{K,i}(u_h)$. Thus, associate to $u_h \in \mathbb{P}_{1d}(\mathscr{T}_h)$ the quantities

$$\Delta_{K,i}(u_h) \quad = \quad \mathrm{minmod}_{\mathrm{TVB}} \left(\tilde{u}_K(\boldsymbol{x}_{K_{J_{i,k}}}),\ \theta \delta_{K,i}(u_h) \right)$$

for all $i = 0 \dots d-1$ and where $\theta \geqslant 1$ is a parameter of the limiter and

$$\mathrm{minmod}_{\mathrm{TVB}}(a,b) \quad = \quad \begin{cases} a & \text{when } |a| \leqslant Mh^2 \\ \mathrm{minmod}(a,b) & \text{otherwise} \end{cases}$$

where $M > 0$ is a tunable parameter which can be evaluated from the curvature of the initial datum at its extrema by setting

$$M = \sup_{\boldsymbol{x} \in \Omega, \nabla u_0(\boldsymbol{x}) = 0} |\nabla \otimes \nabla u_0| \tag{4.7}$$

Introduced by Shu [1987], the basic idea is to deactivate the limiter when space derivatives are of order h^2. This improves the limiter behavior near smooth local extrema. The minmod function is defined by

$$\mathrm{minmod}(a,b) \quad = \quad \begin{cases} \mathrm{sgn}(a) \min(|a|,|b|) & \text{when } \mathrm{sgn}(a) = \mathrm{sgn}(b) \\ 0 & \text{otherwise} \end{cases}$$

Then, for all $i = 0 \dots d-1$ we define

$$\begin{aligned} r_K(u_h) \quad &= \quad \frac{\displaystyle\sum_{j=0}^{d-1} \max(0, -\Delta_{K,j}(u_h))}{\displaystyle\sum_{j=0}^{d-1} \max(0, \Delta_{K,j}(u_h))} \quad \geqslant 0 \\ \hat{\Delta}_{K,i}(u_h) \quad &= \quad \min(1, r_K(u_h))\ \max(0, \Delta_{K,i}(u_h)) \\ &\quad - \min(1, 1/r_K(u_h))\ \max(0, -\Delta_{K,i}(u_h)), \quad i = 0 \dots d-1 \text{ when } r_K(u_h) \neq 0 \end{aligned}$$

Finally, the limited function $\Lambda_h(u_h)$ is defined element by element for all element $K \in \mathscr{T}_h$ for all $\boldsymbol{x} \in K$ by

$$\Lambda_h(u_h)_{|K}(\boldsymbol{x}) \quad = \quad \begin{cases} \bar{u}_K + \displaystyle\sum_{i=1}^{d-1} \Delta_{K,i}(u_h)\ \varphi_i(\boldsymbol{x}) & \text{when } r_K(u_h) = 0 \\ \bar{u}_K + \displaystyle\sum_{i=1}^{d-1} \hat{\Delta}_{K,i}(u_h)\ \varphi_i(\boldsymbol{x}) & \text{otherwise} \end{cases}$$

Note that there are two types of computations involved in the limiter: one part is independent of u_h and depends only upon the mesh: $J_{i,k}$ and $\alpha_{i,k}$ on each element. It can be computed one time for all. The other part depends upon the values of u_h.

Note that the limiter preserves the average value of u_h on each element K and also the functions that are globally affine on the patch ω_K. Also we have, inside each element K and for all side index $i = 0 \dots d-1$:

$$\left| \Lambda_h(u_h)_{|K}(\boldsymbol{x}_{S_i}) - \bar{u}_K \right| \ \leqslant\ \max\left(|\Delta_{K,i}(u_h)|,\ |\hat{\Delta}_{K,i}(u_h)| \right) \ \leqslant\ |\Delta_{K,i}(u_h)| \ \leqslant\ \left| u_{h|K}(\boldsymbol{x}_{S_i}) - \bar{u}_K \right|$$

It means that, inside each element, the gradient of the P_1 limited function is no larger than that of the original one.

The limiter on an element close to the boundary should takes into account the inflow condition, see Cockburn et al. [1990].

4.2.3 Example: the Burgers equation

As an illustration, let us consider now the test with the one-dimensional ($d = 1$) Burgers equation for a propagating slant step (see e.g. Carey and Jianng, 1988, p. 87) in $\Omega =]0, 1[$. We have $\mathbf{f}(u) = u^2/2$, for all $u \in \mathbb{R}$. In that case, the Godunov flux (4.3d), introduced page 157, can be computed explicitly for any $\boldsymbol{\nu} = (\nu_0) \in \mathbb{R}^d$ and $a, b \in \mathbb{R}$:

$$\Phi(\boldsymbol{\nu}; a, b) = \begin{cases} \nu_0 \min\left(a^2, b^2\right)/2 & \text{when } \nu_0 \geqslant 0 \text{ and } a \leqslant b \text{ or } \nu_0 \leqslant 0 \text{ and } a \geqslant b \\ \nu_0 \max\left(a^2, b^2\right)/2 & \text{otherwise} \end{cases}$$

File 4.4: `burgers.icc`

```
point f (const Float& u) { return point (sqr(u)/2); }
```

File 4.5: `burgers_flux_godunov.icc`

```
Float phi (const point& nu, Float a, Float b) {
  if ((nu[0] >= 0 && a <= b) || (nu[0] <= 0 && a >= b))
    return nu[0]*min(sqr(a),sqr(b))/2;
  else
    return nu[0]*max(sqr(a),sqr(b))/2;
}
```

Computing an exact solution

An exact solution is useful for testing numerical methods. The computation of such an exact solution for the one dimensional Burgers equation is described by Harten et al. [1987]. The authors consider first the problem with a periodic boundary condition:

(P): find $u :]0, T[\times]-1, 1[\longmapsto \mathbb{R}$ such that

$$\begin{aligned} \frac{\partial u}{\partial t} + \frac{\partial}{\partial x}\left(\frac{u^2}{2}\right) &= 0 \quad \text{in }]0, T[\times]-1, 1[\\ u(t{=}0, x) &= \alpha + \beta \sin(\pi x + \gamma), \quad \text{a.e. } x \in]-1, 1[\\ u(t, x{=}-1) &= u(t, x{=}1) \quad \text{a.e. } t \in]0, T[\end{aligned}$$

where α, β and γ are real parameters. Let us denote w the solution of the problem when $\beta = 1$ and $\alpha = \gamma = 0$; i.e. with the initial condition $w(t{=}0, x) = \sin(\pi x)$, a.e. $x \in]-1, 1[$. For any $x \in [0, 1[$ and $t > 0$, the solution $\bar{w} = w(t, x)$ satisfies the characteristic relation

$$\bar{w} = \sin(\pi(x - \bar{w}t))$$

This nonlinear relation can be solved by a Newton algorithm. Then, for $x \in]-1, 0[$, the solution is completed by symmetry: $w(t, x) = -w(t, -x)$. Finally, the general solution for any α, β and $\gamma = 0$ writes $u(t, x) = \alpha + w(\beta t,\, x - \alpha t + \gamma)$. File '`harten.h`' implements this approach.

File 4.6: harten.h

```
#include "harten0.h"
struct harten {
  Float operator() (const point& x) const {
    Float x0 = x[0]-a*t+c;
    Float shift = -2*floor((x0+1)/2);
    Float xs = x0 + shift;
    check_macro (xs >= -1 && xs <= 1, "invalid xs="<<xs);
    return a + b*h0 (point(xs));
  }
  harten (Float t1=0, Float a1=1, Float b1=0.5, Float c1=0):
    h0(b1*t1), t(t1), a(a1), b(b1), c(c1) {}
  Float M() const { Float pi = acos(-1.0); return sqr(pi)*b; }
  Float min() const { return a-b; }
  Float max() const { return a+b; }
protected:
  harten0 h0;
  Float t, a, b, c;
};
using u_init = harten;
using g = harten;
```

File 4.7: harten_show.cc

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
#include "harten.h"
int main(int argc, char**argv) {
  environment rheolef (argc, argv);
  geo omega (argv[1]);
  space Xh (omega, argv[2]);
  size_t nmax = (argc > 3) ? atoi(argv[3]) : 1000;
  Float tf    = (argc > 4) ? atof(argv[4]) : 2.5;
  Float a     = (argc > 5) ? atof(argv[5]) : 1;
  Float b     = (argc > 6) ? atof(argv[6]) : 0.5;
  Float c     = (argc > 7) ? atof(argv[7]) : 0;
  branch even("t","u");
  for (size_t n = 0; n <= nmax; ++n) {
    Float t = n*tf/nmax;
    field pi_h_u = lazy_interpolate (Xh, harten(t,a,b,c));
    dout << even(t,pi_h_u);
  }
}
```

The included file 'harten0.h' is not shown here but is available in the example directory.

Comments

Note that the constant M, used by the limiter in (4.7), can be explicitly computed for this solution: $M = \beta\pi^2$.

The animation of this exact solution is performed by the following commands:

```
make harten_show
mkgeo_grid -e 2000 -a -1 -b 1 > line2.geo
./harten_show line2 P1 1000 2.5 > line2-exact.branch
branch line2-exact -gnuplot
```

Fig. 4.8 shows the solution u for $\alpha = 1, \beta = 1/2$ and $\gamma = 0$. It is regular until $t = 2/\pi$ (Fig. 4.8.c) and then develops a chock for $t > 2/\pi$ (Fig. 4.8.d). After its apparition, this chock interacts with the expansion wave in $]-1, 1[$: this brings about a fast decay of the solution (Figs. 4.8.e and f). Fig. 4.8 plots also a numerical solution: its computation is the aim of the next section.

Numerical resolution

When replacing the periodic boundary condition with a inflow one, associated with the boundary data g, we choose g to be the value of the exact solution of the problem with periodic boundary conditions: $g(t,x) = \alpha + w(\beta t,\, x - \alpha t)$ for $x \in \{-1, 1\}$.

File 4.8: `burgers_dg.cc`

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
#include "harten.h"
#include "burgers.icc"
#include "burgers_flux_godunov.icc"
#include "runge_kutta_ssp.icc"
int main(int argc, char**argv) {
  environment rheolef (argc, argv);
  geo omega (argv[1]);
  space Xh (omega, argv[2]);
  Float cfl = 1;
  limiter_option lopt;
  size_t nmax = (argc > 3) ? atoi(argv[3]) : numeric_limits<size_t>::max();
  Float  tf   = (argc > 4) ? atof(argv[4]) : 2.5;
  size_t p    = (argc > 5) ? atoi(argv[5]) : ssp::pmax;
  lopt.M      = (argc > 6) ? atoi(argv[6]) : u_init().M();
  if (nmax == numeric_limits<size_t>::max()) {
      nmax = (size_t)floor(1+tf/(cfl*omega.hmin()));
  }
  Float delta_t = tf/nmax;
  integrate_option iopt;
  iopt.invert = true;
  trial u (Xh); test v (Xh);
  form inv_m = integrate (u*v, iopt);
  vector<field> uh(p+1, field(Xh,0));
  uh[0] = lazy_interpolate (Xh, u_init());
  branch even("t","u");
  dout << catchmark("delta_t") << delta_t << endl
       << even(0,uh[0]);
  for (size_t n = 1; n <= nmax; ++n) {
    for (size_t i = 1; i <= p; ++i) {
      uh[i] = 0;
      for (size_t j = 0; j < i; ++j) {
        field lh =
         - integrate (dot(compose(f,uh[j]),grad_h(v)))
         + integrate ("internal_sides",
             compose (phi, normal(), inner(uh[j]), outer(uh[j]))*jump(v))
         + integrate ("boundary",
             compose (phi, normal(), uh[j], g(n*delta_t))*v);
        uh[i] += ssp::alpha[p][i][j]*uh[j] - delta_t*ssp::beta[p][i][j]*(inv_m*lh);
      }
      uh[i] = limiter(uh[i], g(n*delta_t)(point(-1)), lopt);
    }
    uh[0] = uh[p];
    dout << even(n*delta_t,uh[0]);
  }
}
```

Comments

The Runge-Kutta time discretization combined with the discontinuous Galerkin space discretization is implemented for this test case.

The P_0 approximation is performed by the following commands:

```
make burgers_dg
mkgeo_grid -e 200 -a -1 -b 1 > line2-200.geo
./burgers_dg line2-200.geo P0 1000 2.5 > line2-200-P0.branch
```

```
branch line2-200-P0.branch -gnuplot
```

The two last commands compute the P_0 approximation of the solution, as shown on Fig. 4.8. Observe the robust behavior of the solution at the vicinity of the chock. By replacing `P0` by `P1d` in the previous commands, we obtain the P_1-discontinuous approximation.

```
./burgers_dg line2-200.geo P1d 1000 2.5 > line2-200-P1d.branch
branch line2-200-P1d.branch -gnuplot
```

Fig. 4.9 plots the error vs h for $k = 0$ and $k = 1$. Fig. 4.9.a plots in a time interval $[0, T]$ with $T = 1/\pi$, before the chock that occurs at $t = 2/\pi$. In that interval, the solution is regular and the error approximation behaves as $\mathcal{O}(h^{k+1})$. The time interval has been chosen sufficiently small for the error to depend only upon h. Fig. 4.9.b plots in a larger time interval $[0, T]$ with $T = 5/2$, that includes the chock. Observe that the error behaves as $\mathcal{O}(h)$ for both $k = 0$ and 1. This is optimal when $k = 0$ but not when $k = 1$. This is due to the loss of regularity of the exact solution that presents a discontinuity; A similar observation can be found in Zhong and Shu [2013], table 4.1.

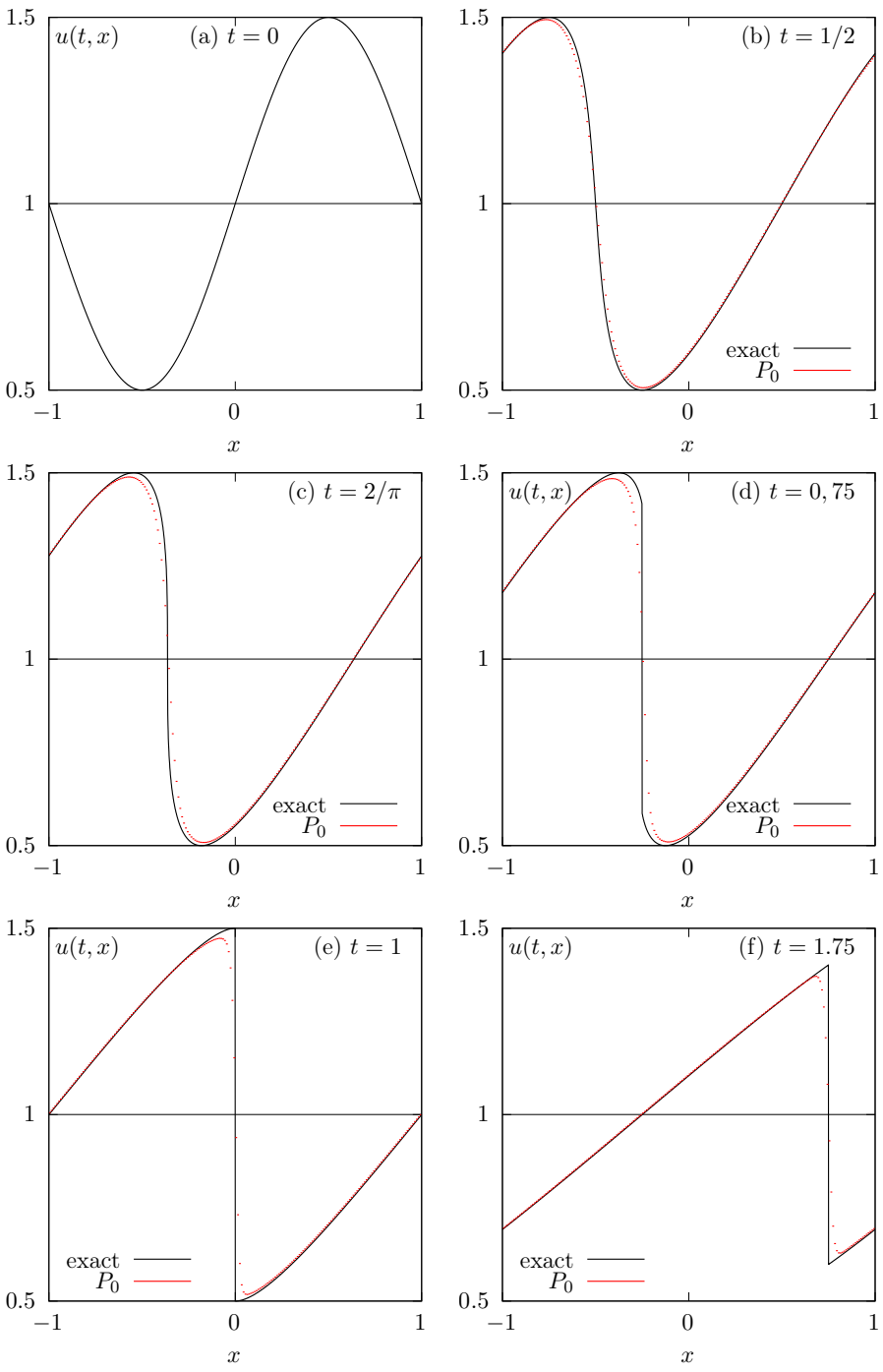


Figure 4.8: Harten's exact solution of the Burgers equation ($\alpha = 1, \beta = 1/2, \gamma = 0$). Comparison with the P_0 approximation ($h = 1/100$, RK-SSP(3)).

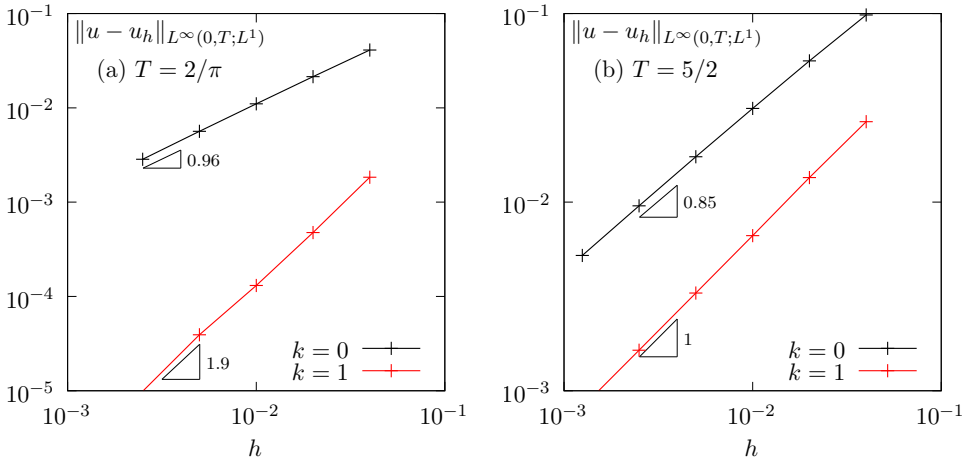


Figure 4.9: Burgers equation: error between the P_0 approximation and the exact solution of the Harten's problem ($\alpha=1, \beta=1/2, \gamma=0$): (a) before chock, with $T=1/\pi$; (b) after chock, with $T=5/2$.

4.3 Scalar second-order problems

4.3.1 The Poisson problem with Dirichlet boundary conditions

The Poisson problem with non-homogeneous Dirichlet boundary conditions has been already introduced in volume 1, section 1.1.12, page 24:

(P): *find* u, *defined in* Ω, *such that*

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

where f and g are given.

The *discontinuous* finite element space is defined by:

$$X_h = \{v_h \in L^2(\Omega); v_{h|K} \in P_k,\ \forall K \in \mathcal{T}_h\}$$

where $k \geqslant 1$ is the polynomial degree. As elements of X_h do not belongs to $H^1(\Omega)$, due to discontinuities at inter-elements, we introduce the broken Sobolev space:

$$H^1(\mathcal{T}_h) = \{v \in L^2(\Omega);\ v_{|K} \in H^1(K),\ \forall K \in \mathcal{T}_h\}$$

such that $X_h \subset H^1(\mathcal{T}_h)$. We introduce the folowing bilinear form $a_h(.,.)$ and linear for $l_h(.)$, defined for all $u, v \in H^1(\mathcal{T}_h)$ by (see e.g. di Pietro and Ern, 2012, p. 125 and 127, eqn. (4.12)):

$$a_h(u,v) = \int_\Omega \nabla_h u.\nabla_h v \,\mathrm{d}x + \sum_{S\in\mathscr{S}_h} \int_S \left(\kappa_s \left[\!\left[u\right]\!\right]\left[\!\left[v\right]\!\right] - \left\{\!\left\{\nabla_h u.\mathbf{n}\right\}\!\right\}\left[\!\left[v\right]\!\right] - \left[\!\left[u\right]\!\right]\left\{\!\left\{\nabla_h v.\mathbf{n}\right\}\!\right\}\right) \mathrm{d}s \quad (4.8)$$

$$l_h(v) = \int_\Omega f\, u \,\mathrm{d}x + \int_{\partial\Omega} \left(\kappa_s\, g\, v - g\, \nabla_h v.\mathbf{n}\right) \mathrm{d}s \quad (4.9)$$

The last term involves a sum over $\mathscr{S}_h$, the set of *all sides* of the mesh $\mathcal{T}_h$, i.e. the internal sides and the boundary sides. By convenience, the definition of the jump and average are extended to all boundary sides as $\left[\!\left[u\right]\!\right] = \left\{\!\left\{u\right\}\!\right\} = u$. Note that, as for the previous transport problem, the Dirichlet boundary condition $u = g$ is weakly imposed on $\partial\Omega$ via the integrals on the boundary. Finally, $\kappa_s > 0$ is a stabilization parameter on a side S. The stabilization term associated to κ_s is present in order to achieve coercivity: it penalize interface and boundary jumps. A common choice is $\kappa_s = \beta\, h_s^{-1}$ where $\beta > 0$ is a constant and h_s is a local length scale associated to the current side S. One drawnback to this choice is that it requires the end user to specify the numerical constant β. From one hand, if the value of this parameter is not sufficiently large, the form $a_h(.,.)$ is not coercive and the approximate solution develops instabilities an do not converge [Epshteyn and Rivière, 2007]. From other hand, if the value of this parameter is too large, its affect the overall efficiency of the iterative solver of the linear system: the spectral condition number of the matrix associated to $a_h(.,.)$ grows linearly with this paramater [Castillo, 2002]. An explicit choice of penalty parameter is proposed [Shahbazi, 2005]: $\kappa_s = \beta\, \varpi_s$ where $\beta = (k+1)(k+d)/d$ and

$$\varpi_s = \begin{cases} \dfrac{\mathrm{meas}(\partial K)}{\mathrm{meas}(K)} & \text{when } S = K \cap \partial\Omega \text{ is a boundary side} \\ \max\left(\dfrac{\mathrm{meas}(\partial K_0)}{\mathrm{meas}(K_0)}, \dfrac{\mathrm{meas}(\partial K_1)}{\mathrm{meas}(K_1)}\right) & \text{when } S = K_0 \cap K_1 \text{ is an internal side} \end{cases}$$

Note that ϖ_s scales as h_s^{-1}. Now, the computation of the penalty parameter is fully automatic and the convergence of the method is always guaranted to converge. Moreover, this choice has been founded to be sharp and it preserves the optimal efficiency of the iterative solvers. Finally, the discrete variational formulation writes:

$(FV)_h$: *find* $u_h \in X_h$ *such that*

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

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

File 4.9: `dirichlet_dg.cc`

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
#include "cosinusprod_laplace.h"
int main(int argc, char**argv) {
  environment rheolef (argc, argv);
  geo omega (argv[1]);
  space Xh (omega, argv[2]);
  size_t d = omega.dimension();
  size_t k = Xh.degree();
  Float beta = (k+1)*(k+d)/Float(d);
  trial u (Xh); test v (Xh);
  form  a = integrate (dot(grad_h(u),grad_h(v)))
          + integrate ("sides", beta*penalty()*jump(u)*jump(v)
                - jump(u)*average(dot(grad_h(v),normal()))
                - jump(v)*average(dot(grad_h(u),normal())));
  field lh = integrate (f(d)*v)
           + integrate ("boundary", beta*penalty()*g(d)*v
                - g(d)*dot(grad_h(v),normal()));
  a.uu().set_definite_positive(true);
  field uh(Xh);
  problem p (a);
  p.solve (lh, uh);
  dout << uh;
}
```

Comments

The `penalty()` pseudo-function implements the computation of ϖ_s in **Rheolef**. The right-hand side f and g are given by (1.4), volume 1, page 25. In that case, the exact solution is known. Running the one-dimensional case writes:

```
make dirichlet_dg
mkgeo_grid -e 10 > line.geo
./dirichlet_dg line P1d | field -
./dirichlet_dg line P2d | field -
```

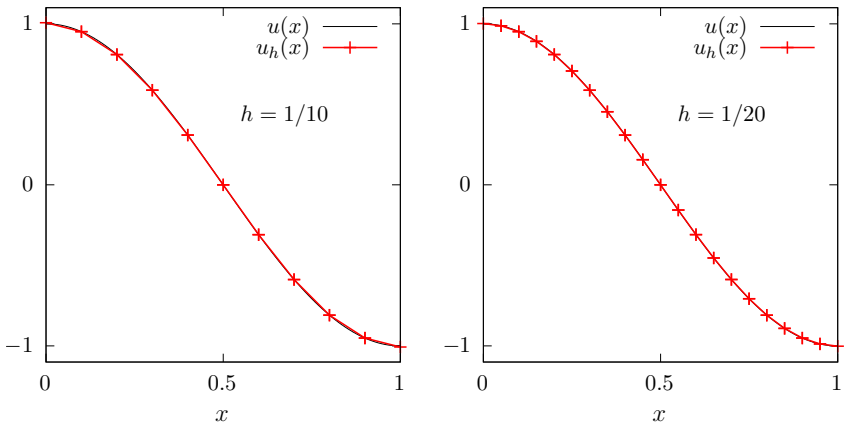


Figure 4.10: The discontinuous Galerkin method for the Poisson problem when $k = 1$ and $d = 1$.

Fig. 4.10 plots the one-dimensional solution when $k = 1$ for two meshes. Observe that the jumps at inter-element nodes decreases very fast with mesh refinement and are no more perceptible on the

plots. Recall that the Dirichlet boundary conditions at $x = 0$ and $x = 1$ is only weakly imposed: the corresponding jump at the boundary is also not perceptible.

The two-dimensional case writes:

```
mkgeo_grid -t 10 > square.geo
./dirichlet_dg square P1d | field -elevation -
./dirichlet_dg square P2d | field -elevation -
```

and the three-dimensional one

```
mkgeo_grid -T 10 > cube.geo
./dirichlet_dg cube P1d | field -
./dirichlet_dg cube P2d | field -
```

Error analysis

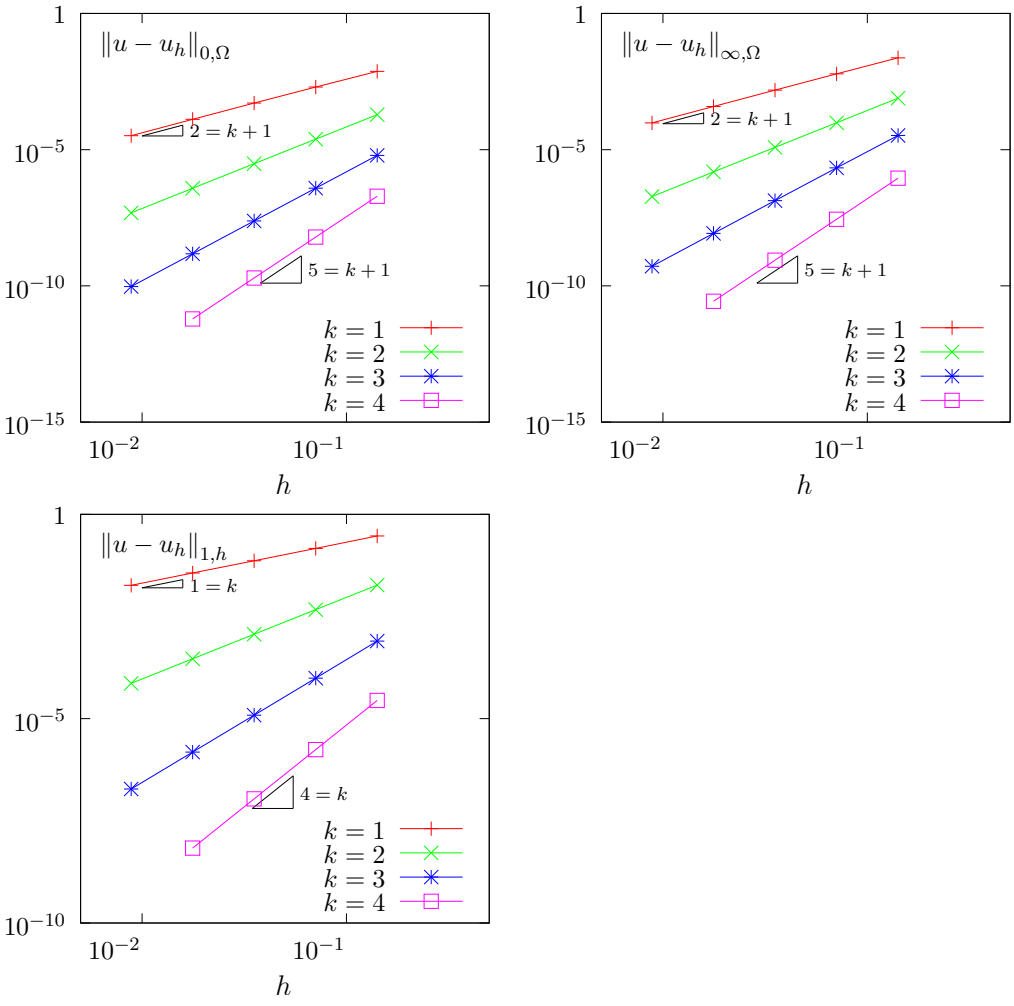


Figure 4.11: The discontinuous Galerkin method for the Poisson problem: convergence when $d = 2$.

The space $H^1(\mathcal{T}_h)$ is equiped with the norm $\|.\|_{1,h}$, defined for all $v \in H^1(\mathcal{T}_h$ by di Pietro and Ern

[2012, p. 128]:

$$\|v\|_{1,h}^2 = \|\nabla_h v\|_{0,\Omega}^2 + \sum_{S\in\mathscr{S}_h} \int_S h_s^{-1} \left[\!\left[v\right]\!\right]^2 \,\mathrm{d}s$$

The code `cosinusprod_error_dg.cc` compute the error in these norms. This code it is not listed here but is available in the **Rheolef** example directory. The computation of the error writes:

```
make cosinusprod_error_dg
./dirichlet_dg square P1d | cosinusprod_error_dg
```

Fig. 4.11 plots the error $u - u_h$ in L^2, L^∞ and the $\|.\|_{1,h}$ norms. The L^2 and L^∞ error norms behave as $\mathcal{O}\left(h^{k+1}\right)$ for all $k \geqslant 0$, while the $\|.\|_{1,h}$ one behaves as $\mathcal{O}\left(h^k\right)$, which is optimal.

4.3.2 The Helmholtz problem with Neumann boundary conditions

The Poisson problem with non-homogeneous Neumann boundary conditions has been already introduced in volume 1, section 1.2, page 32:

(P): *find u, defined in Ω, such that*

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

where f and g are given. We introduce the folowing bilinear form $a_h(.,.)$ and linear for $l_h(.)$, defined for all $u, v \in H^1(\mathcal{T}_h)$ by (see e.g. di Pietro and Ern, 2012, p. 127, eqn. (4.16)):

$$\begin{aligned} a_h(u,v) &= \int_\Omega (u\,v + \nabla_h u.\nabla_h v)\,\mathrm{d}x && (4.10)\\ &\quad + \sum_{S\in\mathscr{S}_h^{(i)}} \int_S (\beta\varpi_s \left[\!\left[u\right]\!\right]\left[\!\left[v\right]\!\right] - \left\{\!\left\{\nabla_h u.\mathbf{n}\right\}\!\right\}\left[\!\left[v\right]\!\right] - \left[\!\left[u\right]\!\right]\left\{\!\left\{\nabla_h v.\mathbf{n}\right\}\!\right\})\,\mathrm{d}s && (4.11)\\ l_h(v) &= \int_\Omega f\,u\,\mathrm{d}x + \int_{\partial\Omega} g\,v\,\mathrm{d}s && (4.12) \end{aligned}$$

Let us comment the changes between these forms and those used for the Poisson problem with Dirichlet boundary conditions. The Poisson operator $-\Delta$ has been replaced by the Helmholtz one $I - \Delta$ in order to have an unique solution. Remark also that the sum is performed in (4.8) for all internal sides in $\mathscr{S}_h^{(i)}$, while, in (4.8), for Dirichlet boundary conditions, it was for all sides in $\mathscr{S}_h$, i.e. for both boundary and internal sides. Also, the right-hand-side linear form $l_h(.)$. do no more involves any sum over sides.

Finally, the discrete variational formulation writes:

$(FV)_h$: *find $u_h \in X_h$ such that*

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

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

File 4.10: neumann_dg.cc

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
#include "sinusprod_helmholtz.h"
int main(int argc, char**argv) {
  environment rheolef (argc, argv);
  geo omega (argv[1]);
  space Xh (omega, argv[2]);
  size_t d = omega.dimension();
  size_t k = Xh.degree();
  Float beta = (k+1)*(k+d)/Float(d);
  trial u (Xh); test v (Xh);
  form  a = integrate (u*v + dot(grad_h(u),grad_h(v)))
          + integrate ("internal_sides",
                  beta*penalty()*jump(u)*jump(v)
                - jump(u)*average(dot(grad_h(v),normal()))
                - jump(v)*average(dot(grad_h(u),normal())));
  field lh = integrate (f(d)*v) + integrate ("boundary", g(d)*v);
  field uh(Xh);
  problem p (a);
  p.solve (lh, uh);
  dout << uh;
}
```

Comments

The right-hand side f and g are given by (1.5), volume 1, page 25. In that case, the exact solution is known. Running the program is obtained from the non-homogeneous Dirichlet case, by replacing `dirichlet_dg` by `neumann_dg`.

4.3.3 [New] Heterogeneous diffusion

The problem with with non-constant diffusion coefficients has already been introduced in section 4.3.3, page 171 when studying continuous approximations. It writes

(P): *find u defined in Ω such that:*

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

where f is a given source term. We consider here the important special case when the diffusion coefficient η is piecewise constant:

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

where $(\Omega_{\mathrm{west}}, \Omega_{\mathrm{east}})$ is a partition of Ω in two parts (see Fig. 1.10, page 38). This is the so-called **transmission** problem: the solution and the flux are continuous on the interface $\Gamma_0 = \partial\Omega_{\mathrm{east}} \cap \partial\Omega_{\mathrm{west}}$ between the two domains where the problem reduce to a constant diffusion one:

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

Following Dryja [2003], we use the discontinuous Galerkin with diffusion-dependent weights to formulate the consistency and symmetry terms in the discrete bilinear form and we penalize interface and boundary jumps using a diffusion-dependent parameter scaling as the harmonic mean of the diffusion coeffcient. As pointed out by di Pietro and Ern [2012, p. 150], this a penalty strategy is particularly important in heterogeneous diffusion-convection problems where the diffusion coeffcient takes locally small values leading to advection-dominated regimes. In this context, the exact solution exhibits sharp inner layers which, in practice, are not resolved by the underlying meshes, so that excessive penalty at such layers triggers spurious oscillations. Using the harmonic mean of the diffusion coeffcient to penalize jumps turns out to tune automatically the amount of penalty and thereby avoid such oscillations.

The *weighted average* is defined for any function v and on any side $S \in \mathscr{S}_h^{(i)}$ by

$$\{\!\{v\}\!\}_\omega = \omega_{K_0,S}\, v_{|K_0} + \omega_{K_1,S}\, v_{|K_1}$$

where $K_0, K_1 \in \mathscr{T}_h$ such that $S = K_0 \cap K_1$ and where the two weights $\omega_{K_0,S}$, $\omega_{K_1,S}$ express

$$\omega_{K_0,S} = \frac{\eta_{|K_0}}{\eta_{|K_0} + \eta_{|K_1}} \quad \text{and} \quad \omega_{K_1,S} = \frac{\eta_{|K_1}}{\eta_{|K_0} + \eta_{|K_1}}$$

Note that $\omega_{K_0,S} + \omega_{K_1,S} = 1$. On a boundary side $S \subset \partial\Omega$,, we set $\{\!\{v\}\!\}_\omega = v$.

The bilinear form $a_h(.,.)$ and linear for $l_h(.)$ are obtained by simply using the weighted average $\{\!\{.\}\!\}_\omega$ instead of $\{\!\{.\}\!\}$ and taking into accound upon te mixed Dirichlet and Neumann boundary condition. For all $u, v \in H^1(\mathcal{T}_h)$, they are given by (see e.g. di Pietro and Ern, 2012, p. 155, eqn. (4.64)):

$$a_h(u,v) = \int_\Omega \eta \nabla_h u.\nabla_h v \,\mathrm{d}x + \sum_{S \in \mathscr{S}_h^{(i)} \cup \Gamma_d} \int_S \left(\eta_s \kappa_s \,[\![u]\!]\,[\![v]\!] - \{\!\{\eta \nabla_h u.\mathbf{n}\}\!\}_w \,[\![v]\!] - \{\!\{\eta \nabla_h v.\mathbf{n}\}\!\}_w \,[\![u]\!] \right) \mathrm{d}s$$

$$l_h(v) = \int_\Omega f\, u \,\mathrm{d}x + \int_{\Gamma_d} \left(\kappa_s\, g\, v - g\, \nabla_h v.\mathbf{n} \right) \mathrm{d}s$$

where $\Gamma_d = \Gamma_{\text{left}} \cup \Gamma_{\text{right}}$ denote, for convenience, the boundary domain corresponding to the non-homogeneous Dirichlet boundary condition, $\eta_s = 2/(1/\eta_{|K_0} + 1/\eta_{|K_1})$ is the geometric mean of the diffusion coefficient on an internal side and $\kappa_s = \beta\, \varpi_s$ is the stabilization, as introduced in the previous section 4.3.1. Finally, the discrete variational formulation writes:

$(FV)_h$: *find* $u_h \in X_h$ *such that*

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

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

File 4.11: transmission_dg.cc

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
int main(int argc, char**argv) {
  environment rheolef (argc, argv);
  geo omega (argv[1]);
  string approx = (argc > 2) ? argv[2] : "P1d";
  Float epsilon = (argc > 3) ? atof(argv[3]) : 1e-2;
  space Xh (omega, approx);
  size_t d = omega.dimension();
  size_t k = Xh.degree();
  check_macro (k >= 1, "polynomial degree k="<<k<<" shoud be >= 1");
  Float beta = (k+1)*(k+d)/d;
  field eta_h(Xh);
  eta_h["west"] = epsilon;
  eta_h["east"] = 1;
  geo gamma_d = omega["left"] + omega["right"];
  geo Shd     = omega["internal_sides"] + gamma_d;
  trial u (Xh); test v (Xh);
  auto eta_s = 2/(1/inner(eta_h) + 1/outer(eta_h));
  auto eta_w_o = inner(eta_h)/(inner(eta_h) + outer(eta_h));
  auto eta_w_i = outer(eta_h)/(inner(eta_h) + outer(eta_h));
  auto average_w_u = eta_w_i*inner(eta_h*dot(grad_h(u),normal()))
                   + eta_w_o*outer(eta_h*dot(grad_h(u),normal()));
  auto average_w_v = eta_w_i*inner(eta_h*dot(grad_h(v),normal()))
                   + eta_w_o*outer(eta_h*dot(grad_h(v),normal()));
  form a = integrate (eta_h*dot(grad_h(u),grad_h(v)))
         + integrate (Shd, beta*penalty()*eta_s*jump(u)*jump(v)
                         - jump(u)*average_w_v
                         - jump(v)*average_w_u);
  field lh = integrate (v);
  solver_option sopt;
  sopt.iterative = false;
  problem p (a, sopt);
  field uh (Xh);
  p.solve (lh, uh);
  dout << catchmark("epsilon") << epsilon << endl
       << catchmark("u")       << uh;
}
```

Comments

The test-case from section 4.3.3, page 171, is considered here: recall that the exact solution is known. The `gamma_d` boundary domain represents the union Γ_d of the two boundary domains Γ_{left} and Γ_{right} pre-defined by the mesh. Conversely, the `Shd` domain groups the internal sides from $\mathscr{S}_h^{(i)}$ with those of Γ_d. Note the usage of the summation for merging these domains. Note the usage of the class `solver_option` for imposing a direct linear solver: indeed, when $d = 3$, **Rheolef** selects by default an iterative solver, but due to the bad condition number of the system, it is easier to use a direct one. Running the one-dimensional case writes:

```
make transmission_dg
mkgeo_grid -e 10 -region > line.geo
./transmission_dg line P1d | field -
./transmission_dg line P2d | field -
```

Fig. 4.12 plots the one-dimensional solution when $k = 1$ for two meshes. Observe that the jumps at inter-element nodes decreases very fast with mesh refinement and are no more perceptible on the plots. Recall that the Dirichlet boundary conditions at $x = 0$ and $x = 1$ is only weakly imposed: the corresponding jump at the boundary is small on the finer mesh. When $k \geqslant 2$, the approximate solution coincides with the interpolation of the exact one, since the exact solution is piecewise quadratic and the interface of discontinuity coincides with internal mesh sides.

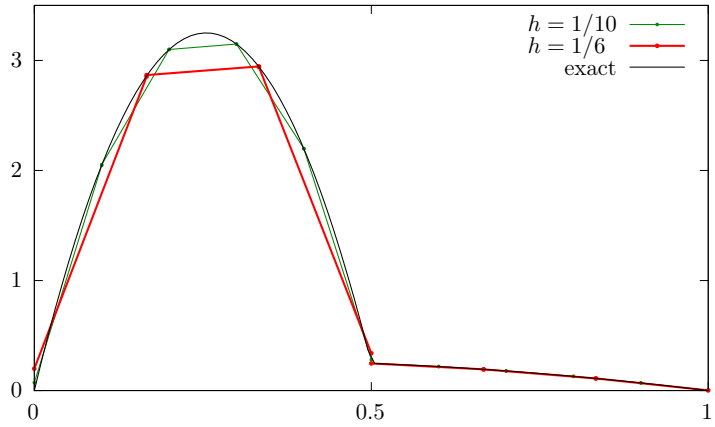


Figure 4.12: The discontinuous Galerkin method for the transmission problem with $\varepsilon = 10^{-2}$, $d = 1$, P_{1d} approximation.

The two-dimensional case writes:

```
mkgeo_grid -t 10 -region > square.geo
./transmission_dg square P1d | field -elevation -
./transmission_dg square P2d | field -elevation -
```

and the three-dimensional one

```
mkgeo_grid -T 10 -region > cube.geo
./transmission_dg cube P1d | field -
./transmission_dg cube P2d | field -
```

4.3.4 Nonlinear scalar hyperbolic problems with diffusion

A time-dependent nonlinear second order problem with nonlinear first order dominated terms problem writes:

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

$$\frac{\partial u}{\partial t} + \operatorname{div}\mathbf{f}(u) - \varepsilon\Delta u = 0 \quad \text{in }]0,T[\times\Omega \tag{4.13a}$$

$$u(t=0) = u_0 \quad \text{in } \Omega \tag{4.13b}$$

$$u = g \text{ on }]0,T[\times\partial\Omega \tag{4.13c}$$

where $\varepsilon > 0$, $T > 0$, $\Omega \subset \mathbb{R}^d$, $d = 1, 2, 3$ and the initial condition $\mathbf{u}_0$ being known. The function $\mathbf{f} : \mathbb{R} \longrightarrow \mathbb{R}^d$ is also known and supposed to be continuously differentiable. The initial data u_0, defined in Ω, and the boundary one, g, defined on $\partial\Omega$ are given.

Comparing (4.13a)-(4.13c) with the non-diffusive case (4.3a)-(4.3c) page 157, the second order term has been added in (4.13a) and the upstream boundary condition has been replaced by a Dirichlet one (4.13c).

4.3.5 Example: the Burgers equation with diffusion

Problem statement and its exact solution

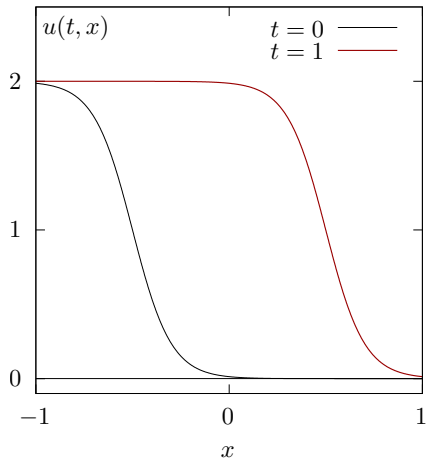


Figure 4.13: An exact solution for the Burgers equation with diffusion ($\varepsilon = 10^{-1}$, $x_0 = -1/2$).

Our model problem in this chapter is the one-dimensional Burgers equation. It was introduced in section 4.2.3, page 161 with the choice $\mathbf{f}(u) = u^2/2$, for all $u \in \mathbb{R}$. Equation (4.13a) admits an exact solution

$$u(t,x) = 1 - \tanh\left(\frac{x - x_0 - t}{2\varepsilon}\right) \tag{4.14}$$

File 4.12: `burgers_diffusion_exact.h`

```
struct u_exact {
  Float operator() (const point& x) const {
    return 1 - tanh((x[0]-x0-t)/(2*epsilon)); }
  u_exact (Float e1, Float t1=0) : epsilon(e1), t(t1), x0(-0.5) {}
  Float M() const { return 0; }
  Float epsilon, t, x0;
};
using u_init = u_exact;
using g = u_exact;
```

The solution is represents on Fig. 4.13. Here x_0 represent the position of the front at $t = 0$ and ε is a characteristic width of the front. The initial and boundary condition are chosen such that $u(t, x)$ is the solution of (4.13a)-(4.13c).

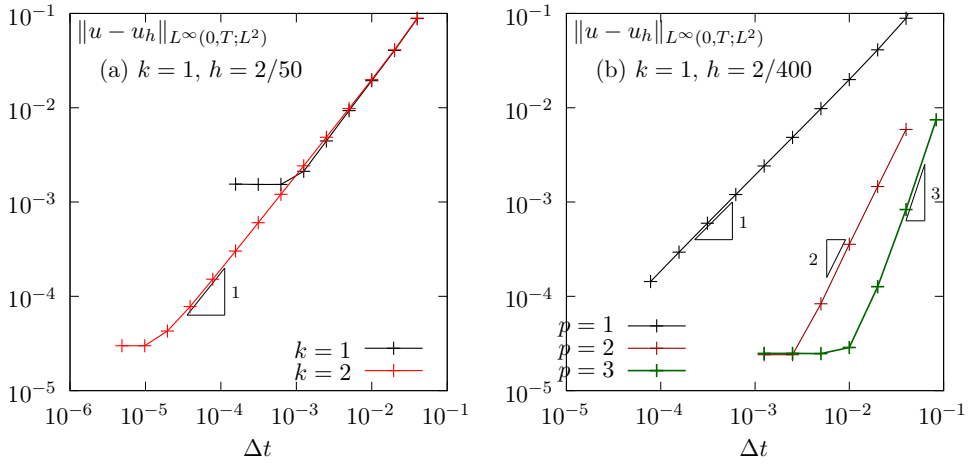


Figure 4.14: Convergence of the first order semi-implicit scheme for the Burgers equation with diffusion ($\epsilon = 0.1$, $T = 1$). (a) first order semi-implicit scheme ; (b) Runge-Kutta semi-implicit scheme with $p = 3$.

Fig. 4.14.a plots the error versus Δt for the semi-implicit scheme when $k = 1$ and 2, and for $h = 2/50$. The time step for which the error becomes independent upon Δt and depends only upon h is of about $\Delta t = 10^{-3}$ when $k = 1$ and of about 10^{-5} when $k = 2$. This approach is clearly inefficient for high order polynomial k and a hiher order time scheme is required.

Fig. 4.14.b plots the error versus Δt for the Runge-Kutta semi-implicit scheme with $p = 3$, $k = 1$ and $h = 2/200$. The scheme is clearly only first-order, which is still unexpected. More work is required...

Space discretization

The *discontinuous* finite element space is defined by:

$$X_h = \{v_h \in L^2(\Omega); v_{h|K} \in P_k, \ \forall K \in \mathcal{T}_h\}$$

where $k \geqslant 1$ is the polynomial degree. As elements of X_h do not belongs to $H^1(\Omega)$, due to discontinuities at inter-elements, we introduce the broken Sobolev space:

$$H^1(\mathcal{T}_h) = \{v \in L^2(\Omega); \ v_{|K} \in H^1(K), \ \forall K \in \mathcal{T}_h\}$$

such that $X_h \subset H^1(\mathcal{T}_h)$. As for the Dirichlet problem, introduce the folowing bilinear form $a_h(.,.)$ and linear for $l_h(.)$, defined for all $u, v \in H^1(\mathcal{T}_h)$ by (see e.g. di Pietro and Ern, 2012,

p. 125 and 127, eqn. (4.12)):

$$a_h(u,v) = \int_\Omega \nabla_h u.\nabla_h v \,\mathrm{d}x + \sum_{S\in\mathscr{S}_h}\int_S \left(\kappa_s \left[\!\left[u\right]\!\right]\left[\!\left[v\right]\!\right] - \left\{\!\left\{\nabla_h u.\mathbf{n}\right\}\!\right\}\left[\!\left[v\right]\!\right] - \left[\!\left[u\right]\!\right]\left\{\!\left\{\nabla_h v.\mathbf{n}\right\}\!\right\}\right) \mathrm{d}s \tag{4.15}$$

$$\ell_h(v) = \int_{\partial\Omega} \left(\kappa_s\, g\, v - g\,\nabla_h v.\mathbf{n}\right) \mathrm{d}s \tag{4.16}$$

The semi-discrete problem writes in variational form [di Pietro and Ern, 2012, p. 100]:

$(P)_h$: *find* $u_h \in C^1([0,T],X_h)$ *such that*

$$\begin{aligned}\int_\Omega \frac{\partial u_h}{\partial t} v_h \,\mathrm{d}x \int_\Omega G_h(u_h)\, v_h \,\mathrm{d}x + \varepsilon\, a_h(u_h,v_h) &= \varepsilon\,\ell_h(v_h), \quad \forall v_h \in X_h\\ u_h(t=0) &= \pi_h(u_0)\end{aligned}$$

where G_h has been introduced in (4.5), page 157.

Time discretization

Explicit Runge-Kutta scheme is possible for this problem but it leads to an excessive Courant-Friedrichs-Levy condition for the time step Δt, that is required to be lower than an upper bound that varies in $\mathcal{O}(h^2)$. The idea here is to continue to explicit the first order nonlinear terms and implicit the linear second order terms. Semi-implicit second order Runge-Kutta scheme was first introduced by Ascher, Ruuth, and Spiteri [1997] and then extended to third and fourth order by Calvo, de Frutos, and Novo [2001]. Wang, Shu, and Zhang [2015a,b] applied it in the context of the discontinuous Galerkin method. The finite dimensional problem can be rewritten as

$(P)_h$: *find* $u_h \in C^1([0,T],X_h)$ *such that*

$$\begin{aligned}\frac{\partial u_h}{\partial t} + G_h(t,u_h) + A_h(t,u_h) &= 0, \quad \forall t \in]0,T[\\ u_h(t=0) &= \pi_h(u_0)\end{aligned}$$

where G_h has been introduced in (4.5), page 157 and A_h denotes the diffusive term. The semi-implicit Runge-Kutta scheme with $p \geqslant 0$ intermediate steps writes at time step t_n:

$$u_h^{n,0} = u_h^n \tag{4.17a}$$

$$u_h^{n,i} = u_h^n - \Delta t \sum_{j=1}^{i} \alpha_{i,j} A_h\left(t_{n,j},\, u_h^{n,j}\right) - \Delta t \sum_{j=0}^{i-1} \tilde{\alpha}_{i,j} G_h\left(t_{n,j},\, u_h^{n,j}\right), \quad i = 1,\ldots,p \tag{4.17b}$$

$$u_h^{n+1} = u_h^n - \Delta t \sum_{i=1}^{p} \beta_i A_h\left(t_{n,i},\, u_h^{n,i}\right) - \Delta t \sum_{i=0}^{p} \tilde{\beta}_i G_h\left(t_{n,i},\, u_h^{n,i}\right) \tag{4.17c}$$

where $\left(u_h^{n,i}\right)_{1\leqslant i\leqslant p}$ are the $p \geqslant 1$ intermediate states, $t_{n,i} = t_n + \gamma_i \Delta t$, $\gamma_i = \sum_{j=1}^{i} \alpha_{i,j} = \sum_{j=0}^{i-1} \tilde{\alpha}_{i,j}$, and $(\alpha_{i,j})_{0\leqslant i,j\leqslant p}$, $(\tilde{\alpha}_{i,j})_{0\leqslant i,j\leqslant p}$, $(\beta_i)_{0\leqslant i\leqslant p}$ and $(\tilde{\beta}_i)_{0\leqslant i\leqslant p}$ are the coefficients of the scheme [Ascher et al., 1997, Calvo et al., 2001, Wang et al., 2015a]. At each time step, have to solve p linear systems. From (4.17b) we get for all $i = 1,\ldots,p$:

$$\left(I + \Delta t\, \alpha_{i,i} A_h\left(t_{n,i}\right)\right) u_h^{n,i} = u_h^n - \Delta t \sum_{j=1}^{i-1} \alpha_{i,j} A_h\left(t_{n,j},\, u_h^{n,j}\right) - \Delta t \sum_{j=0}^{i-1} \tilde{\alpha}_{i,j} G_h\left(t_{n,j},\, u_h^{n,j}\right)$$

Note that when the matrix coefficients of $A_h(t,.)$ are indepencdent of t, the matrix involved on the right-hand-side of the previous equation can be factored one time for all.

File 4.13: burgers_diffusion_dg.cc

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
#include "burgers.icc"
#include "burgers_flux_godunov.icc"
#include "runge_kutta_semiimplicit.icc"
#include "burgers_diffusion_exact.h"
#undef  NEUMANN
#include "burgers_diffusion_operators.icc"
int main(int argc, char**argv) {
  environment rheolef (argc, argv);
  geo omega (argv[1]);
  space Xh (omega, argv[2]);
  size_t k = Xh.degree();
  Float epsilon = (argc > 3) ? atof(argv[3]) : 0.1;
  size_t nmax   = (argc > 4) ? atoi(argv[4]) : 500;
  Float  tf     = (argc > 5) ? atof(argv[5]) : 1;
  size_t p      = (argc > 6) ? atoi(argv[6]) : min(k+1,rk::pmax);
  Float delta_t = tf/nmax;
  size_t d = omega.dimension();
  Float beta = (k+1)*(k+d)/Float(d);
  trial u (Xh); test v (Xh);
  form m = integrate (u*v);
  integrate_option iopt;
  iopt.invert = true;
  form inv_m = integrate (u*v, iopt);
  form a = epsilon*(
               integrate (dot(grad_h(u),grad_h(v)))
#ifdef  NEUMANN
            + integrate ("internal_sides",
#else //  NEUMANN
            + integrate ("sides",
#endif //  NEUMANN
                    beta*penalty()*jump(u)*jump(v)
                  - jump(u)*average(dot(grad_h(v),normal()))
                  - jump(v)*average(dot(grad_h(u),normal())));
  vector<problem> pb (p+1);
  for (size_t i = 1; i <= p; ++i) {
    form ci = m + delta_t*rk::alpha[p][i][i]*a;
    pb[i] = problem(ci);
  }
  vector<field> uh(p+1, field(Xh,0));
  uh[0] = lazy_interpolate (Xh, u_init(epsilon));
  branch even("t","u");
  dout << catchmark("epsilon") << epsilon << endl
       << even(0,uh[0]);
  for (size_t n = 0; n < nmax; ++n) {
    Float tn = n*delta_t;
    Float t  = tn + delta_t;
    field uh_next = uh[0] - delta_t*rk::tilde_beta[p][0]*(inv_m*gh(epsilon, tn, uh[0], v));
    for (size_t i = 1; i <= p; ++i) {
      Float ti = tn + rk::gamma[p][i]*delta_t;
      field rhs = m*uh[0] - delta_t*rk::tilde_alpha[p][i][0]*gh(epsilon, tn, uh[0], v);
      for (size_t j = 1; j <= i-1; ++j) {
        Float tj = tn + rk::gamma[p][j]*delta_t;
        rhs -= delta_t*( rk::alpha[p][i][j]*(a*uh[j] - lh(epsilon,tj,v))
                       + rk::tilde_alpha[p][i][j]*gh(epsilon, tj, uh[j], v));
      }
      rhs += delta_t*rk::alpha[p][i][i]*lh (epsilon, ti, v);
      pb[i].solve (rhs, uh[i]);
      uh_next -= delta_t*(inv_m*( rk::beta[p][i]*(a*uh[i] - lh(epsilon,ti,v))
                                + rk::tilde_beta[p][i]*gh(epsilon, ti, uh[i], v)));
    }
    uh_next = limiter(uh_next);
    dout << even(tn+delta_t,uh_next);
    uh[0] = uh_next;
  }
}
```

File 4.14: burgers_diffusion_operators.icc

```
field lh (Float epsilon, Float t, const test& v) {
#ifdef  NEUMANN
  return field (v.get_vf_space(), 0);
#else //  NEUMANN
  size_t d = v.get_vf_space().get_geo().dimension();
  size_t k = v.get_vf_space().degree();
  Float beta = (k+1)*(k+d)/Float(d);
  return epsilon*integrate ("boundary",
         beta*penalty()*g(epsilon,t)*v
       - g(epsilon,t)*dot(grad_h(v),normal()));
#endif //  NEUMANN
}
field gh (Float epsilon, Float t, const field& uh, const test& v) {
  return - integrate (dot(compose(f,uh),grad_h(v)))
         + integrate ("internal_sides",
             compose (phi, normal(), inner(uh), outer(uh))*jump(v))
         + integrate ("boundary",
             compose (phi, normal(), uh, g(epsilon,t))*v);
}
```

The included file 'runge_kutta_semiimplicit.icc' is not shown here but is available in the example directory.

Running the program

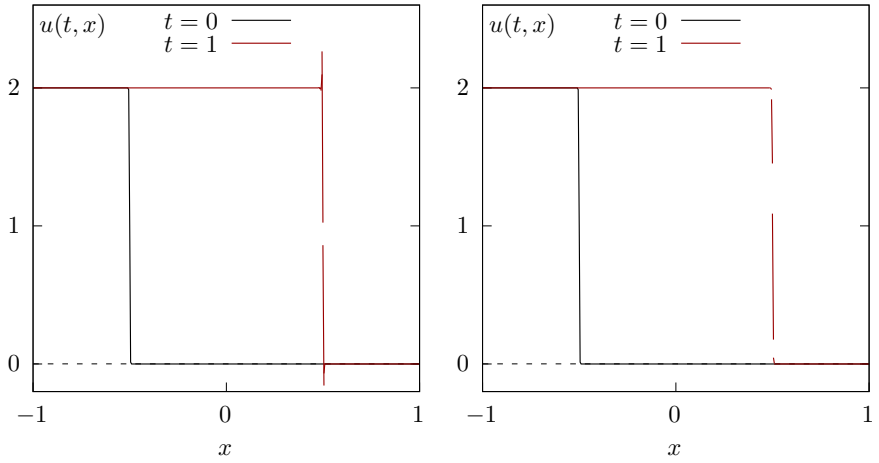


Figure 4.15: Burgers equation with a small diffusion ($\varepsilon = 10^{-3}$). Third order in time semi-implicit scheme with P_{1d} element. (left) without limiter ; (right) with limiter.

Running the program writes with $h = 2/400$ and $\varepsilon = 10^{-2}$ writes:

```
make burgers_diffusion_dg
mkgeo_grid -e 400 -a -1 -b 1 > line.geo
mpirun -np 8 ./burgers_diffusion_dg line P1d 0.01 1000 1 3 > line.branch
branch -gnuplot line.branch -umin -0.1 -umax 2.1
```

It could take about one minute, depending on your computer. Decreasing $\varepsilon = 10^{-3}$ leads to a sharper solution:

```
mpirun -np 8 ./burgers_diffusion_dg line P1d 0.001 1000 1 3 > line.branch
branch -gnuplot line.branch -umin -0.1 -umax 2.1
```

As mentioned by Wang et al. [2015a], the time step should be chosen smaller when ε decreases. The result is shown on Fig. 4.15.left. Observe the oscillations near the smoothed shock when there is no limiter while the value goes outside $[0, 2]$. Conversely, with a limiter (see Fig. 4.15.right) the approximate solution is decreasing and there are no more oscillations: the values remains in the range $[0 : 2]$.

4.4 Fluids and solids computations revisited

4.4.1 The linear elasticity problem

The elasticity problem (2.2) has been introduced in volume 1, section 2.1.1, page 43.
(P): *find* $\mathbf{u}$ *such that*

$$\begin{aligned} -\mathbf{div}\left(\lambda \mathrm{div}(\mathbf{u}).I + 2D(\mathbf{u})\right) &= \mathbf{f} \text{ in } \Omega \\ \mathbf{u} &= \mathbf{g} \text{ on } \partial\Omega \end{aligned}$$

where $\lambda \geqslant -1$ is a constant and $\mathbf{f}, \mathbf{g}$ given. This problem is a natural extension to vector-valued field of the Poisson problem with Dirichlet boundary conditions.
The variational formulation writes:
$(FV)_h$: *find* $\mathbf{u} \in \mathbf{V}(\mathbf{g})$ *such that*

$$a(\mathbf{u}, \mathbf{v}) = l_h(\mathbf{v}),\ \forall \mathbf{v} \in \mathbf{V}(0)$$

where

$$\begin{aligned} \mathbf{V}(\mathbf{g}) &= \{\mathbf{v} \in H^1(\Omega)^d;\ \mathbf{v} = \mathbf{g} \text{ on } \partial\Omega\} \\ a(\mathbf{u}, \mathbf{v}) &= \int_\Omega \left(\lambda\, \mathrm{div}(\mathbf{u})\, \mathrm{div}(\mathbf{v}) + 2D(\mathbf{u}) : D(\mathbf{v})\right)\, \mathrm{d}x \\ l(\mathbf{v}) &= \int_\Omega \mathbf{f}.\mathbf{v}\, \mathrm{d}x \end{aligned}$$

The discrete variational formulation writes:
$(FV)_h$: *find* $u_h \in \mathbf{X}_h$ *such that*

$$a_h(u_h, v_h) = l_h(v_h),\ \forall v_h \in \mathbf{X}_h$$

where

$$\begin{aligned} \mathbf{X}_h &= \{\mathbf{v}_h \in L^2(\Omega)^d; \mathbf{v}_{h|K} \in P_k^d,\ \forall K \in \mathcal{T}_h\} \\ a_h(\mathbf{u}, \mathbf{v}) &= \int_\Omega \left(\lambda\, \mathrm{div}_h(\mathbf{u})\, \mathrm{div}_h(\mathbf{v}) + 2D_h(\mathbf{u}) : D_h(\mathbf{v})\right)\, \mathrm{d}x \\ &+ \sum_{S \in \mathscr{S}_h} \int_S \left(\beta \varpi_s [\![\mathbf{u}]\!].[\![\mathbf{v}]\!] - [\![\mathbf{u}]\!].\{\!\{\lambda \mathrm{div}_h(\mathbf{v})\mathbf{n} + 2D_h(\mathbf{v})\mathbf{n}\}\!\} - [\![\mathbf{v}]\!].\{\!\{\lambda \mathrm{div}_h(\mathbf{u})\mathbf{n} + 2D_h(\mathbf{u})\mathbf{n}\}\!\}\right)\, \mathrm{d}s \\ l_h(\mathbf{v}) &= \int_\Omega \mathbf{f}.\mathbf{v}\, \mathrm{d}x + \int_{\partial\Omega} \mathbf{g}.\left(\beta \varpi_s \mathbf{v} - \lambda \mathrm{div}_h(\mathbf{v})\mathbf{n} - 2D_h(\mathbf{v})\mathbf{n}\right)\, \mathrm{d}s \end{aligned}$$

where $k \geqslant 1$ is the polynomial degree in $\mathbf{X}_h$.

File 4.15: `elasticity_taylor_dg.cc`

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
#include "taylor.h"
int main(int argc, char**argv) {
  environment rheolef (argc, argv);
  geo omega (argv[1]);
  space Xh (omega, argv[2], "vector");
  Float lambda = (argc > 3) ? atof(argv[3]) : 1;
  size_t d = omega.dimension();
  size_t k = Xh.degree();
  Float beta = (k+1)*(k+d)/Float(d);
  trial u (Xh); test v (Xh);
  form  a = integrate (lambda*div_h(u)*div_h(v) + 2*ddot(Dh(u),Dh(v)))
          + integrate (omega.sides(),
                  beta*penalty()*dot(jump(u),jump(v))
                - lambda*dot(jump(u),average(div_h(v)*normal()))
                - lambda*dot(jump(v),average(div_h(u)*normal()))
                - 2*dot(jump(u),average(Dh(v)*normal()))
                - 2*dot(jump(v),average(Dh(u)*normal())));
  field lh = integrate (dot(f(),v))
           + integrate (omega.boundary(),
                  beta*penalty()*dot(g(),jump(v))
                - lambda*dot(g(),average(div_h(v)*normal()))
                - 2*dot(g(),average(Dh(v)*normal())));
  field uh(Xh);
  problem p (a);
  p.solve (lh, uh);
  dout << uh;
}
```

Comments

The data are given when $d = 2$ by:

$$\mathbf{g}(x) = \begin{pmatrix} -\cos(\pi x_0)\sin(\pi x_1) \\ \sin(\pi x_0)\cos(\pi x_1) \end{pmatrix} \quad \text{and} \quad \mathbf{f} = 2\pi^2\mathbf{g} \tag{4.18}$$

This choice is convenient since the exact solution is known $\mathbf{u} = \mathbf{g}$. This benchmark solution was proposed by Taylor [1923] in the context of the Stokes problem. Note that the solution is independent of λ since $\mathrm{div}(\mathbf{u}) = 0$.

File 4.16: `taylor.h`

```
struct g {
  point operator() (const point& x) const {
    return point(-cos(pi*x[0])*sin(pi*x[1]),
                  sin(pi*x[0])*cos(pi*x[1])); }
  g() : pi(acos(Float(-1.0))) {}
  const Float pi;
};
struct f {
  point operator() (const point& x) const { return 2*sqr(pi)*_g(x); }
  f() : pi(acos(Float(-1.0))), _g() {}
  const Float pi; g _g;
};
```

As the exact solution is known, the error can be computed. The code `elasticity_taylor_error_dg.cc` and its header file `taylor_exact.h` compute the error in L^2, L^∞ and energy norms. These files are not listed here but are available in the **Rheolef** example directory. The computation writes:

```
make elasticity_taylor_dg elasticity_taylor_error_dg
mkgeo_grid -t 10 > square.geo
```

```
./elasticity_taylor_dg square P1d | ./elasticity_taylor_error_dg
./elasticity_taylor_dg square P2d | ./elasticity_taylor_error_dg
```

4.4.2 The Stokes problem

Let us consider the Stokes problem for the driven cavity in $\Omega =]0,1[^d$, $d = 2, 3$. The problem has been introduced in volume 1, section 2.1.4, page 53.

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

$$\begin{array}{rcrcl} -\mathbf{div}(2D(\mathbf{u})) & + & \boldsymbol{\nabla} p & = & \mathbf{f} \text{ in } \Omega, \\ -\operatorname{div}\mathbf{u} & & & = & 0 \text{ in } \Omega, \\ \mathbf{u} & & & = & \mathbf{g} \text{ on } \partial\Omega \end{array}$$

where $\mathbf{f}$ and $\mathbf{g}$ are given. This problem is the extension to divergence free vector fields of the elasticity problem. The variational formulation writes:

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

$$\begin{array}{rcrcl} a(\mathbf{u},\mathbf{v}) & + & b(\mathbf{v},p) & = & l(\mathbf{v}),\ \forall \mathbf{v} \in \mathbf{V}(0), \\ b(\mathbf{u},q) & & & = & 0,\ \forall q \in L^2(\Omega) \end{array} \tag{4.19}$$

where

$$\begin{aligned} \mathbf{V}(\mathbf{g}) &= \{\mathbf{v} \in H^1(\Omega)^d;\ \mathbf{v} = \mathbf{g} \ \text{ on } \ \partial\Omega\} \\ a(\mathbf{u},\mathbf{v}) &= \int_\Omega 2D(\mathbf{u}) : D(\mathbf{v})\,\mathrm{d}x \\ b(\mathbf{u},q) &= -\int_\Omega \operatorname{div}(\mathbf{u})\, q\,\mathrm{d}x \\ l(\mathbf{v}) &= \int_\Omega \mathbf{f}.\mathbf{v}\,\mathrm{d}x \end{aligned}$$

The discrete variational formulation writes:

$(VF)_h$ *find* $\mathbf{u}_h \in \mathbf{X}_h$ *and* $p_h \in Q_h$ *such that:*

$$\begin{array}{rcrcl} a_h(\mathbf{u}_h,\mathbf{v}_h) & + & b_h(\mathbf{v}_h,p_h) & = & l_h(\mathbf{v}_h),\ \forall \mathbf{v}_h \in \mathbf{X}_h, \\ b_h(\mathbf{u}_h,q_h) & - & c_h(p_h,q_h) & = & k_h(q),\ \forall q_h \in Q_h. \end{array} \tag{4.20}$$

The discontinuous finite element spaces are defined by:

$$\begin{aligned} \mathbf{X}_h &= \{\mathbf{v}_h \in L^2(\Omega)^d; \mathbf{v}_{h|K} \in P_k^d,\ \forall K \in \mathcal{T}_h\} \\ Q_h &= \{q_h \in L^2(\Omega)^d; q_{h|K} \in P_k^d,\ \forall K \in \mathcal{T}_h\} \end{aligned}$$

where $k \geqslant 1$ is the polynomial degree. Note that velocity and pressure are approximated by the same polynomial order. This method was introduced by Cockburn et al. [2002] and some recent theoretical results can be founded in di Pietro and Ern [2010]. The forms are defined for all

$u, v \in H^1(\mathcal{T}_h)^d$ and $q \in L^2(\Omega)$ by (see e.g. di Pietro and Ern, 2012, p. 249):

$$
\begin{aligned}
a_h(\mathbf{u},\mathbf{v}) &= \int_\Omega 2D_h(\mathbf{u}) : D_h(\mathbf{v})\,\mathrm{d}x \\
&\quad + \sum_{S\in\mathscr{S}_h} \int_S \left(\beta\varpi_s [\![\mathbf{u}]\!].[\![\mathbf{v}]\!] - [\![\mathbf{u}]\!].\{\!\{2D_h(\mathbf{v})\mathbf{n}\}\!\} - [\![\mathbf{v}]\!].\{\!\{2D_h(\mathbf{u})\mathbf{n}\}\!\}\right)\,\mathrm{d}s \\
b_h(\mathbf{u},q) &= \int_\Omega \mathbf{u}.\nabla_h q\,\mathrm{d}x - \sum_{S\in\mathscr{S}_h^{(i)}} \int_S \{\!\{\mathbf{u}\}\!\}.\mathbf{n}\,[\![q]\!]\,\mathrm{d}s \\
c_h(p,q) &= \sum_{S\in\mathscr{S}_h^{(i)}} \int_S h_s\,[\![p]\!]\,[\![q]\!]\,\mathrm{d}s \\
l_h(\mathbf{v}) &= \int_\Omega \mathbf{f}.\mathbf{v}\,\mathrm{d}s + \int_{\partial\Omega} \mathbf{g}.\left(\beta\varpi_s\,\mathbf{v} - 2D_h(\mathbf{v})\,\mathbf{n}\right)\,\mathrm{d}s \\
k_h(q) &= \int_{\partial\Omega} \mathbf{g}.\mathbf{n}\,q\,\mathrm{d}s
\end{aligned}
$$

The stabilization form c_h controls the pressure jump across internal sides. This stabilization term is necessary when using equal order polynomial approximation for velocity and pressure. The definition of the forms is grouped in a subroutine: it will be reused later for the Navier-Stokes problem.

File 4.17: `stokes_dirichlet_dg.icc`

```
void stokes_dirichlet_dg (const space& Xh, const space& Qh,
  form& a, form& b, form& c, form& mp, field& lh, field& kh,
  integrate_option iopt = integrate_option())
{
  size_t k = Xh.degree();
  size_t d = Xh.get_geo().dimension();
  Float beta = (k+1)*(k+d)/Float(d);
  trial u (Xh), p (Qh);
  test  v (Xh), q (Qh);
  a  = integrate (2*ddot(Dh(u),Dh(v)), iopt)
     + integrate ("sides",    beta*penalty()*dot(jump(u),jump(v))
                 - 2*dot(jump(u),average(Dh(v)*normal()))
                 - 2*dot(jump(v),average(Dh(u)*normal())), iopt);
  lh = integrate (dot(f(),v), iopt)
     + integrate ("boundary", beta*penalty()*dot(g(),v)
                 - 2*dot(g(),Dh(v)*normal()), iopt);
  b  = integrate (dot(u,grad_h(q)), iopt)
     + integrate ("internal_sides", - dot(average(u),normal())*jump(q), iopt);
  kh = integrate ("boundary", dot(g(),normal())*q, iopt);
  c  = integrate ("internal_sides", h_local()*jump(p)*jump(q), iopt);
  mp = integrate (p*q, iopt);
}
```

A simple test program writes:

File 4.18: stokes_taylor_dg.cc

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
#include "taylor.h"
#include "stokes_dirichlet_dg.icc"
int main(int argc, char**argv) {
  environment rheolef (argc, argv);
  geo omega (argv[1]);
  space Xh (omega, argv[2], "vector");
  space Qh (omega, argv[2]);
  form a, b, c, mp;
  field lh, kh;
  stokes_dirichlet_dg (Xh, Qh, a, b, c, mp, lh, kh);
  field uh (Xh, 0), ph (Qh, 0);
  problem_mixed stokes (a, b, c);
  stokes.set_metric (mp);
  stokes.solve (lh, kh, uh, ph);
  dout << catchmark("u")  << uh
       << catchmark("p")  << ph;
}
```

Comments

The data are given when $d = 2$ by (4.18). This choice is convenient since the exact solution is known $\mathbf{u} = \mathbf{g}$ and $p = 0$. The code `stokes_taylor_error_dg.cc` compute the error in L^2, L^∞ and energy norms. This code it is not listed here but is available in the **Rheolef** example directory. The computation writes:

```
make stokes_taylor_dg stokes_taylor_error_dg
mkgeo_grid -t 10 > square.geo
./stokes_taylor_dg square P1d | ./stokes_taylor_error_dg
./stokes_taylor_dg square P2d | ./stokes_taylor_error_dg
```

4.5 The stationnary Navier-Stokes equations

4.5.1 Problem statemment

The Navier-Stokes problem has been already introduced in volume 1, section 4.5 page 185. Here we consider the stationary version of this problem. Let $Re \geqslant 0$ be the Reynolds number. The problem writes:

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

$$\begin{array}{rrcrcl} Re\,(\mathbf{u}.\nabla)\mathbf{u} & -\,\mathbf{div}(2D(\mathbf{u})) & + & \boldsymbol{\nabla} p & = & \mathbf{f} \text{ in } \Omega, \\ & -\,\mathrm{div}\,\mathbf{u} & & & = & 0 \text{ in } \Omega, \\ & \mathbf{u} & & & = & \mathbf{g} \text{ on } \partial\Omega \end{array}$$

Note that, when $Re > 0$, the problem is nonlinear, due to the inertia term $\mathbf{u}.\nabla\mathbf{u}$. When $Re = 0$ the problem reduces to the linear Stokes problem, presented in the previous section/

The variational formulation of this nonlinear problem writes:

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

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

where the space $\mathbf{V}(\mathbf{g})$ and forms a, b and l are given as in the previous section 4.4.2 for the Stokes

problem and the trilinear form $t(.;.,.)$ is given by:

$$t(\mathbf{w};\, \mathbf{u},\, \mathbf{v}) \quad = \quad \int_\Omega ((\mathbf{w}.\nabla)\mathbf{u}).\mathbf{v}\,\mathrm{d}x$$

4.5.2 The discrete problem

Let

$$t(\mathbf{w};\, \mathbf{u},\, \mathbf{u}) \quad = \quad \int_\Omega (\mathbf{w}.\nabla\mathbf{u}).\mathbf{u}\,\mathrm{d}x$$

Observe that, for all $\mathbf{u},\, \mathbf{w} \in H^1(\Omega)^d$ we have

$$\begin{aligned}
\int_\Omega (\mathbf{w}.\nabla\mathbf{u}).\mathbf{u}\,\mathrm{d}x &= \sum_{i,j=0}^{d-1} \int_\Omega u_i\, w_j\, \partial_j(u_i)\,\mathrm{d}x \\
&= \sum_{i,j=0}^{d-1} -\int_\Omega u_i\, \partial_j(u_i\, w_j)\,\mathrm{d}x + \int_{\partial\Omega} u_i^2\, w_j\, n_j\,\mathrm{d}x \\
&= \sum_{i,j=0}^{d-1} -\int_\Omega u_i\, \partial_j(u_i)\, w_j\,\mathrm{d}x - \int_\Omega u_i^2\, \partial_j(w_j)\,\mathrm{d}x + \int_{\partial\Omega} u_i^2\, w_j\, n_j\,\mathrm{d}x \\
&= -\int_\Omega (\mathbf{w}.\nabla\mathbf{u}).\mathbf{u}\,\mathrm{d}x - \int_\Omega \mathrm{div}(\mathbf{w})\,|\mathbf{u}|^2\,\mathrm{d}x + \int_{\partial\Omega} \mathbf{w}.\mathbf{n}\,|\mathbf{u}|^2\,\mathrm{d}s \qquad (4.21)
\end{aligned}$$

Thus

$$t(\mathbf{w};\, \mathbf{u},\, \mathbf{u}) = \int_\Omega (\mathbf{w}.\nabla\mathbf{u}).\mathbf{u}\,\mathrm{d}x = -\frac{1}{2}\int_\Omega \mathrm{div}(\mathbf{w})\,|\mathbf{u}|^2\,\mathrm{d}x + \frac{1}{2}\int_{\partial\Omega} \mathbf{w}.\mathbf{n}\,|\mathbf{u}|^2\,\mathrm{d}s$$

When $\mathrm{div}(\mathbf{w}) = 0$, the trilinear form $t(.;.,.)$ reduces to a boundary term: it is formally skew-symmetric. The skew-symmetry of t is an important property: let $(\mathbf{v}, q) = (\mathbf{u}, p)$ as test functions in (FV). We obtain:

$$a(\mathbf{u}, \mathbf{u}) = l(\mathbf{u})$$

In other words, we obtain the same energy balance as for the Stokes flow and inertia do not contribute to the energy balance. This is an important property and we aim at obtaining the same one at the discrete level. As the discrete solution $\mathbf{u}_h$ is not exactly divergence free, following Temam, we introduce the following modified trilinear form:

$$t^*(\mathbf{w};\, \mathbf{u},\, \mathbf{v}) \quad = \quad \int_\Omega \left((\mathbf{w}.\nabla\mathbf{u})\,.\mathbf{v} + \frac{1}{2}\mathrm{div}(\mathbf{w})\,\mathbf{u}.\mathbf{v} \right) \mathrm{d}x - \frac{1}{2}\int_{\partial\Omega} (\mathbf{w}.\mathbf{n})\,\mathbf{u}.\mathbf{v}\,\mathrm{d}s,\ \forall \mathbf{u}, \mathbf{v}, \mathbf{w} \in H^1(\Omega)^d$$

This form integrates the non-vanishing terms and we have:

$$t^*(\mathbf{w};\, \mathbf{u},\, \mathbf{u}) \quad = \quad 0,\ \forall \mathbf{u}, \mathbf{w} \in H^1(\Omega)^d$$

When the discrete solution is not exactly divergence free, it is better to use t^* than t.

The discontinuous finite element spaces $\mathbf{X}_h$ and Q_h and forms a_h, b_h, c_h, l_h and k_h are defined as in the previous section. Let us introduce t_h^*, the following discrete trilinear form, defined for all $\mathbf{u}_h, \mathbf{v}_h, \mathbf{w}_h \in \mathbf{X}_h$:

$$t_h^*(\mathbf{w}_h;\, \mathbf{u}_h,\, \mathbf{v}_h) \quad = \quad \int_\Omega \left((\mathbf{w}_h.\nabla_h\mathbf{u}_h)\,.\mathbf{v}_h + \frac{1}{2}\mathrm{div}_h(\mathbf{w}_h)\,\mathbf{u}_h.\mathbf{v}_h \right) \mathrm{d}x - \frac{1}{2}\int_{\partial\Omega} (\mathbf{w}_h.\mathbf{n})\,\mathbf{u}_h.\mathbf{v}_h\,\mathrm{d}s$$

Note that t_h^* is similar to t^*: the gradient and divergence has been replaced by their broken counterpart in the first term. As $\mathbf{X}_h \not\subset H^1(\Omega)^d$, the skew-symmetry property is not expected to be true at the discrete level. Then

$$t_h^*(\mathbf{w}_h;\, \mathbf{u}_h,\, \mathbf{u}_h) \;=\; \sum_{K\in\mathscr{T}_h} \int_K \left((\mathbf{w}_h.\nabla\mathbf{u}_h)\,.\mathbf{u}_h + \frac{1}{2}\mathrm{div}(\mathbf{w}_h)\,|\mathbf{u}_h|^2 \right) \mathrm{d}x - \frac{1}{2}\int_{\partial\Omega} (\mathbf{w}_h.\mathbf{n})\,|\mathbf{u}_h|^2\,\mathrm{d}s$$

As the restriction of $\mathbf{u}_h$ and $\mathbf{w}_h$ to each $K \in \mathscr{T}_h$ belongs to $H^1(K)^d$, we have, using a similar integration by part:

$$\int_K (\mathbf{w}_h.\nabla\mathbf{u}_h).\mathbf{u}_h\,\mathrm{d}x = -\frac{1}{2}\int_K \mathrm{div}(\mathbf{w}_h)\,|\mathbf{u}_h|^2\,\mathrm{d}x + \frac{1}{2}\int_{\partial K} (\mathbf{w}_h.\mathbf{n})\,|\mathbf{u}_h|^2\,\mathrm{d}s$$

Thus

$$t_h^*(\mathbf{w}_h;\, \mathbf{u}_h,\, \mathbf{u}_h) \;=\; \frac{1}{2}\sum_{K\in\mathscr{T}_h} \int_{\partial K} (\mathbf{w}_h.\mathbf{n})\,|\mathbf{u}_h|^2\,\mathrm{d}s - \frac{1}{2}\int_{\partial\Omega} (\mathbf{w}_h.\mathbf{n})\,|\mathbf{u}_h|^2\,\mathrm{d}s$$

The terms on boundary sides vanish while those on internal sides can be grouped:

$$t_h^*(\mathbf{w}_h;\, \mathbf{u}_h,\, \mathbf{u}_h) \;=\; \frac{1}{2}\sum_{S\in\mathscr{S}_h^{(i)}} \int_S [\![|\mathbf{u}_h|^2\mathbf{w}_h]\!].\mathbf{n}\,\mathrm{d}s$$

The jump term $[\![(\mathbf{u}_h.\mathbf{v}_h)\,\mathbf{w}_h]\!].\mathbf{n}$ is not easily manageable and could be developed. A short computation shows that, for all scalar fields ϕ, φ we have on any internal side the following identity:

$$[\![\phi\varphi]\!] \;=\; [\![\phi]\!]\{\!\{\varphi\}\!\} + \{\!\{\phi\}\!\}[\![\varphi]\!] \tag{4.22}$$

$$\{\!\{\phi\varphi\}\!\} \;=\; \{\!\{\phi\}\!\}\{\!\{\varphi\}\!\} + \frac{1}{4}[\![\phi]\!][\![\varphi]\!] \tag{4.23}$$

Then

$$\begin{aligned} t_h^*(\mathbf{w}_h;\, \mathbf{u}_h,\, \mathbf{u}_h) \;&=\; \frac{1}{2}\sum_{S\in\mathscr{S}_h^{(i)}} \int_S \left(\{\!\{\mathbf{w}_h\}\!\}.\mathbf{n}\,[\![|\mathbf{u}_h|^2]\!] + [\![\mathbf{w}_h]\!].\mathbf{n}\,\{\!\{ |\mathbf{u}_h|^2 \}\!\} \right) \mathrm{d}s \\ &=\; \sum_{S\in\mathscr{S}_h^{(i)}} \int_S \left(\{\!\{\mathbf{w}_h\}\!\}.\mathbf{n}\,([\![\mathbf{u}_h]\!].\{\!\{\mathbf{u}_h\}\!\}) + \frac{1}{2}[\![\mathbf{w}_h]\!].\mathbf{n}\,\{\!\{ |\mathbf{u}_h|^2 \}\!\} \right) \mathrm{d}s \end{aligned}$$

Thus, as expected, the skew-symmetry property is no more satisfied at the discrete level, due to the jumps of the fields at the inter-element boundaries. Following the previous idea, we introduce the following modified discrete trilinear form:

$$\begin{aligned} t_h(\mathbf{w}_h;\, \mathbf{u}_h,\, \mathbf{v}_h) \;&=\; t_h^*(\mathbf{w}_h;\, \mathbf{u}_h,\, \mathbf{v}_h) - \sum_{S\in\mathscr{S}_h^{(i)}} \int_S \left(\{\!\{\mathbf{w}_h\}\!\}.\mathbf{n}\,([\![\mathbf{u}_h]\!].\{\!\{\mathbf{v}_h\}\!\}) + \frac{1}{2}[\![\mathbf{w}_h]\!].\mathbf{n}\,\{\!\{\mathbf{u}_h.\mathbf{v}_h\}\!\} \right) \mathrm{d}s \\ &=\; \int_\Omega \left((\mathbf{w}_h.\nabla_h\mathbf{u}_h)\,.\mathbf{v}_h + \frac{1}{2}\mathrm{div}_h(\mathbf{w}_h)\,\mathbf{u}_h.\mathbf{v}_h \right) \mathrm{d}x - \frac{1}{2}\int_{\partial\Omega} (\mathbf{w}_h.\mathbf{n})\,\mathbf{u}_h.\mathbf{v}_h\,\mathrm{d}s \\ &\quad - \sum_{S\in\mathscr{S}_h^{(i)}} \int_S \left(\{\!\{\mathbf{w}_h\}\!\}.\mathbf{n}\,([\![\mathbf{u}_h]\!].\{\!\{\mathbf{v}_h\}\!\}) + \frac{1}{2}[\![\mathbf{w}_h]\!].\mathbf{n}\,\{\!\{\mathbf{u}_h.\mathbf{v}_h\}\!\} \right) \mathrm{d}s \end{aligned} \tag{4.24}$$

This expression has been proposed by di Pietro and Ern [2010, p. 22], eqn (72) (see also di Pietro and Ern, 2012, p. 272, eqn (6.57)). The boundary term introduced in t_h may be compensated in the right-hand side:

$$l_h^*(\mathbf{v}) \;:=\; l_h(\mathbf{v}) - \frac{Re}{2}\int_{\partial\Omega} (\mathbf{g}.\mathbf{n})\,\mathbf{g}.\mathbf{v}_h\,\mathrm{d}s$$

Note that the boundary term introduced in t_h is compensated in the right-hand side l_h^*.

File 4.19: `inertia.h`

```
template<class W, class U, class V>
form inertia (W w, U u, V v,
  integrate_option iopt = integrate_option())
{
  return
    integrate (dot(grad_h(u)*w,v) + 0.5*div_h(w)*dot(u,v), iopt)
  + integrate ("boundary", - 0.5*dot(w,normal())*dot(u,v), iopt)
  + integrate ("internal_sides",
     - dot(average(w),normal())*dot(jump(u),average(v))
     - 0.5*dot(jump(w),normal())
          *(dot(average(u),average(v)) + 0.25*dot(jump(u),jump(v))), iopt);
}
field inertia_fix_rhs (test v,
  integrate_option iopt = integrate_option())
{
  return integrate("boundary", - 0.5*dot(g(),normal())*dot(g(),v), iopt);
}
```

The discrete problem is

$(FV)_h$: *find* $\mathbf{u}_h \in \mathbf{X}_h$ *and* $p \in Q_h$ *such that*

$$\begin{array}{rcccccl} Re\, t_h(\mathbf{u}_h;\, \mathbf{u}_h,\, \mathbf{v}_h) & + & a_h(\mathbf{u}_h, \mathbf{v}_h) & + & b_h(\mathbf{v}_h, p_h) & = & l_h^*(\mathbf{v}_h),\ \forall \mathbf{v}_h \in \mathbf{X}_h, \\ & & b_h(\mathbf{u}_h, q_h) & - & c_h(p_h, q_h) & = & k_h(q),\ \forall q_h \in Q_h \end{array} \tag{4.25}$$

The simplest approach for solving the discrete problem is to consider a fixed-point algorithm. The sequence $\left(\mathbf{u}_h^{(k)}\right)_{k \geqslant 0}$ is defined by recurrence as:

- $k = 0$: let $\mathbf{u}_h^{(0)} \in \mathbf{X}_h$ being known.

- $k \geqslant 0$: let $\mathbf{u}_h^{(k-1)} \in \mathbf{X}_h$ given. Find $\mathbf{u}_h^{(k)} \in \mathbf{X}_h$ and $p_h^{(k)} \in Q_h$ such that

$$\begin{array}{rcccccl} Re\, t_h\left(\mathbf{u}_h^{(k-1)};\, \mathbf{u}_h^{(k)},\, \mathbf{v}_h\right) & + & a_h\left(\mathbf{u}_h^{(k)}, \mathbf{v}_h\right) & + & b_h\left(\mathbf{v}_h, p_h^{(k)}\right) & = & l_h^*(\mathbf{v}_h),\ \forall \mathbf{v}_h \in \mathbf{X}_h, \\ & & b_h\left(\mathbf{u}_h^{(k)}, q_h\right) & - & c_h\left(p_h^{(k)}, q_h\right) & = & k_h(q),\ \forall q_h \in Q_h. \end{array}$$

At each step $k \geqslant 0$, this algorithm involves a linear subproblem of Stokes-type.

File 4.20: navier_stokes_taylor_dg.cc

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
#include "taylor.h"
#include "stokes_dirichlet_dg.icc"
#include "inertia.h"
int main(int argc, char**argv) {
  environment rheolef (argc, argv);
  geo omega (argv[1]);
  space Xh (omega, argv[2], "vector");
  space Qh (omega, argv[2]);
  Float  Re       = (argc > 3) ? atof(argv[3]) : 1;
  size_t max_iter = (argc > 4) ? atoi(argv[4]) : 1;
  form a, b, c, mp;
  field lh, kh;
  stokes_dirichlet_dg (Xh, Qh, a, b, c, mp, lh, kh);
  field uh (Xh, 0), ph (Qh, 0);
  problem_mixed stokes (a, b, c);
  stokes.set_metric (mp);
  stokes.solve (lh, kh, uh, ph);
  trial u (Xh); test v (Xh);
  form a1 = a + Re*inertia (uh, u, v);
  lh += Re*inertia_fix_rhs (v);
  derr << "#k r as" << endl;
  for (size_t k = 0; k < max_iter; ++k) {
    stokes = problem_mixed (a1, b, c);
    stokes.set_metric (mp);
    stokes.solve (lh, kh, uh, ph);
    form th = inertia (uh, u, v);
    a1 = a + Re*th;
    field rh = a1*uh + b.trans_mult(ph) - lh;
    derr << k << " " << rh.max_abs() << " " << th(uh,uh) << endl;
  }
  dout << catchmark("Re") << Re << endl
       << catchmark("u")  << uh
       << catchmark("p")  << ph;
}
```

Comments

The data are given when $d = 2$ by (4.18). This choice is convenient since the exact solution is known $\mathbf{u} = \mathbf{g}$ and $p = -(Re/4)(\cos(2\pi x_0) + \cos(2\pi x_1))$. The code `navier_stokes_taylor_error_dg.cc` compute the error in L^2, L^∞ and energy norms. This code it is not listed here but is available in the **Rheolef** example directory. The computation writes:

```
make navier_stokes_taylor_dg navier_stokes_taylor_error_dg
./navier_stokes_taylor_dg square P1d 10 10 | ./navier_stokes_taylor_error_dg
./navier_stokes_taylor_dg square P2d 10 10 | ./navier_stokes_taylor_error_dg
```

4.5.3 A conservative variant

Remark the identity

$$\mathbf{div}(\mathbf{u} \otimes \mathbf{u}) = (\mathbf{u}.\nabla)\mathbf{u} + \mathrm{div}(\mathbf{u})\,\mathbf{u}$$

The momentum conservation can be rewritten in conservative form and the problem writes:
($\tilde{P}$): *find* $\mathbf{u}$ *and* p, *defined in* Ω, *such that*

$$\begin{array}{rcrcl} \mathbf{div}(Re\,\mathbf{u} \otimes \mathbf{u} - 2D(\mathbf{u})) & + & \boldsymbol{\nabla} p & = & \mathbf{f} \text{ in } \Omega, \\ -\mathrm{div}\,\mathbf{u} & & & = & 0 \text{ in } \Omega, \\ \mathbf{u} & & & = & \mathbf{g} \text{ on } \partial\Omega \end{array}$$

Note the Green formulae (see volume 1, appendix A.1.2, page 279):

$$\int_\Omega \mathbf{div}(\mathbf{u}\otimes\mathbf{u}).\mathbf{v}\,\mathrm{d}x = -\int_\Omega (\mathbf{u}\otimes\mathbf{u}):\nabla\mathbf{v}\,\mathrm{d}x + \int_{\partial\Omega}(\mathbf{u}.\mathbf{n})\,(\mathbf{u}.\mathbf{v})\,\mathrm{d}s$$

The variational formulation is:
($\widetilde{FV}$): *find* $\mathbf{u}\in\mathbf{V}(\mathbf{g})$ *and* $p\in L^2(\Omega)$ *such that*

$$\begin{array}{rcccccl} Re\,\tilde{t}(\mathbf{u};\,\mathbf{u},\,\mathbf{v}) & + & a(\mathbf{u},\mathbf{v}) & + & b(\mathbf{v},p) & = & \tilde{l}(\mathbf{v}),\ \forall\mathbf{v}\in\mathbf{V}(0), \\ & & b(\mathbf{u},q) & & & = & 0,\ \forall q\in L^2(\Omega) \end{array}$$

where the forms $\tilde{t}$ and $\tilde{l}_h$ are given by:

$$\begin{aligned} \tilde{t}(\mathbf{w};\,\mathbf{u},\,\mathbf{v}) &= -\int_\Omega (\mathbf{w}\otimes\mathbf{u}):\nabla\mathbf{v}\,\mathrm{d}x \\ \tilde{l}(\mathbf{v}) &= l(\mathbf{v}) - Re\int_{\partial\Omega}(\mathbf{g}.\mathbf{n})\,(\mathbf{g}.\mathbf{v})\,\mathrm{d}s \end{aligned}$$

Note that the right-hand side $\tilde{l}$ contains an additional term that compensates those coming from the integration by parts. Then, with $\mathbf{v}=\mathbf{u}$:

$$\begin{aligned} \tilde{t}(\mathbf{w};\,\mathbf{u},\,\mathbf{u}) &= -\int_\Omega (\mathbf{w}\otimes\mathbf{u}):\nabla\mathbf{u}\,\mathrm{d}x \\ &= \int_\Omega \mathbf{div}(\mathbf{w}\otimes\mathbf{u}).\mathbf{u}\,\mathrm{d}x - \int_{\partial\Omega}(\mathbf{w}\otimes\mathbf{u}):(\mathbf{u}\otimes\mathbf{n})\,\mathrm{d}x \\ &= \int_\Omega \left(((\mathbf{u}.\nabla)\mathbf{w}).\mathbf{u} + \mathrm{div}(\mathbf{u})\,(\mathbf{u}.\mathbf{w})\right)\,\mathrm{d}x - \int_{\partial\Omega}(\mathbf{u}.\mathbf{n})\,(\mathbf{u}.\mathbf{w})\,\mathrm{d}x \end{aligned}$$

From an integration by part similar to (4.21):

$$\int_\Omega (\mathbf{u}.\nabla\mathbf{w}).\mathbf{u}\,\mathrm{d}x = -\int_\Omega (\mathbf{u}.\nabla\mathbf{u}).\mathbf{w}\,\mathrm{d}x - \int_\Omega \mathrm{div}(\mathbf{u})\,(\mathbf{u}.\mathbf{w})\,\mathrm{d}x + \int_{\partial\Omega}(\mathbf{u}.\mathbf{n})\,(\mathbf{u}.\mathbf{w})\,\mathrm{d}s$$

The term $(\mathbf{u}.\nabla\mathbf{w}).\mathbf{u}$ do not reapper after the integration by parts: instead, it appears $(\mathbf{u}.\nabla\mathbf{u}).\mathbf{w}$. Thus, the structure of the $\tilde{t}$ trilinear form do not permit a general skew-symmetry property as it was the case for t. It requires the three arguments to be the same:

$$\tilde{t}(\mathbf{u};\,\mathbf{u},\,\mathbf{u}) = \int_\Omega \left(((\mathbf{u}.\nabla)\mathbf{u}).\mathbf{u} + \mathrm{div}(\mathbf{u})\,|\mathbf{u}|^2\right)\,\mathrm{d}x - \int_{\partial\Omega}(\mathbf{u}.\mathbf{n})\,|\mathbf{u}|^2\,\mathrm{d}x$$

Using (4.21) with $\mathbf{w}=\mathbf{u}$ leads to:

$$\int_\Omega ((\mathbf{u}.\nabla)\mathbf{u}).\mathbf{u}\,\mathrm{d}x = -\frac{1}{2}\int_\Omega \mathrm{div}(\mathbf{u})\,|\mathbf{u}|^2\,\mathrm{d}x + \frac{1}{2}\int_{\partial\Omega}(\mathbf{u}.\mathbf{n})\,|\mathbf{u}|^2\,\mathrm{d}s \tag{4.26}$$

Then

$$\tilde{t}(\mathbf{u};\,\mathbf{u},\,\mathbf{u}) = \frac{1}{2}\int_\Omega \mathrm{div}(\mathbf{u})\,|\mathbf{u}|^2\,\mathrm{d}x - \frac{1}{2}\int_{\partial\Omega}(\mathbf{u}.\mathbf{n})\,|\mathbf{u}|^2\,\mathrm{d}s$$

When working with velocities that are not divergence-free, a possible modification of the trilinear form $\tilde{t}$ is to consider

$$\begin{aligned} \tilde{t}^*(\mathbf{w};\,\mathbf{u},\,\mathbf{v}) &= \tilde{t}(\mathbf{w};\,\mathbf{u},\,\mathbf{v}) - \frac{1}{2}\int_\Omega \mathrm{div}(\mathbf{v})\,(\mathbf{u}.\mathbf{w})\,\mathrm{d}x + \frac{1}{2}\int_{\partial\Omega}(\mathbf{v}.\mathbf{n})\,(\mathbf{u}.\mathbf{w})\,\mathrm{d}s \\ &= -\int_\Omega \left((\mathbf{w}\otimes\mathbf{u}):D(\mathbf{v}) + \frac{1}{2}\mathrm{div}(\mathbf{v})\,(\mathbf{u}.\mathbf{w})\right)\,\mathrm{d}x + \frac{1}{2}\int_{\partial\Omega}(\mathbf{v}.\mathbf{n})\,(\mathbf{u}.\mathbf{w})\,\mathrm{d}s \end{aligned}$$

Then we have

$$\tilde{t}^*(\mathbf{u};\,\mathbf{u},\,\mathbf{u}) \quad = \quad 0,\ \forall \mathbf{u} \in H^1(\Omega)^d$$

The new variational formulation is:
$(\widetilde{FV})^*$: *find* $\mathbf{u} \in \mathbf{V}(\mathbf{g})$ *and* $\tilde{p} \in L^2(\Omega)$ *such that*

$$\begin{array}{rcccccl} Re\,\tilde{t}^*(\mathbf{u};\,\mathbf{u},\,\mathbf{v}) & + & a(\mathbf{u},\mathbf{v}) & + & b(\mathbf{v},\tilde{p}) & = & \tilde{l}(\mathbf{v}),\ \forall \mathbf{v} \in \mathbf{V}(0), \\ & & b(\mathbf{u},q) & & & = & 0,\ \forall q \in L^2(\Omega) \end{array}$$

One can easily check that when $(\mathbf{u},\tilde{p})$ is a solution of $(\widetilde{FV})^*$, then $(\mathbf{u},p)$ is a solution of $(\widetilde{FV})$ with $p = \tilde{p} + Re|\mathbf{u}|/2$. The apparition of the kinetic energy term $Re|\mathbf{u}|/2$ in the modified pressure field $\tilde{p}$ is due to the introduction of the $\mathrm{div}(\mathbf{v})\,(\mathbf{u}.\mathbf{w})$ term in the trilinear form $\tilde{t}^*$.
At the discrete level, let us define for all $\mathbf{u}_h, \mathbf{v}_h, \mathbf{w}_h \in \mathbf{X}_h$:

$$\begin{aligned} \tilde{t}_h^*(\mathbf{w}_h;\,\mathbf{u}_h,\,\mathbf{v}_h) \quad = \quad & -\int_\Omega \left((\mathbf{w}_h \otimes \mathbf{u}_h) : \nabla_h \mathbf{v}_h + \frac{1}{2}\mathrm{div}_h(\mathbf{v}_h)\,(\mathbf{u}_h.\mathbf{w}_h) \right) \mathrm{d}x \\ & + \frac{1}{2}\int_{\partial\Omega} (\mathbf{v}_h.\mathbf{n})\,(\mathbf{u}_h.\mathbf{w}_h)\,\mathrm{d}s \end{aligned}$$

Note that $\tilde{t}_h^*$ is similar to $\tilde{t}^*$: the gradient and divergence has been replaced by their broken counterpart in the first term. As $\mathbf{X}_h \not\subset H^1(\Omega)^d$, the skew-symmetry property is not expected to be true at the discrete level. Then

$$\tilde{t}_h^*(\mathbf{u}_h;\,\mathbf{u}_h,\,\mathbf{u}_h) \quad = \quad -\int_\Omega \left((\mathbf{u}_h \otimes \mathbf{u}_h) : \nabla_h \mathbf{u}_h + \frac{1}{2}\mathrm{div}_h(\mathbf{u}_h)\,|\mathbf{u}_h|^2 \right) \mathrm{d}x + \frac{1}{2}\int_{\partial\Omega} (\mathbf{u}_h.\mathbf{n})\,|\mathbf{u}_h|^2\,\mathrm{d}s$$

Next, using (4.26) in each K, and then developing thanks to (4.22)-(4.23), we get

$$\begin{aligned} \tilde{t}_h^*(\mathbf{u}_h;\,\mathbf{u}_h,\,\mathbf{u}_h) \quad &= \quad \frac{1}{2}\int_{\partial\Omega} (\mathbf{u}_h.\mathbf{n})\,|\mathbf{u}_h|^2\,\mathrm{d}s - \frac{1}{2}\sum_{K\in\mathscr{T}_h}\int_{\partial K} (\mathbf{u}_h.\mathbf{n})\,|\mathbf{u}_h|^2\,\mathrm{d}s \\ &= \quad -\frac{1}{2}\sum_{S\in\mathscr{S}_h^{(i)}}\int_S [\![(\mathbf{u}_h.\mathbf{n})\,|\mathbf{u}_h|^2]\!]\,\mathrm{d}s \\ &= \quad -\frac{1}{2}\sum_{S\in\mathscr{S}_h^{(i)}}\int_S \left((\{\!\{\mathbf{u}_h\}\!\}.\mathbf{n})\,[\![|\mathbf{u}_h|^2]\!] + ([\![\mathbf{u}_h]\!].\mathbf{n})\,\{\!\{|\mathbf{u}_h|^2\}\!\} \right) \mathrm{d}s \\ &= \quad -\sum_{S\in\mathscr{S}_h^{(i)}}\int_S \left((\{\!\{\mathbf{u}_h\}\!\}.\mathbf{n})\,(\{\!\{\mathbf{u}_h\}\!\}.[\![\mathbf{u}_h]\!]) + \frac{1}{2}([\![\mathbf{u}_h]\!].\mathbf{n})\,\{\!\{|\mathbf{u}_h|^2\}\!\} \right) \mathrm{d}s \end{aligned}$$

The idea is to integrate this term in the definition of a discrete $\tilde{t}_h$. One of the possibilities is

$$\begin{aligned} \tilde{t}_h(\mathbf{w}_h;\,\mathbf{u}_h,\,\mathbf{v}_h) \quad &= \quad \tilde{t}_h^*(\mathbf{w}_h;\,\mathbf{u}_h,\,\mathbf{v}_h) + \sum_{S\in\mathscr{S}_h^{(i)}}\int_S \left((\{\!\{\mathbf{u}_h\}\!\}.\mathbf{n})\,(\{\!\{\mathbf{w}_h\}\!\}.[\![\mathbf{v}_h]\!]) + \frac{1}{2}\{\!\{\mathbf{u}_h.\mathbf{w}_h\}\!\}\,([\![\mathbf{v}_h]\!].\mathbf{n}) \right) \mathrm{d}s \\ &= \quad -\int_\Omega \left((\mathbf{w}_h \otimes \mathbf{u}_h) : \nabla_h \mathbf{v}_h + \frac{1}{2}\mathrm{div}_h(\mathbf{v}_h)\,(\mathbf{u}_h.\mathbf{w}_h) \right) \mathrm{d}x \\ &\quad + \frac{1}{2}\int_{\partial\Omega} (\mathbf{v}_h.\mathbf{n})\,(\mathbf{u}_h.\mathbf{w}_h)\,\mathrm{d}s \\ &\quad + \sum_{S\in\mathscr{S}_h^{(i)}}\int_S \left((\{\!\{\mathbf{u}_h\}\!\}.\mathbf{n})\,(\{\!\{\mathbf{w}_h\}\!\}.[\![\mathbf{v}_h]\!]) + \frac{1}{2}\{\!\{\mathbf{u}_h.\mathbf{w}_h\}\!\}\,([\![\mathbf{v}_h]\!].\mathbf{n}) \right) \mathrm{d}s \end{aligned} \tag{4.27}$$

This expression was proposed by di Pietro and Ern [2010, p. 21], eqn (73) (see also di Pietro and Ern, 2012, p. 282) following and original idea introduced in Cockburn et al. [2005].

File 4.21: `inertia_cks.icc`

```
form inertia (field w, trial u, test v,
  integrate_option iopt = integrate_option())
{
  return
    integrate (- dot(trans(grad_h(v))*w,u) - 0.5*div_h(v)*dot(u,w), iopt)
  + integrate ("internal_sides",
            dot(average(u),normal())*dot(jump(v),average(w))
     + 0.5*dot(jump(v),normal())
            *(dot(average(u),average(w)) + 0.25*dot(jump(u),jump(w))), iopt)
  + integrate ("boundary", 0.5*dot(v,normal())*dot(u,w), iopt);
}
field inertia_fix_rhs (test v,
  integrate_option iopt = integrate_option())
{
  return integrate("boundary", -dot(g(),normal())*dot(g(),v), iopt);
}
```

The discrete problem is

$(\widetilde{FV})_h$: *find* $\mathbf{u}_h \in \mathbf{X}_h$ *and* $\tilde{p} \in Q_h$ *such that*

$$\begin{array}{rcccccl} Re\,\tilde{t}_h(\mathbf{u}_h;\,\mathbf{u}_h,\,\mathbf{v}_h) & + & a_h(\mathbf{u}_h,\mathbf{v}_h) & + & b_h(\mathbf{v}_h,\tilde{p}_h) & = & \tilde{l}_h^*(\mathbf{v}_h),\ \forall \mathbf{v}_h \in \mathbf{X}_h, \\ & & b_h(\mathbf{u}_h,q_h) & - & c_h(p_h,q_h) & = & k_h(q),\ \forall q_h \in Q_h \end{array}$$

A simple test program is obtained by replacing in `navier_stokes_taylor_dg.cc` the include `inertia.h` by `inertia_cks.icc`. The compilation and run are similar.

4.5.4 Newton solver

The discrete problems $(FV)_h$ can be put in a compact form:

$$F(\mathbf{u}_h, p_h) = 0$$

where F is defined in variational form:

$$\langle F(\mathbf{u}_h,p_h),\,(\mathbf{v}_h,q_h)\rangle = \left(\begin{array}{rcccccl} Re\,t_h(\mathbf{u}_h;\,\mathbf{u}_h,\,\mathbf{v}_h) & + & a_h(\mathbf{u}_h,\mathbf{v}_h) & + & b_h(\mathbf{v}_h,p_h) & - & l_h^*(\mathbf{v}_h) \\ & & b_h(\mathbf{u}_h,q_h) & - & c_h(p_h,q_h) & - & k_h(q) \end{array} \right)$$

for all $(\mathbf{v}_h, q_h) \in \mathbf{X}_h \times Q_h$. Note that, after some minor modifications in the definition of F, this method could also applies for the locally conservative formulation $(\widetilde{FV})_h$. The previous formulation is simply the variational expression of $F(\mathbf{u}_h, p_h) = 0$. The Newton method defines the sequence $\left(\mathbf{u}_h^{(k)}\right)_{k\geqslant 0}$ by recurrence as:

- $k = 0$: let $\mathbf{u}_h^{(0)} \in \mathbf{X}_h$ being known.
- $k \geqslant 0$: let $\mathbf{u}_h^{(k-1)} \in \mathbf{X}_h$ given. Find $\delta\mathbf{u}_h \in \mathbf{X}_h$ and $\delta p_h \in Q_h$ such that

$$F'\left(\mathbf{u}_h^{(k-1)},\,p_h^{(k-1)}\right).(\delta\mathbf{u}_h,\delta p_h) = -F\left(\mathbf{u}_h^{(k-1)},p_h^{(k-1)}\right)$$

  and then defines

$$\mathbf{u}_h^{(k)} = \mathbf{u}_h^{(k-1)} + \delta\mathbf{u}_h \quad \text{and} \quad p_h^{(k)} = p_h^{(k-1)} + \delta p_h$$

At each step $k \geqslant 0$, this algorithm involves a linear subproblem involving the Jacobean F' that is defined by its variational form:

$$\begin{array}{l} \langle F'\left(\mathbf{u}_h^{(k-1)},\,p_h^{(k-1)}\right).(\delta\mathbf{u}_h,\delta p_h),\ (\mathbf{v}_h,q_h)\rangle \\ = \left(\begin{array}{rcccccl} Re\,\left(t_h(\delta\mathbf{u}_h;\,\mathbf{u}_h,\,\mathbf{v}_h) + t_h(\mathbf{u}_h;\,\delta\mathbf{u}_h,\,\mathbf{v}_h)\right) & + & a_h(\delta\mathbf{u}_h,\mathbf{v}_h) & + & b_h(\mathbf{v}_h,\delta p_h) \\ & & b_h(\delta\mathbf{u}_h,q_h) & - & c_h(\delta p_h,q_h) \end{array} \right) \end{array}$$

File 4.22: navier_stokes_taylor_newton_dg.cc

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
#include "taylor.h"
#include "stokes_dirichlet_dg.icc"
#include "inertia.h"
#include "navier_stokes_dg.h"
int main(int argc, char**argv) {
  environment rheolef (argc, argv);
  Float eps = numeric_limits<Float>::epsilon();
  geo omega (argv[1]);
  string approx   = (argc > 2) ?       argv[2]   : "P1d";
  Float  Re       = (argc > 3) ? atof(argv[3]) : 100;
  Float  tol      = (argc > 4) ? atof(argv[4]) : eps;
  size_t max_iter = (argc > 5) ? atoi(argv[5]) : 100;
  string restart  = (argc > 6) ?       argv[6]   : "";
  navier_stokes_dg F (Re, omega, approx);
  navier_stokes_dg::value_type xh = F.initial (restart);
  int status = damped_newton (F, xh, tol, max_iter, &derr);
  dout << catchmark("Re") << Re << endl
       << catchmark("u") << xh[0]
       << catchmark("p") << xh[1];
  return status;
}
```

Comments

The implementation of the Newton method follows the generic approach introduced in volume 1, section 3.2.3, page 119. For that purpose we define a class navier_stokes_dg.

File 4.23: navier_stokes_dg.h

```
struct navier_stokes_dg {
  typedef Float                    float_type;
  typedef Eigen::Matrix<field,2,1> value_type;
  navier_stokes_dg (Float Re, const geo& omega, string approx);
  value_type initial (string restart) const;
  value_type residue     (const value_type& uh) const;
  void update_derivative (const value_type& uh) const;
  value_type derivative_solve      (const value_type& mrh) const;
  value_type derivative_trans_mult (const value_type& mrh) const;
  Float space_norm       (const value_type& uh) const;
  Float dual_space_norm  (const value_type& mrh) const;
  Float Re;
  space Xh, Qh;
  integrate_option iopt;
  form  a0, b, c, mu, mp;
  field lh0, lh, kh;
  problem pmu, pmp;
  mutable form a1;
  mutable problem_mixed stokes1;
};
#include "navier_stokes_dg1.icc"
#include "navier_stokes_dg2.icc"
```

The member functions of the class are defined in two separate files.

File 4.24: navier_stokes_dg1.icc

```
navier_stokes_dg::navier_stokes_dg (
 Float Re1, const geo& omega, string approx)
 : Re(Re1), Xh(), Qh(), iopt(), a0(), b(), c(), mu(), mp(), lh0(), lh(), kh(),
   pmu(), pmp(), a1(), stokes1()
{
  Xh = space (omega, approx, "vector");
  Qh = space (omega, approx);
  iopt.set_family(integrate_option::gauss);
  iopt.set_order(2*Xh.degree()+1);
  stokes_dirichlet_dg (Xh, Qh, a0, b, c, mp, lh0, kh, iopt);
  trial u (Xh); test v (Xh);
  lh = lh0 + Re*inertia_fix_rhs (v, iopt);
  mu = integrate (dot(u,v), iopt);
  pmu = problem (mu);
  pmp = problem (mp);
}
navier_stokes_dg::value_type
navier_stokes_dg::initial (string restart) const {
  value_type xh = { field(Xh, 0), field(Qh, 0) };
  Float Re0 = 0;
  if (restart == "") {
    problem_mixed stokes0 (a0, b, c);
    stokes0.set_metric (mp);
    stokes0.solve (lh0, kh, xh[0], xh[1]);
  } else {
    idiststream in (restart);
    in >> catchmark("Re") >> Re0
       >> catchmark("u")  >> xh[0]
       >> catchmark("p")  >> xh[1];
    check_macro (xh[1].get_space() == Qh, "unexpected "
       << xh[0].get_space().name() << " approximation in file \""
       << restart << "\" (" << Xh.name() << " expected)");
  }
  derr << "# continuation: from Re=" << Re0 << " to " << Re << endl;
  return xh;
}
navier_stokes_dg::value_type
navier_stokes_dg::residue (const value_type& xh) const {
  trial u (Xh); test v (Xh);
  form a = a0 + Re*inertia(xh[0], u, v, iopt);
  value_type mrh = { a*xh[0] + b.trans_mult(xh[1]) - lh,
                     b*xh[0] -            c*xh[1]  - kh};
  return mrh;
}
void navier_stokes_dg::update_derivative (const value_type& xh) const {
  trial u (Xh); test v (Xh);
  a1 = a0 + Re*(inertia(xh[0], u, v, iopt) + inertia(u, xh[0], v, iopt));
  stokes1 = problem_mixed (a1, b, c);
  stokes1.set_metric (mp);
}
navier_stokes_dg::value_type
navier_stokes_dg::derivative_solve (const value_type& mrh) const {
  value_type delta_xh = { field(Xh, 0), field(Qh, 0) };
  stokes1.solve (mrh[0], mrh[1], delta_xh[0], delta_xh[1]);
  return delta_xh;
}
navier_stokes_dg::value_type
navier_stokes_dg::derivative_trans_mult (const value_type& mrh) const {
  value_type rh = { field (Xh), field (Qh) };
  pmu.solve (mrh[0], rh[0]);
  pmp.solve (mrh[1], rh[1]);
  value_type mgh = { a1.trans_mult(rh[0]) + b.trans_mult(rh[1]),
                     b*rh[0]  -            c*rh[1]             };
  return mgh;
}
```

File 4.25: navier_stokes_dg2.icc

```
Float navier_stokes_dg::space_norm (const value_type& xh) const {
  return sqrt (mu(xh[0],xh[0]) + mp(xh[1],xh[1]));
}
Float navier_stokes_dg::dual_space_norm (const value_type& mrh) const {
  value_type rh = { field(Xh,0), field (Qh,0) };
  pmu.solve (mrh[0], rh[0]);
  pmp.solve (mrh[1], rh[1]);
  return sqrt (dual(rh[0],mrh[0]) + dual(rh[1],mrh[1]));
}
```

```
make navier_stokes_taylor_newton_dg navier_stokes_taylor_error_dg
./navier_stokes_taylor_newton_dg square P2d 1000 | ./navier_stokes_taylor_error_dg
```

4.5.5 Application to the driven cavity benchmark

File 4.26: cavity_dg.h

```
struct g {
  point operator() (const point& x) const {
    return point((abs(1-x[1]) < 1e-7) ? 1 : 0, 0, 0); }
};
struct f {
  point operator() (const point& x) const { return point(0,0,0); }
};
```

The program navier_stokes_cavity_newton_dg.cc is obtained by replacing in navier_stokes_taylor_newton_dg.cc the include taylor.h by cavity_dg.h that defines the boundary conditions. The compilation and run are similar.

```
make navier_stokes_cavity_newton_dg streamf_cavity
./navier_stokes_cavity_newton_dg  square P1d 500 > square.field
field square.field -proj -field | ./streamf_cavity | \
                  field -bw -n-iso-negative 10 -
```

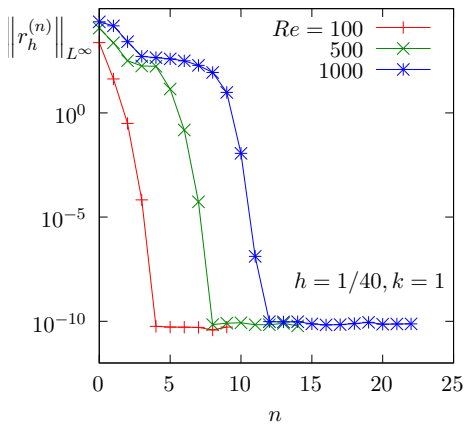


Figure 4.16: The discontinuous Galerkin method for the Navier-Stokes problem on the driven cavity benchmark when $k = 1$ and $d = 2$: convergence of the damped Newton algorithm.

4.5.6 Upwinding

The skew symmetry property is generalized to the requirement that t_h be non-dissipative (see di Pietro and Ern, 2012, p. 282, eqn (6.68)):

$$t_h(\mathbf{w}_h;\, \mathbf{u}_h,\, \mathbf{u}_h) \geqslant 0,\ \forall \mathbf{w}_h,\, \mathbf{u}_h \in \mathbf{X}_h$$

A way to satisfy this property is to add an *upwinding* term in t_h:

$$\breve{t}_h(\mathbf{w}_h;\, \mathbf{u}_h,\, \mathbf{v}_h) := t_h(\mathbf{w}_h;\, \mathbf{u}_h,\, \mathbf{v}_h) + s_h(\mathbf{w}_h;\, \mathbf{u}_h,\, \mathbf{v}_h)$$

with

$$s_h(\mathbf{w}_h;\, \mathbf{u}_h,\, \mathbf{v}_h) = \frac{1}{2} \sum_{S \in \mathscr{S}_h^{(i)}} \int_S |\{\!\{\mathbf{w}_h\}\!\}.\mathbf{n}|\ ([\![\mathbf{u}_h]\!].[\![\mathbf{v}_h]\!])\, \mathrm{d}s$$

We aim at using a Newton method. We replace t_h by its extension $\breve{t}_h$ containing the upwind terms in the definition of F, and then we compute its Jacobean F'. As the absolute value is not differentiable, the functions s_h, $\breve{t}_h$ and then F are also not differentiable with respect to $\mathbf{w}_h$. Nevertheless, the absolute value is convex and we can use some concepts of the subdifferential calculus. Let us introduce the multi-valued sign function:

$$\mathrm{sgn}(x) = \begin{cases} \{1\} & \text{when } x > 0 \\ [-1,1] & \text{when } x = 0 \\ \{-1\} & \text{when } x < 0 \end{cases}$$

Then, the subdifferential of the absolute value function is $\mathrm{sgn}(x)$ and for all $\delta\mathbf{w}_h, \mathbf{w}_h,\, \mathbf{u}_h, \mathbf{v}_h \in \mathbf{X}_h$, we define a generalization of the partial derivative as

$$\frac{\partial s_h}{\partial \mathbf{w}_h}(\mathbf{w}_h;\, \mathbf{u}_h,\, \mathbf{v}_h).(\delta\mathbf{w}_h) = \frac{1}{2} \sum_{S \in \mathscr{S}_h^{(i)}} \int_S \mathrm{sgn}(\{\!\{\mathbf{w}_h\}\!\}.\mathbf{n})\ (\{\!\{\delta\mathbf{w}_h\}\!\}.\mathbf{n})\ ([\![\mathbf{u}_h]\!].[\![\mathbf{v}_h]\!])\, \mathrm{d}s$$

File 4.27: `inertia_upw.icc`

```
#include "sgn.icc"
form inertia_upw (field w, trial u, test v,
  integrate_option iopt = integrate_option())
{
  return integrate ("internal_sides",
     0.5*abs(dot(average(w),normal()))*dot(jump(u),jump(v)));
}
form d_inertia_upw (field w, trial dw, field u, test v,
  integrate_option iopt = integrate_option())
{
  return integrate ("internal_sides",
     0.5*compose (sgn, dot(average(w),normal()))
         *dot(average(dw),normal())*dot(jump(u),jump(v)));
}
```

A multi-valued Jacobean F' is then defined:

$$\begin{aligned} & \langle F'\left(\mathbf{u}_h^{(k-1)},\, p_h^{(k-1)}\right).(\delta\mathbf{u}_h, \delta p_h),\ (\mathbf{v}_h, q_h)\rangle \\ = \ & Re \begin{pmatrix} t_h(\delta\mathbf{u}_h;\, \mathbf{u}_h,\, \mathbf{v}_h) + t_h(\mathbf{u}_h;\, \delta\mathbf{u}_h,\, \mathbf{v}_h) + \dfrac{\partial s_h}{\partial \mathbf{w}_h}(\mathbf{u}_h;\, \mathbf{u}_h,\, \mathbf{v}_h).(\delta\mathbf{u}_h) + s_h(\mathbf{u}_h;\, \delta\mathbf{u}_h,\, \mathbf{v}_h) \\ \\ 0 \end{pmatrix} \\ & + \begin{pmatrix} a_h(\delta\mathbf{u}_h, \mathbf{v}_h) & + & b_h(\mathbf{v}_h, \delta p_h) \\ \\ b_h(\delta\mathbf{u}_h, q_h) & - & c_h(\delta p_h, q_h) \end{pmatrix} \end{aligned}$$

We are able to extend the Newton method to the F function that allows a multi-valued subdifferential F'. During iterations, we can choose any of the available directions in the subdifferential. One the possibilities is then to replace the multi-valued sign function by a single-value one:

$$\widetilde{\mathrm{sgn}}(x) = \begin{cases} 1 & \text{when } x \geqslant 0 \\ -1 & \text{when } x < 0 \end{cases}$$

File 4.28: `sgn.icc`

```
Float sgn (Float x) { return (x >= 0) ? 1 : -1; }
```

File 4.29: `navier_stokes_upw_dg.h`

```
#include "navier_stokes_dg.h"
struct navier_stokes_upw_dg: navier_stokes_dg {
  typedef Float                          float_type;
  typedef navier_stokes_dg::value_type value_type;
  navier_stokes_upw_dg (Float Re, const geo& omega, string approx);
  value_type residue     (const value_type& uh) const;
  void update_derivative (const value_type& uh) const;
};
#include "navier_stokes_upw_dg.icc"
```

File 4.30: `navier_stokes_upw_dg.icc`

```
#include "inertia_upw.icc"
navier_stokes_upw_dg::navier_stokes_upw_dg (
 Float Re1, const geo& omega, string approx)
 : navier_stokes_dg (Re1, omega, approx) {}

navier_stokes_upw_dg::value_type
navier_stokes_upw_dg::residue (const value_type& xh) const {
  trial u (Xh); test v (Xh);
  form a = a0 + Re*(  inertia     (xh[0], u, v, iopt)
                    + inertia_upw (xh[0], u, v, iopt));
  value_type mrh(2);
  mrh[0] = a*xh[0] + b.trans_mult(xh[1]) - lh;
  mrh[1] = b*xh[0] -             c*xh[1]  - kh;
  return mrh;
}
void navier_stokes_upw_dg::update_derivative (const value_type& xh) const {
  trial du (Xh); test v (Xh);
  a1 = a0 + Re*(      inertia          (xh[0], du,    v, iopt)
                + inertia_upw          (xh[0], du,    v, iopt)
                +    inertia           (du,    xh[0], v, iopt)
              + d_inertia_upw (xh[0], du,    xh[0], v, iopt));
  stokes1 = problem_mixed (a1, b, c);
  stokes1.set_metric (mp);
}
```

The program `navier_stokes_cavity_newton_upw_dg.cc` is obtained by replacing in `navier_stokes_taylor_newton_dg.cc` the string `navier_stokes_dg` by `navier_stokes_upw_dg` (two occurrences: in the includes and then in the definition of `F`). Also replace the include `taylor.h` by `cavity_dg.h` that defines the boundary conditions. The compilation and run are similar.

```
make navier_stokes_cavity_newton_upw_dg streamf_cavity
mkgeo_grid -t 80 > square.geo
./navier_stokes_cavity_newton_upw_dg  square P1d 500 1e-15 100 > square-500.field
field square-500.field -proj -field | ./streamf_cavity | \
                  field -bw -n-iso 30 -n-iso-negative 20 -
```

It could take about one minute of CPU, depending on your computer. Computations for higher Reynolds numbers are performed by continuation. Starting from a previous computation at $Re = 500$, we compute it at $Re = 1000$ as:

```
./navier_stokes_cavity_newton_upw_dg square P1d 1000 1e-15 100 square-500.field  \
    > square-1000.field
field square-1000.field -proj -field | ./streamf_cavity | \
    field -bw -n-iso 30 -n-iso-negative 20 -
```

Then, for $Re = 1500$:

```
./navier_stokes_cavity_newton_upw_dg square P1d 1500 1e-15 100 square-1000.field \
    > square-1500.field
field square-1500.field -proj -field | ./streamf_cavity | \
    field -bw -n-iso 30 -n-iso-negative 20 -
```

By this way, computations of solutions can be performed until $Re = 25\,000$ without problems. Note that, from $Re \approx 10\,000$, these solution are no more stable with respect to time, but are valid solution of the stationary Navier-Stokes problem and can be interpreted as timed averaged solutions. See also at the end of section 2.5 page 84 for a discussion about the loss of stationary of the solution. In conclusion, the discontinuous Galerkin method is much more robust and accurate than the characteristic method of section 2.5 when looking at stationary solutions of the Navier-Stokes equations. Moreover, this approach could also be easily exended to the non-stationary case by adding a time derivative term and a BDF time approximation, associated to a fully implicit approach.

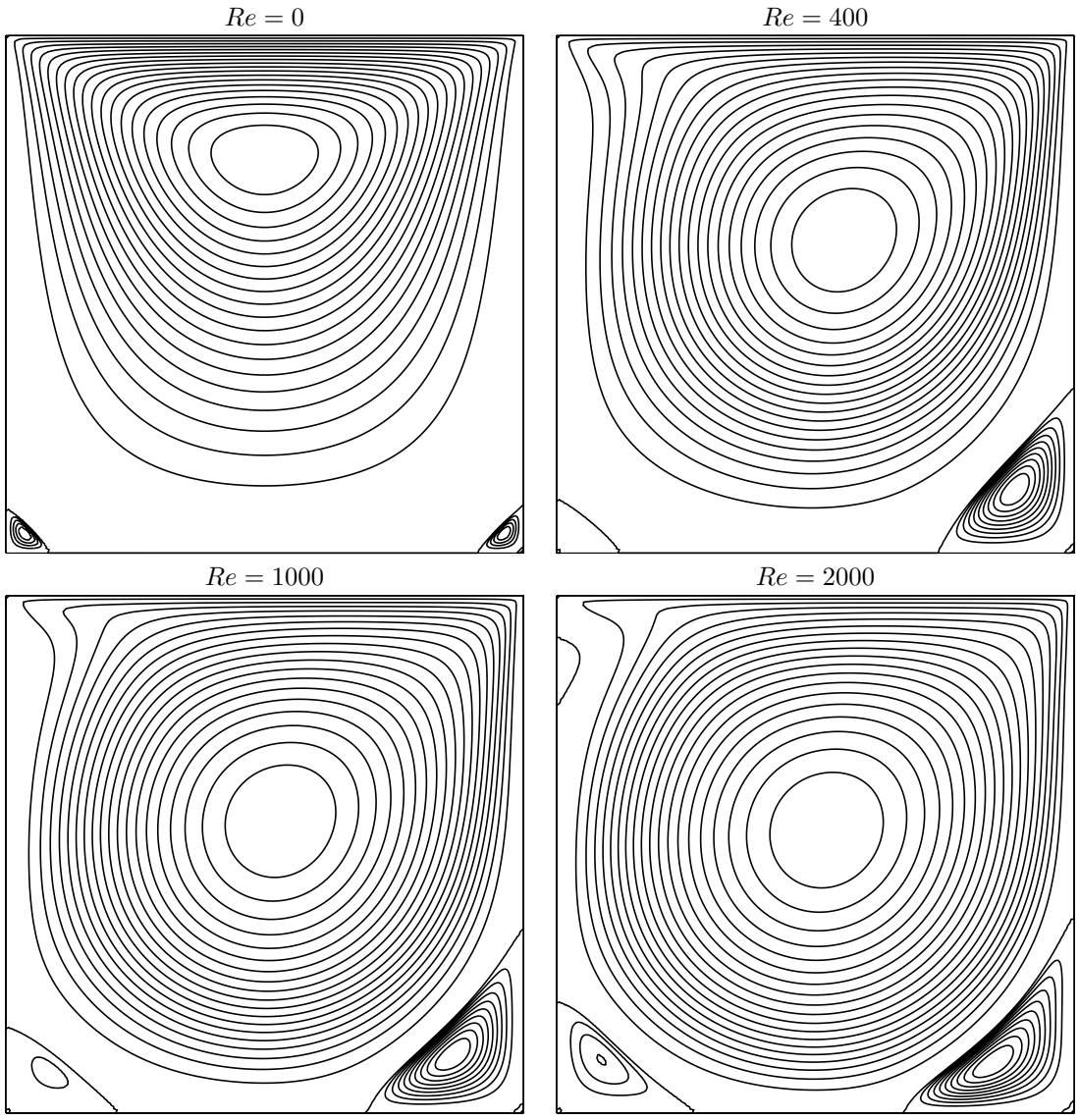


Figure 4.17: The discontinuous Galerkin method for the Navier-Stokes problem on the driven cavity benchmark when $k = 1$ (80×80 grid): stream function isovalues for various Re.

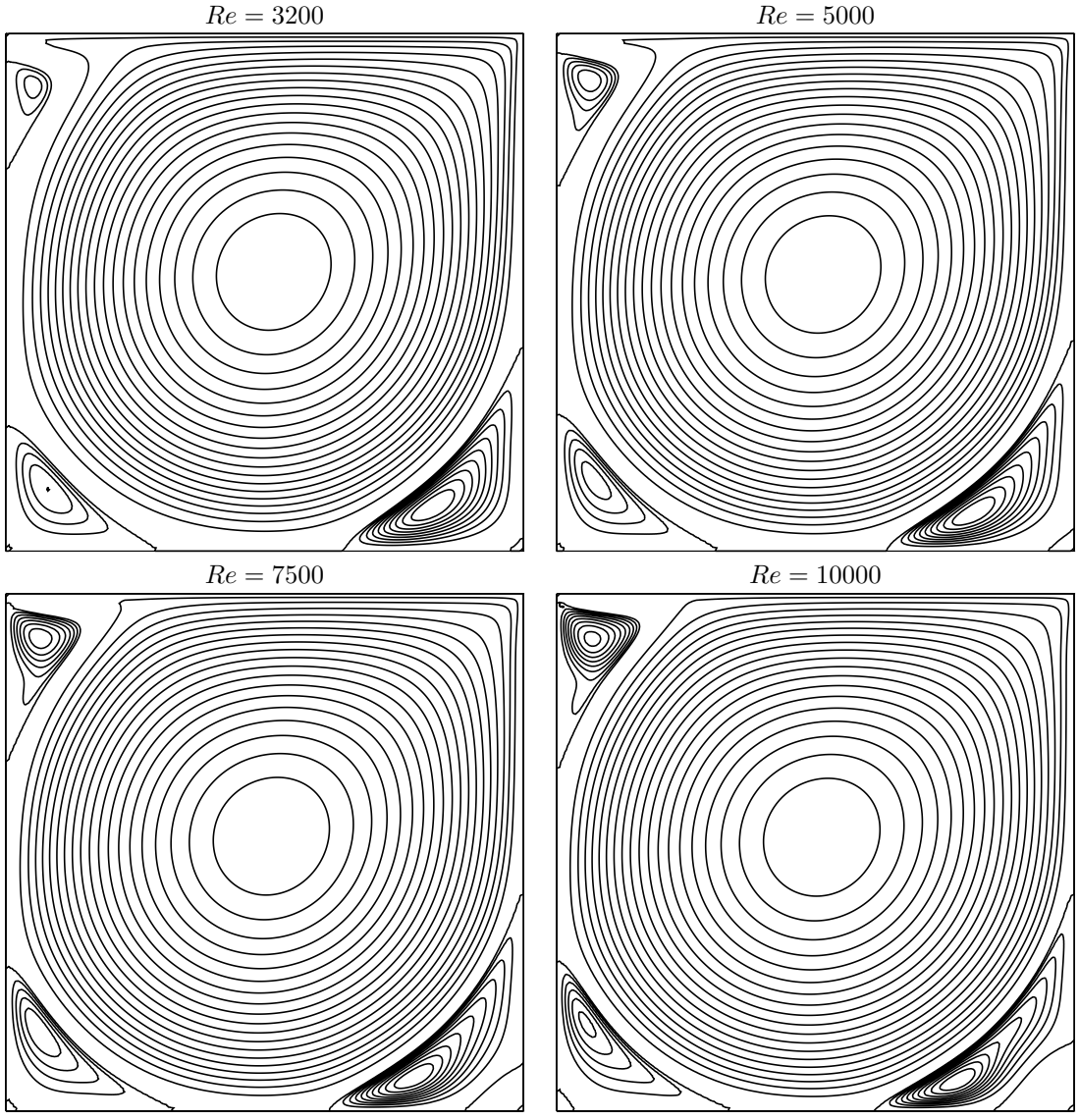


Figure 4.18: The discontinuous Galerkin method for the Navier-Stokes problem on the driven cavity benchmark when $k = 1$ (80×80 grid): stream function isovalues for various Re (cont.).

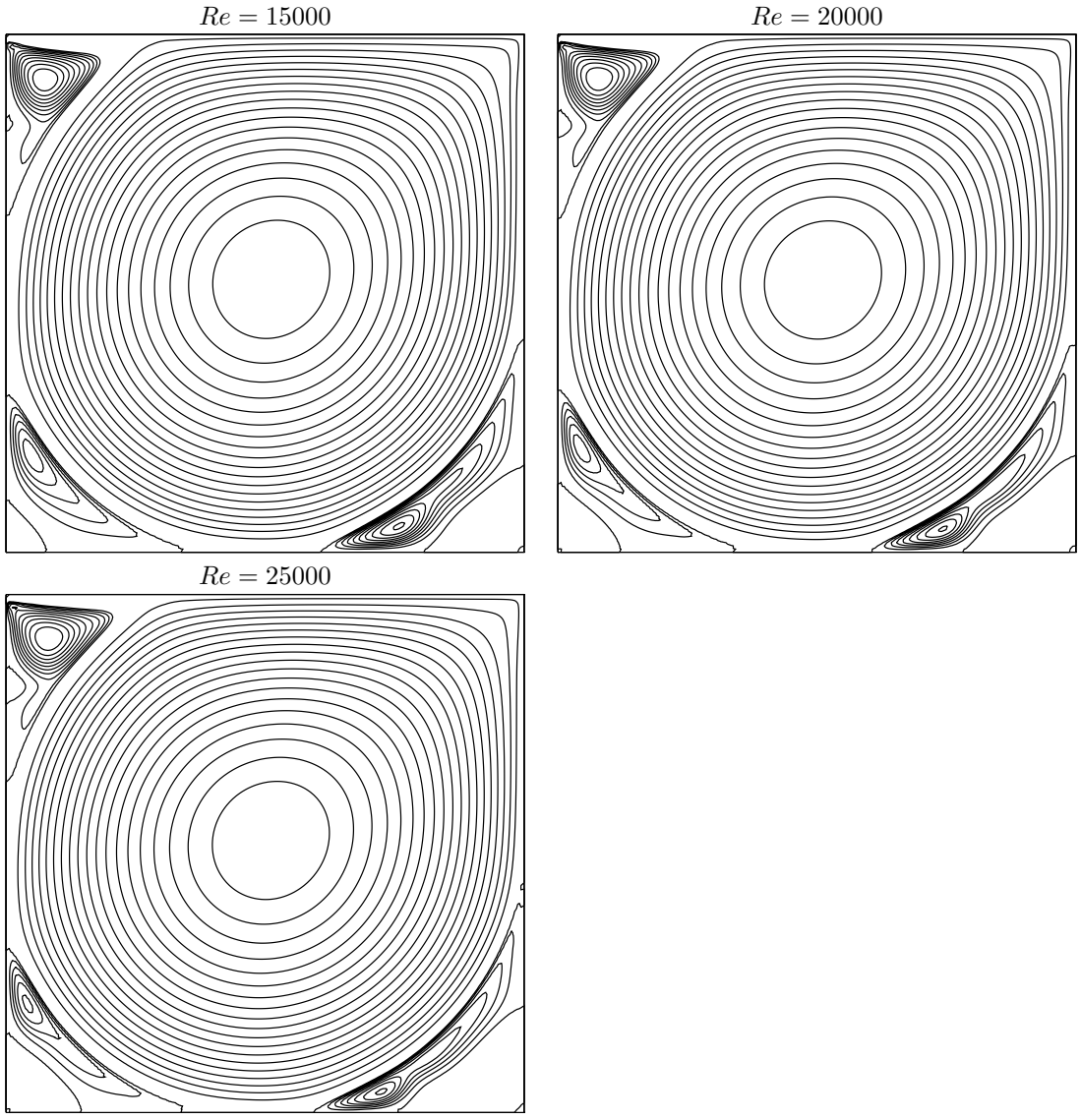


Figure 4.19: The discontinuous Galerkin method for the Navier-Stokes problem on the driven cavity benchmark when $k = 1$ (80×80 grid): stream function isovalues for various Re (cont.).

Chapter 5

Complex fluids

This part presents in details the practical computational aspects of numerical modeling with complex fluids. Most of the examples involve only few lines of code: the concision and readability of codes written with **Rheolef** is certainly a major key-point of this environment. The theoretical background for complex fluids an associated numerical methods can be found in the textbook by the present author [Saramito, 2016b]. We start with yield slip boundary condition as a preliminary problem. Slip at the wall occurs in many applications with complex fluids. This problem is solved both by augmented Lagrangian and Newton methods. Then, viscoplastic fluids are introduced and an augmented Lagrangian method is presented. A prelinary for viscoelastic fluids problems is the linear tensor transport equation: it is solved by a discontinuous Galerkin method. Finally, viscoelastic fluids problems are solved by an operator splitting algorithm, the θ-scheme.

5.1 Yield slip at the wall

5.1.1 Problem statement

The problem of a Newtonian fluid with yield slip at the wall and flowing in a pipe [Roquet and Saramito, 2004] writes:
(P): find u, defined in Ω, such that

$$-\Delta u = f \quad \text{in } \Omega \tag{5.1a}$$

$$\left.\begin{array}{ll} \left|\dfrac{\partial u}{\partial n}\right| \leqslant S & \text{when } u = 0 \\ \dfrac{\partial u}{\partial n} = -C_f |u|^{-1+n} u - S\dfrac{u}{|u|} & \text{otherwise} \end{array}\right\} \quad \text{on } \partial\Omega \tag{5.1b}$$

Here, $S \geqslant 0$ and $C_f > 0$ are respectively the yield slip and the friction coefficient while $n > 0$ is a power-law index. The computational domain Ω represents the cross-section of the pipe and u is the velocity component along the axis of the pipe. The right-hand side f is a given constant, and without loss of generality, we can suppose $f = 1$: the parameters are S, C_f and n. When $S = 0$ and $n = 1$, the problem reduces to a Poisson problem with homogeneous Robin boundary condition that depend upon C_f.

5.1.2 The augmented Lagrangian algorithm

Principe of the algorithm

The problem writes as a minimization one:

$$\min_{u\in H^1(\Omega)} J(u)$$

where

$$J(u) = \frac{C_f}{1+n}\int_{\partial\Omega} |u|^{1+n}\,\mathrm{d}s + S\int_{\partial\Omega}|u|\,\mathrm{d}s + \frac{1}{2}\int_\Omega |\boldsymbol{\nabla} u|^2\,\mathrm{d}x - \int_\Omega f\,u\,\mathrm{d}x$$

This problem is solved by using an augmented Lagrangian algorithm. The auxiliary variable $\gamma = u_{|\partial\Omega}$ is introduced together with the Lagrangian multiplier λ associated to the constraints $u_{|\partial\Omega} - \gamma = 0$. For all $r > 0$, let:

$$\begin{aligned} L((u,\gamma);\lambda) &= \frac{C_f}{1+n}\int_{\partial\Omega} |\gamma|^{1+n}\,\mathrm{d}s + S\int_{\partial\Omega}|\gamma|\,\mathrm{d}s + \frac{1}{2}\int_\Omega |\boldsymbol{\nabla} u|^2\,\mathrm{d}x - \int_\Omega f\,u\,\mathrm{d}x \\ &\quad + \int_{\partial\Omega}\lambda\,(u-\gamma)\,\mathrm{d}s + \frac{r}{2}\int_{\partial\Omega}|u-\gamma|^2\,\mathrm{d}s \end{aligned}$$

An Uzawa-like minimization algorithm writes:

- $k = 0$: let $\lambda^{(0)}$ and $\gamma^{(0)}$ arbitrarily chosen.
- $k \geqslant 0$: let $\lambda^{(k)}$ and $\gamma^{(k)}$ being known.

$$\begin{aligned} u^{(k+1)} &:= \arg\min_{v\in H^1(\Omega)} L((v,\gamma^{(k)});\lambda^{(k)}) \\ \gamma^{(k+1)} &:= \arg\min_{\delta\in L^\infty(\partial\Omega)} L((u^{(k+1)},\delta);\lambda^{(k)}) \\ \lambda^{(k+1)} &:= \lambda^{(k)} + \rho\left(u^{(k+1)} - \gamma^{(k+1)}\right) \quad \text{on } \partial\Omega \end{aligned}$$

The descent step ρ is chosen as $\rho = r$ for sufficiently large r. The Lagrangian L is quadratic in u and thus the computation of $u^{(k+1)}$ reduces to a linear problem. The non-linearity is treated when computing $\gamma^{(k+1)}$. This operation is performed point-by-point on $\partial\Omega$ by minimizing:

$$\gamma := \arg\min_{\delta\in\mathbb{R}} \frac{C_f|\delta|^{1+n}}{1+n} + \frac{r|\delta|^2}{2} + S|\delta| - \xi\,\delta$$

where $\xi = \lambda^{(k)} + r\,u^{(k+1)}$ is given. This problem is convex and its solution is unique. The solution has the form:

$$\gamma = P_{n,r}(\xi) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{when } |\xi| \leqslant S \\ \phi_{n,r}(|\xi| - S)\,\mathrm{sgn}(\xi) & \text{otherwise} \end{cases} \tag{5.2}$$

where $\phi_{n,r}(x) = f_{n,r}^{-1}(x)$ and $f_{n,r}$ is defined for all $y > 0$ by:

$$f_{n,r}(y) = C_f y^n + r\,y \tag{5.3}$$

File 5.1: `projection.h`

```
#include "phi.h"
// p(x) = phi(|x|-a)*sgn(x)
struct projection {
  Float operator() (const Float& x) const {
    if (fabs(x) <= a) return 0;
    return (x > 0) ? _phi(x-a) : -_phi(-x-a);
  }
  projection (Float a1, Float n=1, Float c=1, Float r=0)
   : a(a1), _phi(n,c,r) {}
  Float a;
  phi _phi;
};
```

Observe that $f'_{n,r}(y) = nC_f y^{-1+n} + r > 0$ when $r > 0$: the function is strictly increasing and is thus invertible. When $n = 1$ then $f_{n,r}$ is linear and $\phi_{1,r}(x) = x/(C_f + r)$. When $n = 1/2$ this problem reduces to a second order polynomial equation and the solution is also explicit:

$$\phi_{\frac{1}{2},r}(x) = \frac{1}{4}\left(\left\{1 + \frac{4r\,x}{C_f^2}\right\}^{1/2} - 1\right)^2$$

When $r = 0$, for any $n > 0$ the solution is $\phi_{n,0}(x) = (x/C_f)^{\frac{1}{n}}$. In general, when $n > 0$ and $r > 0$, the solution is no more explicit. We consider the Newton method:

- $i = 0$: Let y_0 being given.
- $i \geqslant 0$: Suppose y_i being known and compute $y_{i+1} = y_i - (f_{n,r}(y_i) - x)/f'_{n,r}(y_i)$.

File 5.2: `phi.h`

```
struct phi {
  phi (Float n1=2, Float c1=1, Float r1=0) : n(n1), c(c1), r(r1) {}
  Float operator() (const Float& x) const {
    if (x <= 0) return 0;
    if (n == 1) return x/(c+r);
    if (r == 0) return pow(x/c,1/n);
    Float y = x/(c+r);
    const Float tol = numeric_limits<Float>::epsilon();
    for (size_t i = 0; true; ++i) {
      Float ry = f(y)-x;
      Float dy = -ry/df_dy(y);
      if (fabs(ry) <= tol && fabs(dy) <= tol) break;
      if (i >= max_iter) break;
      if (y+dy > 0) {
        y += dy;
      } else {
        y /= 2;
        check_macro (1+y != y, "phi: machine precision problem");
      }
    }
    return y;
  }
  Float derivative (const Float& x) const {
    Float phi_x = operator()(x);
    return 1/(r + n*c*pow(phi_x,-1+n));
  }
protected:
  Float f(Float y) const     { return c*pow(y,n) + r*y; }
  Float df_dy(Float y) const { return n*c*pow(y,-1+n) + r; }
  Float n,c,r;
  static const size_t max_iter = 100;
};
```

In the present implementation, in order to avoid too large steps, the Newton step is damped when y_{i+1} becomes negative.

The Uzawa algorithm writes:

- $k = 0$: let $\lambda^{(0)}$ and $\gamma^{(0)}$ arbitrarily chosen.
- $k \geqslant 0$: let $\lambda^{(k)}$ and $\gamma^{(k)}$ being known, find $u^{(k+1)}$, defined in Ω, such that

$$\begin{aligned} -\Delta u^{(k+1)} &= f \ \text{ in } \Omega \\ \frac{\partial u^{(k+1)}}{\partial n} + r\,u^{(k+1)} &= r\,\gamma^{(k)} - \lambda^{(k)} \ \text{ on } \partial\Omega \end{aligned}$$

and then compute explicitely $\gamma^{(k+1)}$ and $\lambda^{(k+1)}$:

$$\begin{aligned}\gamma^{(k+1)} &:= P_{n,r}\left(\lambda^{(k)} + r\,u^{(k+1)}\right) \quad \text{on } \partial\Omega \\ \lambda^{(k+1)} &:= \lambda^{(k)} + r\left(u^{(k+1)} - \gamma^{(k+1)}\right) \quad \text{on } \partial\Omega\end{aligned}$$

This algorithm reduces the nonlinear problem to a sequence of linear and completely standard Poisson problems with a Robin boundary condition and some explicit computations. At convergence, $\lambda = -\dfrac{\partial u}{\partial n}$ and $\gamma = u$ on $\partial\Omega$.
Note that the solution satisfies the following variational formulation:

$$\begin{aligned}\int_\Omega \boldsymbol{\nabla} u.\boldsymbol{\nabla} v\,\mathrm{d}x + \int_{\partial\Omega} v\,\lambda\,\mathrm{d}s &= \int_\Omega f\,v\,\mathrm{d}x, \quad \forall v \in H^1(\Omega) \\ \int_{\partial\Omega} u\,\gamma\,\mathrm{d}s - \int_{\partial\Omega} P_{n,0}(\lambda)\,\gamma\,\mathrm{d}s &= 0, \quad \forall \gamma \in L^\infty(\partial\Omega)\end{aligned}$$

This formulation is the base of the computation of the residual test used for the stopping criteria.

File 5.3: `yield_slip_augmented_lagrangian.cc`

```
#include "rheolef.h"
using namespace std;
using namespace rheolef;
#include "yield_slip_augmented_lagrangian.icc"
#include "poisson_robin.icc"
int main(int argc, char**argv) {
  environment rheolef (argc,argv);
  dlog << noverbose;
  geo omega (argv[1]);
  string approx   = (argc > 2) ?       argv[2]   : "P1";
  Float  S        = (argc > 3) ? atof(argv[3]) : 0.6;
  Float  n        = (argc > 4) ? atof(argv[4]) : 1;
  Float  Cf       = (argc > 5) ? atof(argv[5]) : 1;
  Float  r        = (argc > 6) ? atof(argv[6]) : 1;
  Float  tol      = 1e3*numeric_limits<Float>::epsilon();
  size_t max_iter = 100000;
  space Xh (omega, approx);
  test  v (Xh);
  field lh = integrate(v);
  field uh = poisson_robin (Cf, omega["boundary"], lh);
  space Wh (omega["boundary"], Xh.get_approx());
  field lambda_h = Cf*uh["boundary"];
  int status = yield_slip_augmented_lagrangian(S, n, Cf, omega["boundary"],
                lh, lambda_h, uh, tol, max_iter, r);
  dout << setprecision(numeric_limits<Float>::digits10)
       << catchmark("S")  << S  << endl
       << catchmark("n")  << n  << endl
       << catchmark("Cf") << Cf << endl
       << catchmark("r")  << r  << endl
       << catchmark("u")  << uh
       << catchmark("lambda") << lambda_h;
  return status;
}
```

File 5.4: `poisson_robin.icc`

```
field poisson_robin (Float Cf, const geo& boundary, const field& lh) {
  const space& Xh = lh.get_space();
  trial u (Xh); test  v (Xh);
  form a = integrate(dot(grad(u),grad(v))) + Cf*integrate(boundary, u*v);
  field uh (Xh);
  problem p (a);
  p.solve (lh, uh);
  return uh;
}
```

File 5.5: yield_slip_augmented_lagrangian.icc

```
#include "projection.h"
int yield_slip_augmented_lagrangian (Float S, Float n, Float Cf,
    geo boundary, field lh, field& lambda_h, field& uh,
    Float tol, size_t max_iter, Float r)
{
  const space& Xh = uh.get_space();
  const space& Wh = lambda_h.get_space();
  trial u(Xh), lambda(Wh);
  test  v(Xh),      mu(Wh);
  form m  = integrate (u*v),
       a0 = integrate (dot(grad(u),grad(v))),
       a  = a0 + integrate (boundary, r*u*v),
       mb = integrate (lambda*mu),
       b  = integrate (boundary, u*mu);
  problem pa (a);
  derr << "# k residue" << endl;
  Float residue0 = 0;
  for (size_t k = 0; true; ++k) {
    field gamma_h = lazy_interpolate(Wh,
      compose(projection(S,n,Cf,r), lambda_h + r*uh["boundary"]));
    field delta_lambda_h = r*(uh["boundary"] - gamma_h);
    lambda_h += delta_lambda_h;
    Float residue = delta_lambda_h.max_abs();
    derr << k << " " << residue << endl;
    if (residue <= tol || k >= max_iter) return (residue <= tol) ? 0 : 1;
    field rhs = lh + b.trans_mult(r*gamma_h - lambda_h);
    pa.solve (rhs, uh);
  }
}
```

Observe also that the stopping criterion for breaking the loop bases on the max of the relative error for the λ_h variable. For this algorithm, this stopping criterion guaranties that all residual terms of the initial problem are also converging to zero, as it will be checked here. Moreover, this stopping criterion is very fast to compute while the full set of residual terms of the initial problem would take more computational time inside the loop.

Running the program

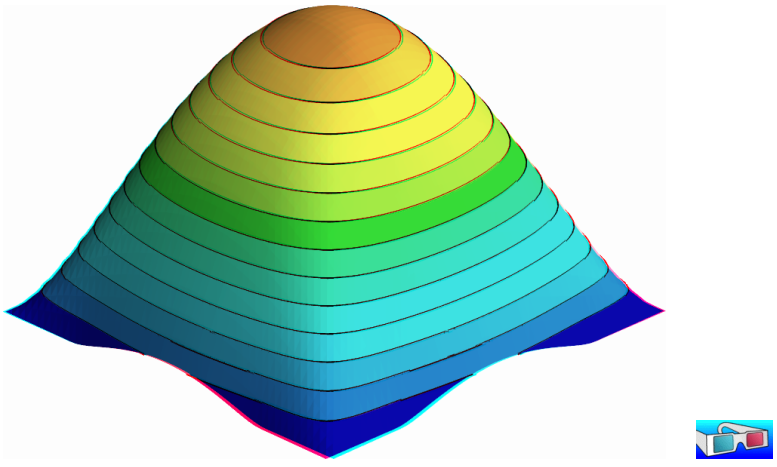


Figure 5.1: The yield slip problem for $S = 0.6$ and $n = 1$.

Assume that the previous code is contained in the file '`yield-slip-augmented-lagrangian.cc`'. Compile and run the program as usual:

```
make yield_slip_augmented_lagrangian
mkgeo_grid -a -1 -b 1 -c -1 -d 1 -t 50 > square.geo
./yield_slip_augmented_lagrangian square.geo P1 0.6 1 > square.field
```

Also you can replace `P1` by `P2`. The solution can be represented in elevation view (see Fig. 5.1):

```
field square.field -elevation -stereo
```

As analysed by [Roquet and Saramito, 2004], when $S \leqslant 0.382$, the fluid slips at the wall, when $0.382 < S < 0.674$, the fluid partially sticks and when $S \geqslant 0.674$ the fluid completely sticks. Remark that the velocity is not zero along the boundary: there is a stick-slip transition point. The velocity along along the $0x_0$ axis and the top boundary are available as (see Fig. 5.2):

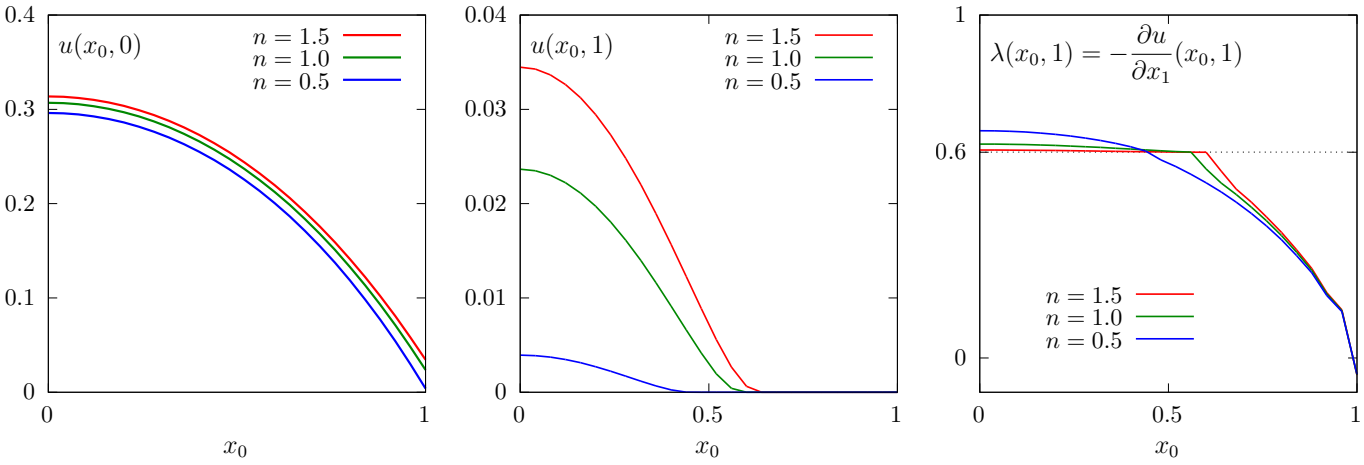


Figure 5.2: The yield slip problem for $S = 0.6$: cut of the velocity (left) along the $0x_0$ axis ; (center) along the boundary ; (right) cut of the normal stress λ along the boundary.

```
field square.field -normal 0 1 -origin 0 0 -cut -gnuplot
field square.field -domain top -elevation -gnuplot
```

The corresponding Lagrange multiplier λ on the boundary can also be viewed as:

```
field square.field -mark lambda -elevation
```

The file '`yield_slip_residue.cc`' implement the computation of the full set of residual terms of the initial problem. This file it is not listed here but is available in the **Rheolef** example directory. The computation of residual terms is obtained by:

```
make yield_slip_residue
./yield_slip_residue < square.field
```

Observe that the residual terms of the initial problem are of about 10^{-10}, as required by the stopping criterion. Fig. 5.3 plots the max of the relative error for the λ_h variable: this quantity is used as stopping criterion. Observe that it behaves asymptotically as $1/k$ for larges meshes, with a final acceleration to machine precision. Note that these convergence properties could be dramatically improved by using a Newton method, as shown in the next section.

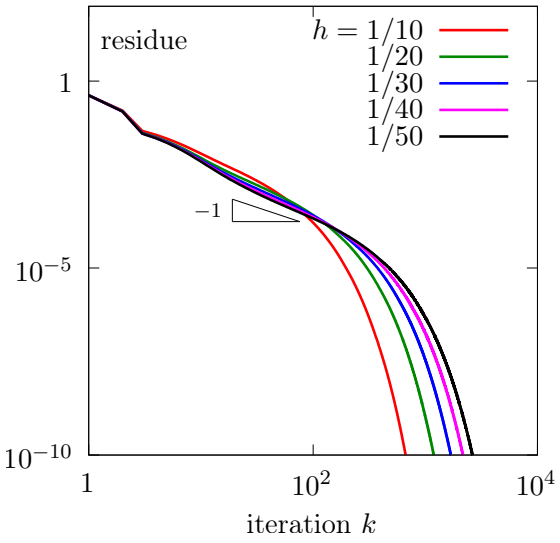


Figure 5.3: The convergence of the augmented Lagrangian algorithm for the yield slip problem with $S = 0.6$ and $n = 1$ and P_1 polynomial approximation.

5.1.3 Newton algorithm

Reformulation of the problem

The idea of this algorithm first proposed by Alart [1997] in the context of contact and friction problems. At convergence, the augmented Lagrangian method solve the following problem:
$(P)_r$: find u, defined in Ω, and λ, defined on $\partial\Omega$, such that

$$\begin{aligned} -\Delta u &= f \ \text{ in } \Omega \\ \frac{\partial u}{\partial n} + \lambda &= 0 \ \text{ on } \partial\Omega \\ u - P_{n,r}(\lambda + r\,u) &= 0 \ \text{ on } \partial\Omega \end{aligned}$$

The solution is independent upon $r \in \mathbb{R}$ and this problem is equivalent to the original one. In order to diagonalize the non-linearity in $P_{n,r}(.)$, let us introduce $\beta = \lambda + r\,u$. The problem becomes:
$(P)_r$: find u, defined in Ω, and β, defined on $\partial\Omega$, such that

$$\begin{aligned} -\Delta u &= f \ \text{ in } \Omega \\ \frac{\partial u}{\partial n} - r\,u + \beta &= 0 \ \text{ on } \partial\Omega \\ u - P_{n,r}(\beta) &= 0 \ \text{ on } \partial\Omega \end{aligned}$$

Variational formulation

Consider the following forms:

$$\begin{aligned} m(u,v) &= \int_\Omega u\,v\,\mathrm{d}x, , \quad \forall u\,v \in L^2(\Omega) \\ a(u,v) &= \int_\Omega \boldsymbol{\nabla} u\,.\boldsymbol{\nabla} v\,\mathrm{d}x - r\int_{\partial\Omega} u\,v\,\mathrm{d}s, \quad \forall u,\,v \in H^1(\Omega) \\ b(v,\gamma) &= \int_\Omega v\,\gamma\,\mathrm{d}s, \quad \forall \gamma \in L^2(\partial\Omega), \quad \forall v \in H^1(\Omega) \\ c(\beta,\gamma) &= \int_\Omega P_{n,r}(\beta)\,\gamma\,\mathrm{d}s, \quad \forall \beta \in L^\infty(\partial\Omega), \forall \gamma \in L^2(\partial\Omega) \end{aligned}$$

Remark that, since $\Omega \subset \mathbb{R}^2$, from the Sobolev embedding theorem, if $u \in H^1(\Omega)$ then $u_{|\partial\Omega} \in L^\infty(\partial\Omega)$. Then, all integrals have sense. The variational formulation writes:
(FV): find $u \in H^1(\Omega)$ and $\beta \in L^\infty(\partial\Omega)$ such that

$$\begin{aligned} a(u,v) + b(v,\beta) &= m(f,v), \quad \forall v \in H^1(\Omega) \\ b(u,\gamma) - c(\beta,\gamma) &= 0, \quad \forall \gamma \in L^2(\partial\Omega) \end{aligned}$$

Let M, A and B the operators associated to forms m, a, b and c. The problem writes also as:

$$\begin{pmatrix} A & B^* \\ B & -C \end{pmatrix} \begin{pmatrix} u \\ \beta \end{pmatrix} = \begin{pmatrix} Mf \\ 0 \end{pmatrix}$$

where B^* denotes the formal adjoint of B. The bilinear form a is symmetric positive definite when $r \in]0, C_f[$. Then A is non-singular and let A^{-1} denotes its inverse. The unknown u can be eliminated: $u = A^{-1}(Mf - B^T\beta)$ and the problem reduces to:

$$\text{find } \beta \in L^\infty(\partial\Omega) \text{ such that } F(\beta) = 0$$

where

$$F(\beta) = C(\beta) + BA^{-1}B^*\beta - BA^{-1}Mf$$

This problem is a good candidate for a Newton method:

$$F'(\beta)\delta\beta = C'(\beta)\delta\beta + BA^{-1}B^*\delta\beta$$

where $C'(\beta)$ is associated to the bilinear form:

$$c_1(\beta;\gamma,\delta) \quad = \quad \int_\Omega P'_{n,r}(\beta)\,\gamma\,\delta\,\mathrm{d}s, \quad \forall\beta\in L^\infty(\partial\Omega), \forall\gamma,\delta\in L^2(\partial\Omega)$$

where, for all $\xi\in\mathbb{R}$:

$$P'_{n,r}(\xi) = \begin{cases} 0 & \text{when } |\xi|\leqslant S \\ \phi'_{n,r}(|\xi|-S) & \text{otherwise} \end{cases} \tag{5.4}$$

Recall that, for all $\zeta>0$, $\phi_{n,r}(\zeta) = f_{n,r}^{-1}(\zeta)$ where $f_{n,r}(y) = C_f y^n + r\,y$, for all $y>0$. Then

$$\phi'_{n,r}(\zeta) = \frac{1}{f'_{n,r}(f_{n,r}^{-1}(\zeta))} = \frac{1}{f'_{n,r}(\phi_{n,r}(\zeta))} = \frac{1}{r+n\,C_f\,\{\phi_{n,r}(\zeta)\}^{-1+n}} \tag{5.5}$$

When $n=1$ we simply have: $\phi'_{1,r}(\zeta) = \dfrac{1}{C_f+r}$. When $r=0$, for any $n>0$ we have $\phi'_{n,0}(\zeta) = \dfrac{\zeta^{-1+1/n}}{n\,C_f^{1/n}}$.

File 5.6: `d_projection_dx.h`

```
#include "projection.h"
struct d_projection_dx {
  Float operator() (const Float& x) const {
    if (fabs(x) <= a) return 0;
    if (n == 1) return 1/(c + r);
    if (r == 0) return pow(fabs(x)-a,-1+1/n)/(n*pow(c,1/n));
    return 1/(r + n*c*pow(_phi(fabs(x)-a),-1+n));
  }
  d_projection_dx (Float a1, Float n1=1, Float c1=1, Float r1=0)
   : a(a1), n(n1), c(c1), r(r1), _phi(n1,c1,r1) {}
  Float a,n,c,r;
  phi _phi;
};
```

File 5.7: yield_slip_damped_newton.cc

```
#include "rheolef.h"
using namespace std;
using namespace rheolef;
#include "yield_slip.h"
int main(int argc, char**argv) {
  environment rheolef (argc,argv);
  geo omega (argv[1]);
  string approx  = (argc > 2) ?      argv[2]  : "P1";
  Float  S       = (argc > 3) ? atof(argv[3]) : 0.6;
  Float  n       = (argc > 4) ? atof(argv[4]) : 1;
  Float  Cf      = (argc > 5) ? atof(argv[5]) : 1;
  Float  r = (argc > 6) ? atof(argv[6]) : 1;
  domain boundary = omega["boundary"];
  yield_slip F (S, n, Cf, r, omega, boundary, approx);
  field beta_h = F.initial();
  Float tol = 10*numeric_limits<Float>::epsilon();
  size_t max_iter = 10000;
  int status = damped_newton (F, beta_h, tol, max_iter, &derr);
  field uh, lambda_h;
  F.post (beta_h, uh, lambda_h);
  dout << setprecision(numeric_limits<Float>::digits10)
       << catchmark("S") << S << endl
       << catchmark("n") << n << endl
       << catchmark("Cf") << Cf << endl
       << catchmark("r") << r << endl
       << catchmark("u") << uh
       << catchmark("lambda") << lambda_h;
  return status;
}
```

File 5.8: yield_slip.h

```
class yield_slip {
public:
  typedef field value_type;
  typedef Float float_type;
  yield_slip (Float S, Float n, Float Cf, Float r,
    const geo& omega, const geo& boundary, string approx = "P1");
  field residue           (const field& beta_h) const;
  void update_derivative (const field& beta_h) const;
  field derivative_solve      (const field& mrh) const;
  field derivative_trans_mult (const field& mrh) const;
  Float space_norm      (const field&) const;
  Float dual_space_norm (const field&) const;
  field initial () const;
  void post (const field& beta_h, field& uh, field& lambda_h) const;
protected:
  Float S, n, Cf, r;
  geo boundary;
  space Xh, Wh, Yh;
  field lh, mkh;
  form m, mb, a, b;
  mutable form c1;
  problem pmb, pa;
  mutable problem pA;
};
#include "yield_slip1.icc"
#include "yield_slip2.icc"
```

File 5.9: yield_slip1.icc

```
#include "d_projection_dx.h"
yield_slip::yield_slip (Float S1, Float n1, Float Cf1, Float r1,
    const geo& omega, const geo& boundary1, string approx)
 : S(S1), n(n1), Cf(Cf1), r(r1), boundary(boundary1), Xh(), Wh(), Yh(),
   lh(), mkh(), m(), mb(), a(), b(), c1(), pmb(), pa(), pA()
{
  Xh = space (omega,    approx);
  Wh = space (boundary, approx);
  Yh = Xh*Wh;
  trial u (Xh), lambda(Wh);
  test  v (Xh),     mu(Wh);
  m  = integrate(u*v);
  mb = integrate(lambda*mu);
  a  = integrate(dot(grad(u),grad(v))) - r*integrate(boundary, u*v);
  b  = integrate(boundary, u*mu);
  lh = integrate(v);
  pmb = problem (mb);
  pa  = problem (a);
  field vh(Xh);
  pa.solve (lh, vh);
  mkh = b*vh;
}
field yield_slip::residue (const field& beta_h) const {
  field vh (Xh);
  field rhs = b.trans_mult (beta_h);
  pa.solve (rhs, vh);
  test mu (Wh);
  field c0h = integrate(mu*compose(projection(S,n,Cf,r), beta_h));
  field mrh = b*vh + c0h - mkh;
  return mrh;
}
void yield_slip::update_derivative (const field& beta_h) const {
  trial lambda (Wh); test mu (Wh);
  c1 = integrate (lambda*mu*compose(d_projection_dx(S,n,Cf,r), beta_h));
  form A = { { a, trans(b) },
             { b,     -c1  } };
  A.set_symmetry (c1.is_symmetric());
  pA = problem(A);
}
```

File 5.10: yield_slip2.icc

```
#include "poisson_robin.icc"
field yield_slip::derivative_solve (const field& mrh) const {
  field mryh(Yh, 0.);
  mryh[1] = -mrh;
  field delta_yh(Yh);
  pA.solve(mryh, delta_yh);
  return delta_yh[1];
}
field yield_slip::derivative_trans_mult (const field& mrh) const {
  field rh (Wh);
  pmb.solve (mrh, rh);
  field rhs = b.trans_mult(rh);
  field delta_vh (Xh, 0.);
  pa.solve (rhs, delta_vh);
  field mgh = b*delta_vh + c1*rh;
  field gh (Wh);
  pmb.solve (mgh, gh);
  return gh;
}
Float yield_slip::space_norm (const field& rh) const {
  return sqrt (mb(rh,rh));
}
Float yield_slip::dual_space_norm (const field& mrh) const {
  field rh (Wh,0);
  pmb.solve (mrh, rh);
  return sqrt (dual (mrh,rh));
}
field yield_slip::initial () const {
  field uh = poisson_robin (Cf, boundary, lh);
  return (Cf+r)*uh["boundary"];
}
void yield_slip::post (const field& beta_h, field& uh, field& lambda_h) const {
  field rhs = lh - b.trans_mult(beta_h);
  uh = field (Xh, 0.);
  pa.solve (rhs, uh);
  lambda_h = beta_h - r*uh["boundary"];
}
```

Running the program

```
make ./yield_slip_damped_newton
./yield_slip_damped_newton square.geo P1 0.6 1 > square.field
field square.field -elevation -stereo
field square.field -mark lambda -elevation
```

Observe on Fig. 5.4.a and 5.4.b that the convergence is super-linear and mesh-independent when $n = 1/2$ and $n = 0.9$. For mesh-independent convergence of the Newton method, see e.g. the p-Laplacian example, Fig. 3.15, page 129. When $n = 0.9$, observe that the convergence depends slightly upon the mesh for rough meshes while it becomes asymptotically mesh independent for fine meshes. When $n \geqslant 1$, the convergence starts to depend also upon the mesh (Fig. 5.4.bottom-left and 5.4.bottom-right). Recall that when $n \geqslant 1$, the problem becomes non-differentiable and the convergence of the Newton method is no more assured. Nevertheless, in that case, the convergence is clearly faster (about 100 times faster) than the corresponding one with the augmented Lagrangian algorithm on the same problem (see Fig. 5.3, page 209) and moreover there is no saturation of the residual terms on large meshes.

5.1.4 Error analysis

Assume that the previous code is contained in the file '`yield-slip-augmented-lagrangian.cc`'. When Ω is the unit circle, the exact solution is known. In polar coordinates (r, θ), as the solution

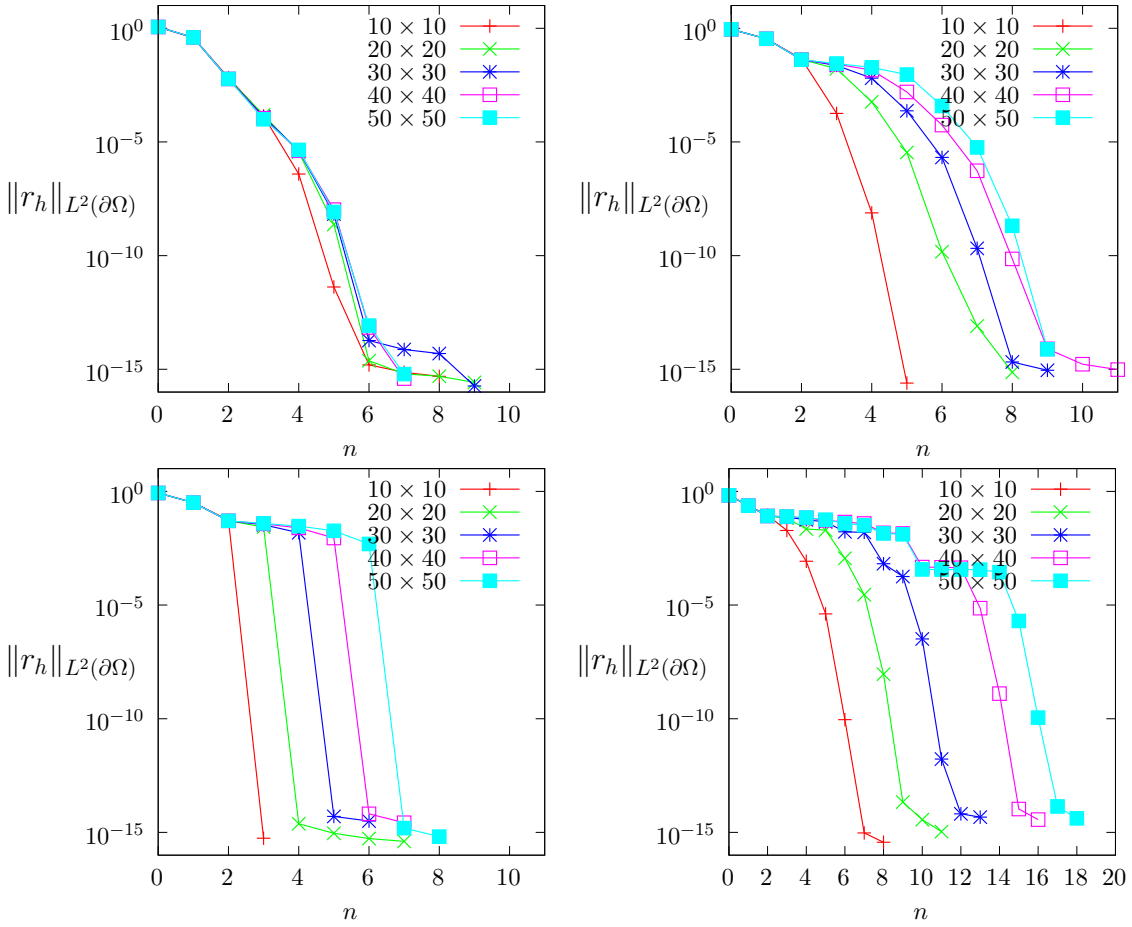


Figure 5.4: The convergence of the damped Newton algorithm for the yield slip problem ($S = 0.6$): (top-left) $n = 0.5$; (top-right) $n = 0.9$; (bottom-left) $n = 1$; (bottom-right) $n = 1.5$.

u depends only of r, equation (5.1a) becomes:

$$-\frac{1}{r}\partial_r(r\partial_r u) = 1$$

and then, from the symmetry, $u(r) = c - r^2/4$, where c is a constant to determine from the boundary condition. On the boundary $r = 1$ we have $\partial_n u = \partial_r u = -1/2$. When $S \geqslant 1/2$ the fluid sticks at the wall and when $S > 1/2$ we have from (5.1b):

$$-C_f u^n - S = \frac{1}{2}$$

From the previous expression $u(r)$, taken for $r = 1$ we obtain the constant c and then:

$$u(r) = \frac{1 - r^2}{4} + \left(\frac{\max(0, 1/2 - S)}{C_f}\right)^{1/n}, \quad 0 \leqslant r \leqslant 1$$

The error computation is implemented in the files '`yield_slip_error.cc`' and '`yield_slip_circle.h`': it is not listed here but is available in the **Rheolef** example directory.

The error can be computed (see Fig. 5.5):

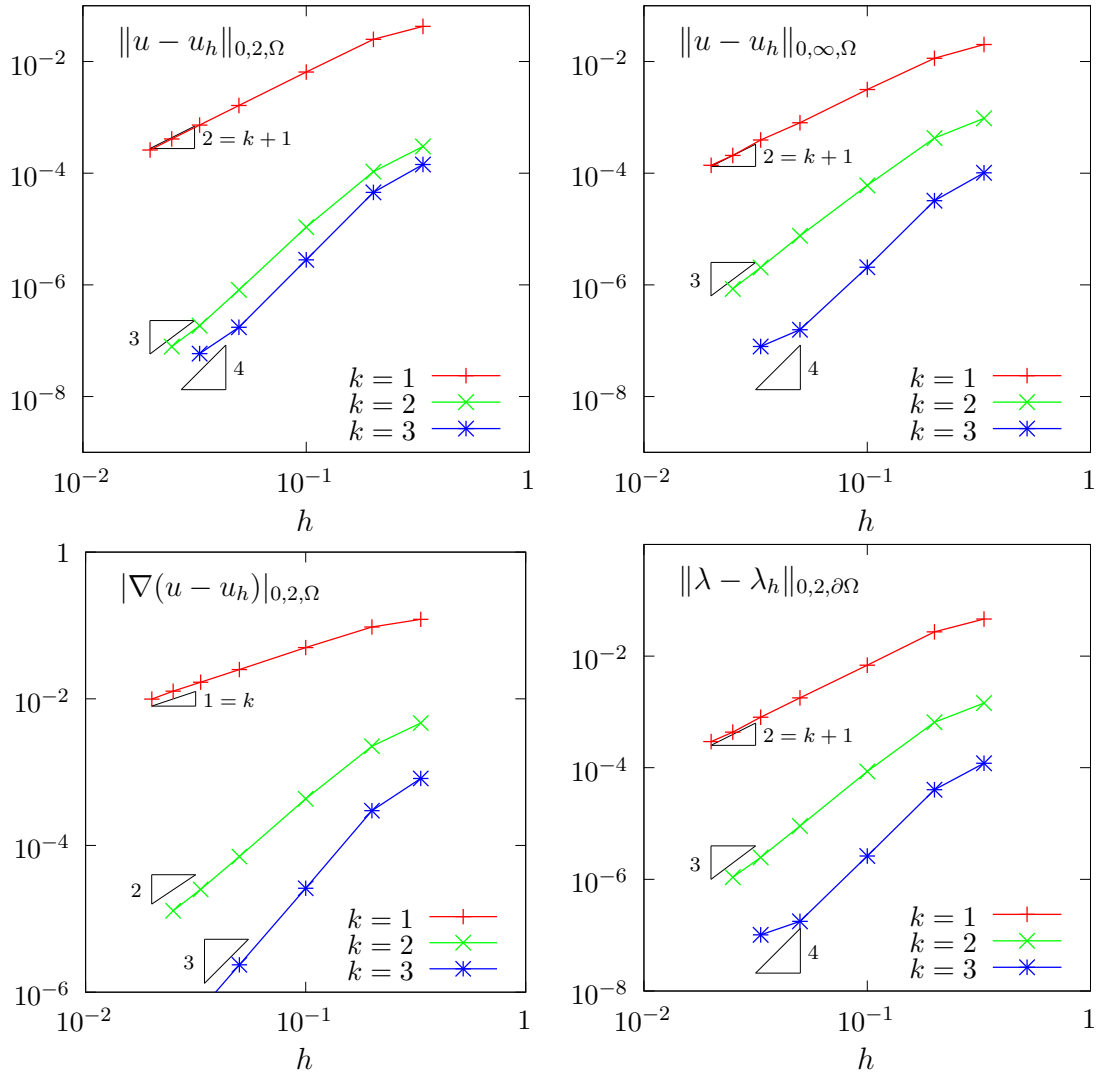


Figure 5.5: The yield slip problem: error analysis.

```
make yield_slip_error
mkgeo_ball -t 20 -order 2 > circle-20-P2.geo
./yield_slip_damped_newton circle-20-P2.geo P2 0.6 1 | ./yield_slip_error
```

It appears that the discrete formulation develops optimal convergence properties versus mesh refinement for $k \geqslant 1$ in H^1, L^2 and L^∞ norms.

5.2 Viscoplastic fluids

5.2.1 Problem statement

Viscoplastic fluids develops an yield stress behavior (see e.g. Saramito, 2016b, Saramito and Wachs, 2017). Mosolov and Miasnikov [1965, 1966, 1967] first investigated the flow of a viscoplastic in a pipe with an arbitrarily cross section. Its numerical investigation by augmented Lagrangian methods was first performed by Saramito and Roquet [2001], Roquet and Saramito [2008]. The Mosolov problem [Saramito and Roquet, 2001] writes:

(P): find $\boldsymbol{\sigma}$ and u, defined in Ω, such that

$$\operatorname{div}\boldsymbol{\sigma} = -f \quad \text{in } \Omega \tag{5.6a}$$

$$u = 0 \quad \text{on } \partial\Omega \tag{5.6b}$$

$$\left.\begin{array}{ll} |\boldsymbol{\sigma}| \leqslant Bi & \text{when } \boldsymbol{\nabla} u = 0 \\ \boldsymbol{\sigma} = |\boldsymbol{\nabla} u|^{-1+n}\boldsymbol{\nabla} u + Bi\dfrac{\boldsymbol{\nabla} u}{|\boldsymbol{\nabla} u|} & \text{otherwise} \end{array}\right\} \quad \text{in } \Omega \tag{5.6c}$$

where $Bi \geqslant 0$ is the Bingham number and $n > 0$ is a power-law index. The computational domain Ω represents the cross-section of the pipe. In the bidimensional case $d = 2$ and when f is constant, this problem describes the stationary flow of an Herschel-Bulkley fluid in a general pipe section Ω. Let Ox_2 be the axis of the pipe and Ox_0x_1 the plane of the section Ω. The vector-valued field $\boldsymbol{\sigma}$ represents the shear stress components $(\sigma_{0,1}, \sigma_{0,2})$ while u is the axial component of velocity along Ox_3. When $Bi = 0$, the problem reduces to the nonlinear p-Laplacian problem. When $n = 1$ the fluid is a Bingham fluid. When $n = 1$ and $Bi = 0$, the problem reduces to the linear Poisson one with $\boldsymbol{\sigma} = \boldsymbol{\nabla} u$.

5.2.2 The augmented Lagrangian algorithm

This problem writes as a minimization of an energy:

$$u = \operatorname*{arg\,min}_{v \in W_0^{1,p}(\Omega)} J(v) \tag{5.7a}$$

where

$$J(v) = \frac{1}{1+n}\int_\Omega |\boldsymbol{\nabla} v|^{1+n}\,\mathrm{d}x + Bi\int_\Omega |\boldsymbol{\nabla} v|\,\mathrm{d}x - \int_\Omega f\,v\,\mathrm{d}x \tag{5.7b}$$

This problem is solved by using an augmented Lagrangian algorithm. The auxiliary variable $\boldsymbol{\gamma} = \boldsymbol{\nabla} u$ is introduced together with the Lagrangian multiplier $\boldsymbol{\sigma}$ associated to the constraint $\boldsymbol{\nabla} u - \boldsymbol{\gamma} = 0$. For all $r > 0$, let:

$$L((v,\boldsymbol{\gamma});\boldsymbol{\sigma}) = \frac{1}{1+n}\int_\Omega |\boldsymbol{\gamma}|^{1+n}\,\mathrm{d}x + Bi\int_\Omega |\boldsymbol{\gamma}|\,\mathrm{d}x - \int_\Omega f\,v\,\mathrm{d}x + \int_\Omega \boldsymbol{\sigma}.(\boldsymbol{\nabla} u - \boldsymbol{\gamma})\,\mathrm{d}x + \frac{r}{2}\int_\Omega |\boldsymbol{\nabla} u - \boldsymbol{\gamma}|^2\,\mathrm{d}x$$

An Uzawa-like minimization algorithm writes:

- $k = 0$: let $\lambda^{(0)}$ and $\gamma^{(0)}$ arbitrarily chosen.
- $k \geqslant 0$: let $\lambda^{(k)}$ and $\gamma^{(k)}$ being known.

$$\begin{aligned} u^{(k+1)} &:= \operatorname*{arg\,min}_{v \in W^{1,p}(\Omega)} L((v,\boldsymbol{\gamma}^{(k)});\boldsymbol{\sigma}^{(k)}) \\ \boldsymbol{\gamma}^{(k+1)} &:= \operatorname*{arg\,min}_{\boldsymbol{\delta} \in L^2(\Omega)^d} L((u^{(k+1)},\boldsymbol{\delta});\boldsymbol{\sigma}^{(k)}) \\ \boldsymbol{\sigma}^{(k+1)} &:= \boldsymbol{\sigma}^{(k)} + \rho\left(\boldsymbol{\nabla} u^{(k+1)} - \boldsymbol{\gamma}^{(k+1)}\right) \quad \text{in } \Omega \end{aligned}$$

The descent step ρ is chosen as $\rho = r$ for sufficiently large r. The Lagrangian L is quadratic in u and thus the computation of $u^{(k+1)}$ reduces to a linear problem. The non-linearity is treated when computing $\boldsymbol{\gamma}^{(k+1)}$. This operation is performed point-by-point in Ω by minimizing:

$$\boldsymbol{\gamma} := \arg\min_{\boldsymbol{\delta}\in\mathbb{R}^d} \frac{|\boldsymbol{\delta}|^{1+n}}{1+n} + \frac{r|\boldsymbol{\delta}|^2}{2} + Bi|\boldsymbol{\delta}| - \boldsymbol{\xi}.\boldsymbol{\delta}$$

where $\boldsymbol{\xi} = \boldsymbol{\sigma}^{(k)} + r\,\boldsymbol{\nabla} u^{(k+1)}$ is given. This problem is convex and its solution is unique. The solution has the form:

$$\boldsymbol{\gamma} = P_{n,r}(\boldsymbol{\xi}) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{when } |\boldsymbol{\xi}| \leqslant S \\ \phi_{n,r}(|\boldsymbol{\xi}| - S)\dfrac{\boldsymbol{\xi}}{|\boldsymbol{\xi}|} & \text{otherwise} \end{cases} \tag{5.8}$$

where $\phi_{n,r}(x) = f_{n,r}^{-1}(x)$ has been introduced in (5.2) page 204 in the context of the yield slip problem together with the scalar projector.

File 5.11: `vector_projection.h`

```
#include "phi.h"
struct vector_projection {
  Float operator() (const Float& x) const {
    if (x <= a) return 0;
    return _phi(x-a)/x;
  }
  vector_projection (Float a1, Float n=1, Float c=1, Float r=0)
   : a(a1), _phi(n,c,r) {}
  Float a;
  phi _phi;
};
```

Finally, the Uzawa-like minimization algorithm [Saramito and Roquet, 2001] writes:

- $k = 0$: let $\boldsymbol{\sigma}^{(0)}$ and $\boldsymbol{\gamma}^{(0)}$ arbitrarily chosen.
- $k \geqslant 0$: let $\boldsymbol{\sigma}^{(k)}$ and $\boldsymbol{\gamma}^{(k)}$ being known, find $u^{(k+1)}$ such that

$$\begin{aligned} -r\Delta u^{(k+1)} &= f + \operatorname{div}\left(\boldsymbol{\sigma}^{(k)} - r\boldsymbol{\gamma}^{(k)}\right) \quad \text{in } \Omega \\ u^{(k+1)} &= 0 \quad \text{on } \partial\Omega \end{aligned}$$

  and then compute explicitly $\boldsymbol{\gamma}^{(k+1)}$ and $\boldsymbol{\sigma}^{(k+1)}$:

$$\begin{aligned} \boldsymbol{\gamma}^{(k+1)} &:= P_{n,r}\left(\boldsymbol{\sigma}^{(k)} + r\boldsymbol{\nabla} u^{(k+1)}\right) \\ \boldsymbol{\sigma}^{(k+1)} &:= \boldsymbol{\sigma}^{(k)} + r\left(\boldsymbol{\nabla} u^{(k+1)} - \boldsymbol{\gamma}^{(k+1)}\right) \end{aligned} \tag{5.9}$$

Here $r > 0$ is a numerical parameter. This algorithm reduces the nonlinear problem to a sequence of linear and completely standard Poisson problems and some explicit computations. For convenience, this algorithm is implemented as the `solve` function member of a class:

File 5.12: mosolov_augmented_lagrangian1.icc

```
#include "vector_projection.h"
int mosolov_augmented_lagrangian::solve (field& sigma_h, field& uh) const {
  test v(Xh);
  derr << "# k residue" << endl;
  for (size_t k = 0; true; ++k) {
    field grad_uh = inv_mt*(b*uh);
    auto c = compose(vector_projection(Bi,n,1,r), norm(sigma_h+r*grad_uh));
    field gamma_h = lazy_interpolate(Th, c*(sigma_h + r*grad_uh));
    field delta_sigma_h = r*(grad_uh - gamma_h);
    sigma_h += delta_sigma_h;
    Float residue = delta_sigma_h.max_abs();
    derr << k << " " << residue << endl;
    if (residue <= tol || k >= max_iter) {
      derr << endl << endl;
      return (pow(residue,3) <= tol) ? 0 : 1;
    }
    field rhs = (1/r)*(lh - integrate(dot(sigma_h - r*gamma_h, grad(v))));
    pa.solve (rhs, uh);
  }
}
```

For convenience, the order of the update of the three variables u, $\boldsymbol{\gamma}$ and $\boldsymbol{\sigma}$ has been rotated: by this way, the algorithm starts with initial values for u and $\boldsymbol{\sigma}$. instead of $\boldsymbol{\gamma}$ and $\boldsymbol{\sigma}$. Observe that the projection step (5.9) is implemented by using the `interpolate` operator: this projection step interprets as point-wise at Lagrange nodes instead as a numerical resolution of the element-wise minimization problem. Note that, for the lowest order $k = 1$, these two approaches are strictly equivalent, while, when $k \geqslant 2$, the numerical solution obtained by this algorithm is no more solution of the discrete version of the saddle-point problem for the Lagrangian L. Nevertheless, the numerical solution is founded to converge to the exact solution of the initial problem (5.6a)-(5.6c): this will be checked here in a forthcoming section, dedicated to the error analysis. Observe also that the stopping criterion for breaking the loop bases on the max of the relative error for the $\boldsymbol{\sigma}_h$ variable. For this algorithm, this stopping criterion guaranties that all residual terms of the initial problem are also converging to zero, as it will be checked here. Moreover, this stopping criterion is very fast to compute while the full set of residual terms of the initial problem would take more computational time inside the loop.

The class declaration contains all model parameters, loop controls, form and space variables, together with some pre- and post-treatments:

File 5.13: mosolov_augmented_lagrangian.h

```
struct mosolov_augmented_lagrangian: adapt_option {
  mosolov_augmented_lagrangian();
  void reset (geo omega, string approx);
  void initial (field& sigma_h, field& uh) const;
  int  solve   (field& sigma_h, field& uh) const;
  void put (odiststream& out, field& sigma_h, field& uh) const;
// data:
  Float Bi, n, r, tol;
  size_t max_iter;
  mutable space Xh, Th;
  mutable field lh;
  mutable form a, b, inv_mt;
  mutable problem pa;
};
#include "mosolov_augmented_lagrangian1.icc"
#include "mosolov_augmented_lagrangian2.icc"
```

File 5.14: mosolov_augmented_lagrangian2.icc

```
mosolov_augmented_lagrangian::mosolov_augmented_lagrangian()
: Bi(0), n(1), r(1), tol(1e-10), max_iter(1000000),
  Xh(), Th(), lh(), a(), b(), inv_mt(), pa()
{}
void mosolov_augmented_lagrangian::reset (geo omega, string approx) {
  Xh = space (omega, approx);
  Xh.block ("boundary");
  string grad_approx = "P" + to_string(Xh.degree()-1) + "d";
  Th = space (omega, grad_approx, "vector");
  trial u (Xh), sigma(Th);
  test  v (Xh), tau  (Th);
  lh = integrate(2*v);
  a = integrate (dot(grad(u),grad(v)));
  b = integrate (dot(grad(u),tau));
  integrate_option iopt;
  iopt.invert = true;
  inv_mt = integrate(dot(sigma,tau), iopt);
  pa = problem (a);
}
void
mosolov_augmented_lagrangian::initial (field& sigma_h, field& uh) const {
  uh = field(Xh);
  uh ["boundary"] = 0;
  pa.solve (lh, uh);
  test tau (Th);
  field mt_grad_uh = integrate(dot(grad(uh),tau));
  sigma_h = inv_mt*mt_grad_uh;
}
void mosolov_augmented_lagrangian::put (odiststream& out,
    field& sigma_h, field& uh) const
{
  out << catchmark("Bi")    << Bi << endl
      << catchmark("n")     << n  << endl
      << catchmark("r")     << r  << endl
      << catchmark("sigma") << sigma_h
      << catchmark("u")     << uh;
}
```

File 5.15: mosolov_augmented_lagrangian.cc

```
#include "rheolef.h"
using namespace std;
using namespace rheolef;
#include "mosolov_augmented_lagrangian.h"
int main(int argc, char**argv) {
  environment rheolef (argc,argv);
  mosolov_augmented_lagrangian pb;
  geo omega (argv[1]);
  string approx  = (argc > 2) ?      argv[2]  : "P1";
  pb.Bi          = (argc > 3) ? atof(argv[3]) : 0.2;
  pb.n           = (argc > 4) ? atof(argv[4]) : 1;
  size_t n_adapt = (argc > 5) ? atoi(argv[5]) : 0;
  pb.max_iter    = (argc > 6) ? atoi(argv[6]) : 10000;
  pb.err         = (argc > 7) ? atof(argv[7]) : 1e-4;
  pb.r           = 100;
  pb.tol         = 1e-10;
  pb.hmin        = 1e-4;
  pb.hmax        = 1e-1;
  pb.ratio       = 3;
  pb.additional  = "-AbsError";
  field sigma_h, uh;
  for (size_t i = 0; true; i++) {
    pb.reset (omega, approx);
    pb.initial (sigma_h, uh);
    int status = pb.solve (sigma_h, uh);
    odiststream out (omega.name(), "field");
    pb.put (out, sigma_h, uh);
    if (i == n_adapt) break;
    space T0h (sigma_h.get_geo(), "P"+to_string(sigma_h.get_space().degree())+"d");
    field ch = lazy_interpolate (T0h, sqrt(abs(dot(sigma_h, grad(uh)))));
    omega = adapt (ch, pb);
    omega.save();
  }
}
```

The main program read parameters from the command line and performs an optional mesh adaptation loop. This implementation supports any $n > 0$, any continuous piecewise polynomial P_k, $k \geqslant 1$ and also isoparametric approximations for curved boundaries.

Running the program

Compile the program as usual:

```
make mosolov_augmented_lagrangian
mkgeo_grid -a -1 -b 1 -c -1 -d 1 -t 10 > square.geo
./mosolov_augmented_lagrangian square.geo P1 0.4 1
field -mark u square.field -elevation
```

Observe on Fig. 5.6.left the central region where the velocity is constant. A cut of the velocity field along the first bisector is obtained by:

```
field -mark u square.field -cut -origin 0 0 -normal 1 1 -gnuplot
```

Observe on Fig. 5.6.right the small regions with zero velocity, near the outer corner of the square pipe section. This region is really small but exists This question will be revisited in the next section dedicated to auto-adaptive mesh refinement.

The file 'mosolov_residue.cc' implement the computation of the full set of residual terms of the initial problem. This file it is not listed here but is available in the **Rheolef** example directory. The computation of residual terms is obtained by:

```
make mosolov_residue
zcat square.field.gz | ./mosolov_residue
```

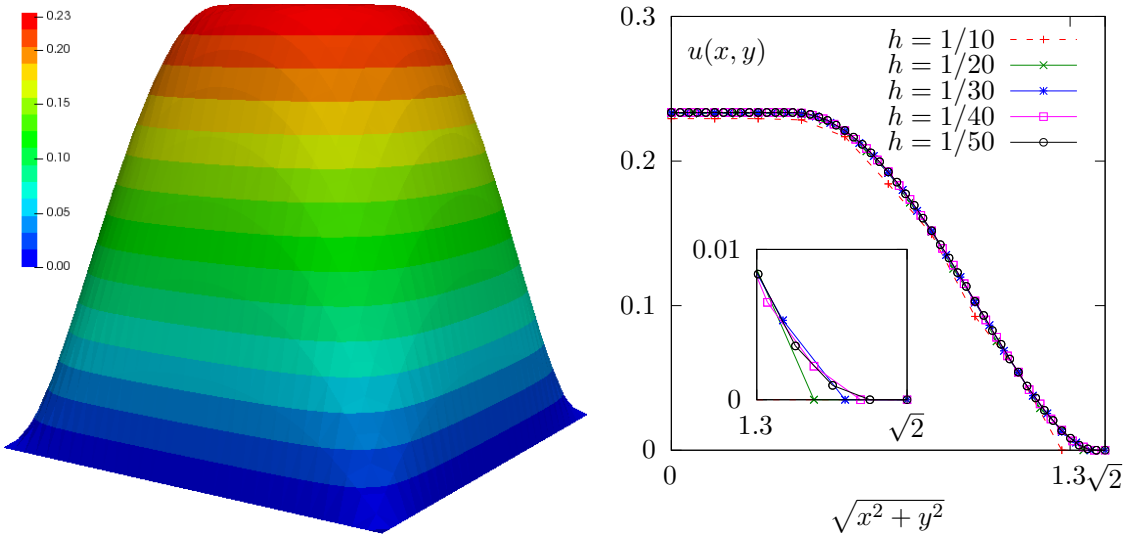


Figure 5.6: The augmented Lagrangian algorithm on the Mosolov problem with $Bi = 0.4$ and $n = 1$: (left) the velocity field in elevation view ($h = 1/30$); (right) velocity cut along the first bisector for various h.

Observe that the residual terms of problem (5.6a)-(5.6c) are of about 10^{-10}, as required by the stopping criterion. Fig. 5.7 plots the max of the relative error for the $\boldsymbol{\sigma}_h$ variable: this quantity is used as stopping criterion. Observe that it behaves asymptotically as $1/k$ for large iteration number k. Note that these convergence properties could be dramatically improved by using a Newton method, as shown by Saramito [2016a].

Finally, computation can be performed for any $n > 0$, any polynomial order $k \geqslant 1$ and in a distributed environment for enhancing performances on larger meshes:

```
mkgeo_grid -a -1 -b 1 -c -1 -d 1 -t 40 > square-40.geo
mpirun -np 8 ./mosolov_augmented_lagrangian square-40.geo P2 0.4 0.5
field -mark u square-40.field -elevation
```

The computation could take about ten minutes. The `mpirun -np 8` prefix is optional and you should omit it if you are running a sequential installation of **Rheolef**.

5.2.3 Mesh adaptation

An important improvement can be obtained by using mesh adaptation, as shown in [Saramito and Roquet, 2001, Roquet and Saramito, 2003, 2008]: with a well chosen criterion, rigid regions, where the velocity is constant, can be accurately determined with reasonable mesh sizes. This is especially true for obtaining an accurate determination of the shape of the small regions with zero velocity in the outer corner of the pipe section. In order to reduce the computational time, we can reduce the pipe section flow to only a sector, thanks to symmetries (see Fig. 5.8):

```
mkgeo_sector
geo sector.geo
```

Then, the computation is run by indicating an adaptation loop with ten successive meshes:

```
make mosolov_augmented_lagrangian
mpirun -np 8 ./mosolov_augmented_lagrangian sector P2 0.5 1 10
```

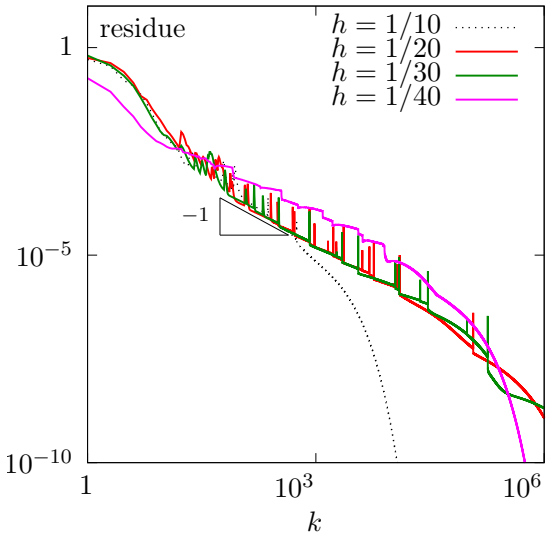


Figure 5.7: The augmented Lagrangian algorithm on the Mosolov problem with $Bi = 0.4$ and $n = 1$: residue versus iteration k for various h.

This computation could take about one hour. The `mpirun -np 8` prefix is optional and you should omit it if you are running a sequential installation of **Rheolef**.

Fig. 5.9 shows the evolution of the mesh size and the minimal edge length during the adaptation loop: observe the convergence of the meshes to an optimal one.

File 5.16: `mosolov_yield_surface.cc`

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
int main(int argc, char**argv) {
  environment rheolef (argc,argv);
  Float tol = (argc > 1) ? atof(argv[1]) : 1e-15;
  Float Bi;
  field sigma_h;
  din >> catchmark("Bi")    >> Bi
      >> catchmark("sigma") >> sigma_h;
  space Th = sigma_h.get_space();
  space Th1 (Th.get_geo(), "P" + to_string(4*(Th.degree()+1)) + "d");
  dout << lazy_interpolate (Th1, norm(sigma_h)-Bi);
}
```

The yield surface is the zero isosurface of the level set function $\phi(\boldsymbol{x} = |\boldsymbol{\sigma}_h(\boldsymbol{x}| - Bi$. Its visualization is easy to obtain by the following commands:

```
make mosolov_yield_surface
zcat sector-010.field.gz | ./mosolov_yield_surface | \
      field - -proj P1 -n-iso 10 -n-iso-negative 5
```

The unyielded zones, associated to negative values, appear in cold colors. Conversely, the yielded ones are represented by warm colors. A combined representation of the solution can be obtained by the following command:

```
bash mkview_mosolov sector-010.field.gz
```

The shell script `mkview_mosolov` invokes `mosolov_yield_surface` and then directly builds the view shown on Fig. 5.8 in the `paraview` graphic render. Please, click to deselect the show box option for completing the view. A cut of the velocity field along the first bisector is obtained by:

```
field -mark u sector-010.field.gz -mark u -domain bisector -elevation -gnuplot
```

Observe on Fig. 5.8.bottom the good capture of the small regions with zero velocity, near the outer corner of the square pipe section. When compared with Fig. 5.6.right, the benefict of mesh adaptation appears clearly.

5.2.4 Error analysis

Theoretical error bounds for this problem can be found in [Roquet et al., 2000]. In order to study the error between the numerical solution and the exact solution of the Mosolov problem, let us investigate a case for which the exact solution can be explicitly expressed. We consider the special case of a flow of a viscoplastic fluid in a circular pipe. The pressure is expressed by $p(z) = -fz$. The velocity has only one nonzero component along the $0z$ axis, denoted as $u(r)$ for simplicity. Conversely, the symmetric tensor $\boldsymbol{\sigma}$ has only one non-zero rz component, denoted as $\sigma(r)$. Thanks to the expression of the tensor-divergence operator in axisymmetric coordinates [Bird et al., 1987, p. 588], the problem reduces to :

(P) : find $\sigma(r)$ and $u(r)$, defined in $]-R,R[$ such that

$$\left\{\begin{array}{ll} \sigma(r) = K|u'(r)|^{-1+n}\,u'(r) + \sigma_0\dfrac{u'(r)}{|u'(r)|}, & \text{when } u'(r) \neq 0 \\ |\sigma(r)| \leqslant \sigma_0 & \text{otherwise} \end{array}\right.$$

$$\begin{aligned} -\frac{1}{r}(r\sigma)' - f &= 0 \quad \text{in }]-R,R[\\ u(-R) = u(R) &= 0 \end{aligned}$$

Let $\Sigma = fR/2$ be a representative stress and U a representative velocity such that $K(U/R)^n = \Sigma$. Then, we consider the following change of unknown:

$$r = R\tilde{r}, \quad u = U\tilde{u}, \quad \sigma = \Sigma\tilde{\sigma}$$

The system reduces to a problem with only two parameters n and the Bingham number $Bi = 2\sigma_0/(fR)$, that measures the ratio between the yield stress and the load. Since there is no more ambiguity, we omit the tildes :

(P) : find σ and u, defined in $]-1,1[$ such that

$$\left\{\begin{array}{ll} \sigma(r) = |u'(r)|^{-1+n}\,u'(r) + Bi\dfrac{u'(r)}{|u'(r)|}, & \text{when } u'(r) \neq 0 \\ |\sigma(r)| \leqslant Bi & \text{otherwise} \end{array}\right.$$

$$\begin{aligned} -\tfrac{1}{r}(r\sigma)' - 2 &= 0 \quad \text{in }]-1,1[\\ u(-1) = u(1) &= 0 \end{aligned}$$

Remark that the solution is even : $u(-r) = u(r)$. We get $\sigma(r) = -r$ and the yield stress criterion leads to $u(x) = 0$ when $|r| \leqslant Bi$: the load is weaker than the yield stress and the flow is null. When $Bi > 1$ the solution is $u = 0$. Otherwise, when $|r| > Bi$, we get $|u'(r)|^n + Bi = |r|$ and finally, with the boundary conditions and the continuity at $r = \pm Bi$:

$$u(r) = \frac{(1-Bi)^{1+\frac{1}{n}} - \max(0,\,|r|-Bi)^{1+\frac{1}{n}}}{1+\frac{1}{n}}$$

When $n = 1$, the second derivative of the solution is discontinuous at $r = Bi$ and its third derivative is not square integrable. For any $n > 0$, an inspection of the integrability of the square of the solution derivatives shows that $u \in H^{1+1/n}(]-1,1[,\, r\,\mathrm{d}r)$ at the best.

File 5.17: `mosolov_exact_circle.h`

```
struct u {
  Float operator() (const point& x) const {
    return (pow(1-Bi,1+1/n) - pow(max(Float(0),norm(x)-Bi),1+1/n))/(1+1/n);
  }
  u (Float Bi1, Float n1) : Bi(Bi1), n(n1) {}
  protected: Float Bi, n;
};
struct grad_u {
  point operator() (const point& x) const {
    Float r = norm(x);
    return (r <= Bi) ? point(0,0) : -pow(r-Bi, 1/n)*(x/r);
  }
  grad_u (Float Bi1, Float n1) : Bi(Bi1), n(n1) {}
  protected: Float Bi, n;
};
struct sigma {
  point operator() (const point& x) const { return -x; }
  sigma (Float=0, Float=0) {}
};
```

When computing on a circular pipe section, the exact solution is known and it is also possible to compute the error: this is implemented in the file '`mosolov_error.cc`'. This file it is not listed here but is available in the **Rheolef** example directory. The error analysis is obtained by:

```
make mosolov_error
mkgeo_ball -order 2 -t 10 > circle-P2-10.geo
./mosolov_augmented_lagrangian circle-P2-10.geo P2 0.2 0.5
zcat circle-P2-10.field | ./mosolov_error
```

Note that we use an high order isoparametric approximation of the flow domain for tacking into account the the curved boundaries. Observe on Fig. 5.10 that both the error in H^1 norm for the velocity u behaves as $\mathcal{O}(h^s)$ with $s = \min(k, 2)$. This is optimal, as, from interpolation theory [Brenner and Scott, 2002, p. 109], we have:

$$\|u - \pi_h(u)\|_{1,2,\Omega} \leqslant Ch^s |u|_{s+1,2,\Omega}$$

with $s = \min(k, 1/n)$ when $u \in H^{1+1/n}(\Omega)$. Then, the convergence rate of the error in H^1 norm versus the mesh size is bounded by $1/n$ for any polynomial order k. The rate for the error in L^∞ norm for for the velocity u is $\min(k + 1, 1/n)$, for any $k \geqslant 1$ and $n > 0$. For the stress $\boldsymbol{\sigma}$, the error in L^2 norm behaves as $\mathcal{O}(h)$ for any polynomial order and, in L^∞ norm, the rate is weaker, of about 3/4. Finally, for $n = 1/2$ there is no advantage of using polynomial order more than $k = 2$ with quasi-uniform meshes. Conversely, for $n = 1$, we obtains that there is no advantage of using polynomial order more than $k = 1$ with quasi-uniform meshes. This limitation can be circumvented by combining mesh adaptation with high order polynomials [Roquet et al., 2000, Saramito and Roquet, 2001].

5.2.5 Error analysis for the yield surface

The limit contour separating the rigid zones are expressed as a level set of the stress norm:

$$\Gamma_h = \{\boldsymbol{x} \in \Omega \; ; \; |\boldsymbol{\sigma}_h(\boldsymbol{x})| = Bi\}$$

This contour is called the *yield surface* and its exact value is known from the exact solution $\boldsymbol{\sigma}(\boldsymbol{x}) = \boldsymbol{x}$ in the circle:

$$\Gamma = \{\boldsymbol{x} \in \Omega \; ; \; |\boldsymbol{x}| = Bi\}$$

Recall the stress $\boldsymbol{\sigma}_h$ converges in L^∞ norm, thus, there is some hope that its point-wise values converge. As these point-wise values appears in the definition of the yield surface, this suggests

that Γ_h could converge to Γ with mesh refinement: our aim is to check this conjecture. Let us introduce the area between Γ_h and Γ as a L^1 measure of the distance between them:

$$\mathrm{dist}(\Gamma,\Gamma_h) = \int_\Omega \delta\left(|\boldsymbol{\sigma}_h| - Bi,\ |\boldsymbol{\sigma}| - Bi\right)\,\mathrm{d}x$$

where

$$\delta(\phi,\psi) = \begin{cases} 0 & \text{when } \phi\psi \geqslant 0 \\ 1 & \text{otherwise} \end{cases}$$

File 5.18: `mosolov_error_yield_surface.cc`

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
#include "mosolov_exact_circle.h"
Float delta (Float f, Float g) { return (f*g >= 0) ? 0 : 1; }
int main(int argc, char**argv) {
  environment rheolef (argc,argv);
  Float tol = (argc > 1) ? atof(argv[1]) : 1e-15;
  Float Bi;
  field sigma_h;
  din >> catchmark("Bi")    >> Bi
      >> catchmark("sigma") >> sigma_h;
  space Th = sigma_h.get_space();
  geo omega = Th.get_geo();
  integrate_option iopt;
  iopt.set_family(integrate_option::gauss);
  iopt.set_order(4*(Th.degree()+1));
  Float err_ys_l1 = integrate (omega,
     compose(delta, norm(sigma_h)-Bi, norm(sigma())-Bi), iopt);
  dout << "err_ys_l1 = " << err_ys_l1 << endl;
  return err_ys_l1 < tol ? 0 : 1;
}
```

The computation of the error for the yield surface prediction writes:

```
make mosolov_error_yield_surface
zcat circle-P2-10.field.gz | ./mosolov_error_yield_surface
```

Fig. 5.11.right shows the result: the yield surface error converges as $\mathcal{O}(h)$ for any $k \geqslant 0$.

Recall that the yield surface is the zero isosurface of the level set function $\phi(\boldsymbol{x} = |\boldsymbol{\sigma}_h(\boldsymbol{x}| - Bi$. Its visualization, shown on Fig. 5.11.left, is provided by the following commands:

```
make mosolov_yield_surface
zcat circle-P2-10.field.gz | ./mosolov_yield_surface | \
      field - -proj P1 -n-iso 10 -n-iso-negative 5
```

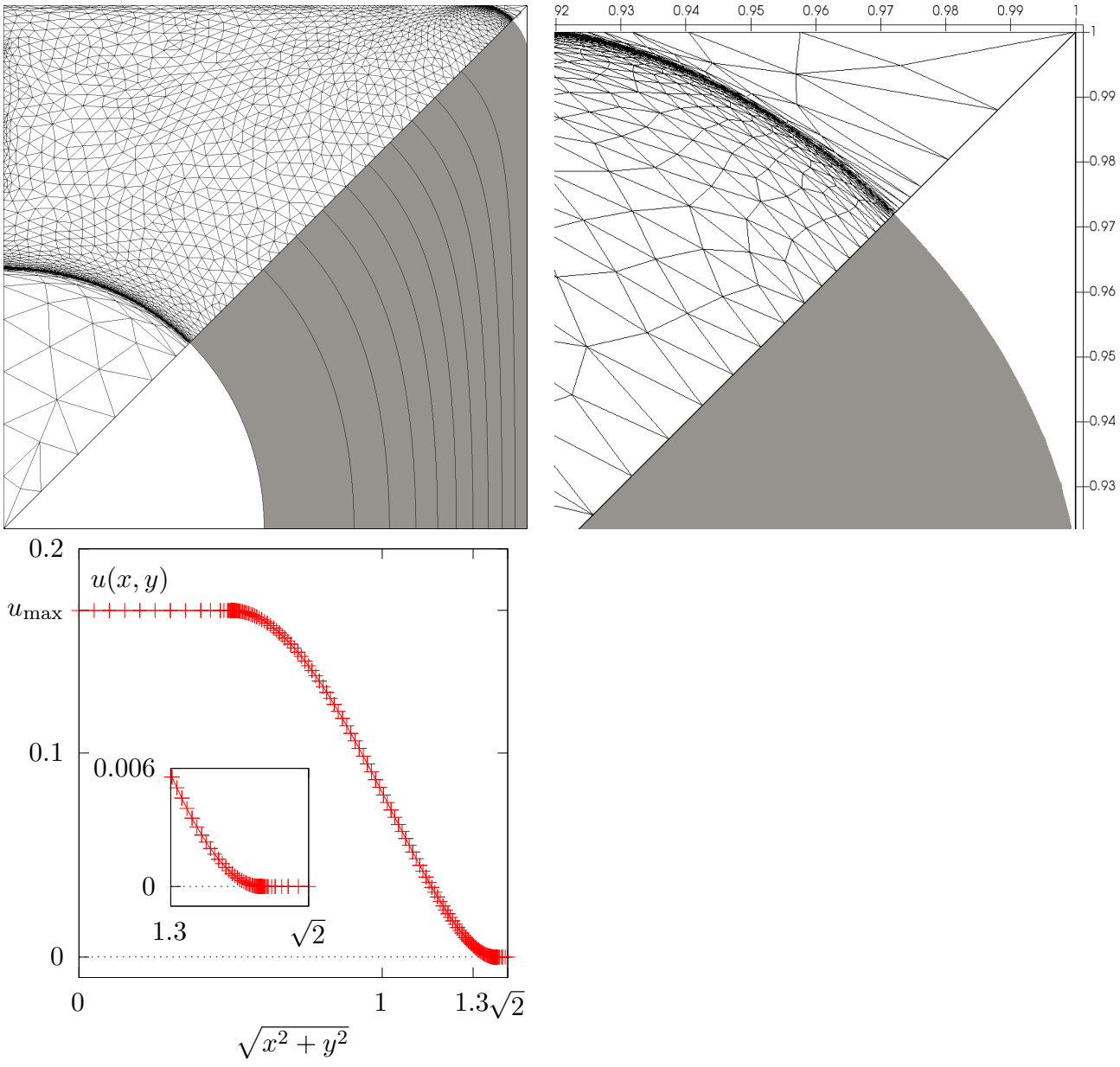


Figure 5.8: Auto-adaptive meshes for the Mosolov problem with $Bi = 0.5$ and $n = 1$ and the P_2 element for the velocity: (top) Yielded regions in gray and nine isovalues of the velocity inside $]0, u_{\max}[$ with $u_{\max} = 0.169865455242435$. The auto-adaptive mesh is mirrored. (top-right) Zoom in the outer corner of the square section. (bottom) Velocity cut along the first bisector.

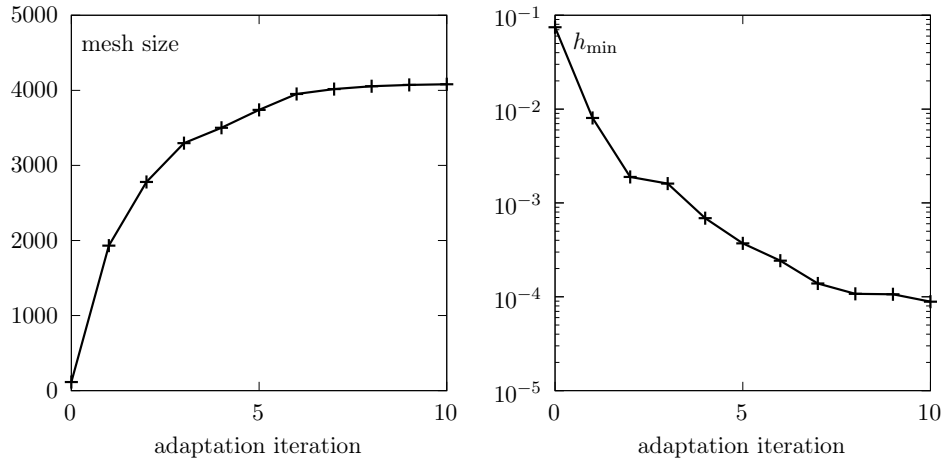


Figure 5.9: Auto-adaptive meshes for the Mosolov problem: evolution of the mesh size (left) and the minimal edge length during the adaptation loop.

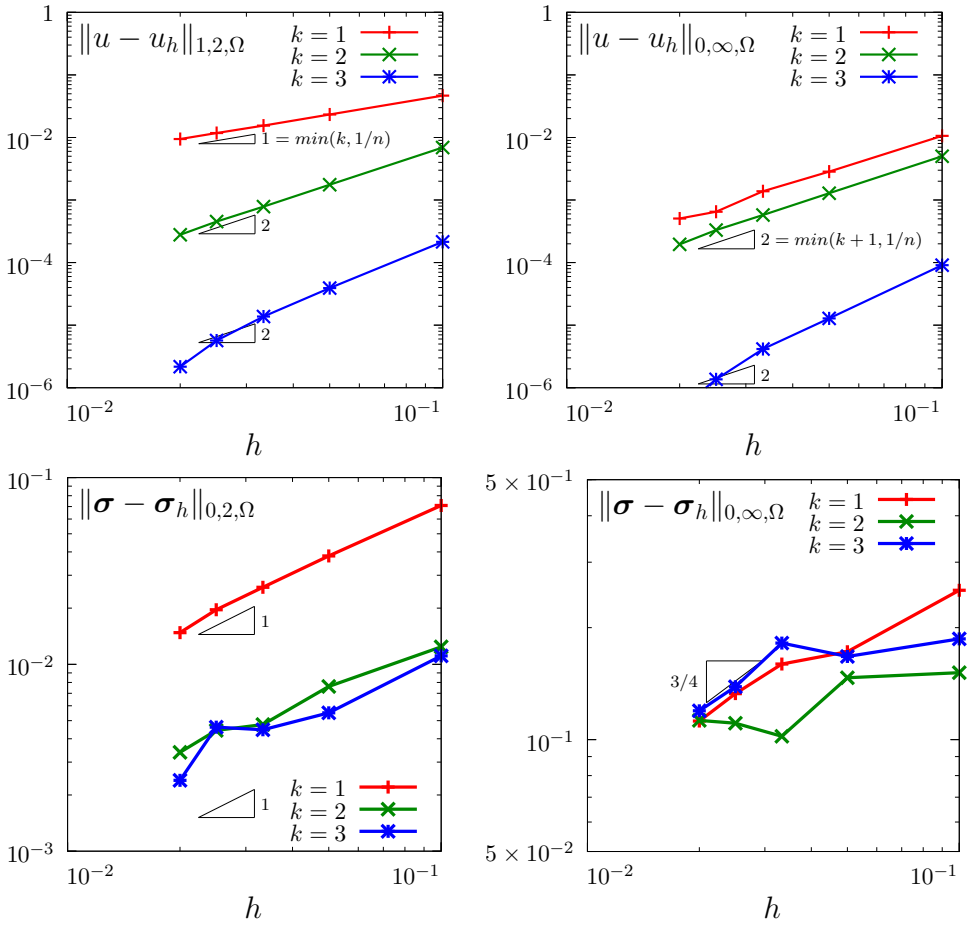


Figure 5.10: The augmented Lagrangian algorithm on the Mosolov problem with $Bi = 0.2$ and $n = 1/2$: error versus mesh paremeter h for various polynomial order k.

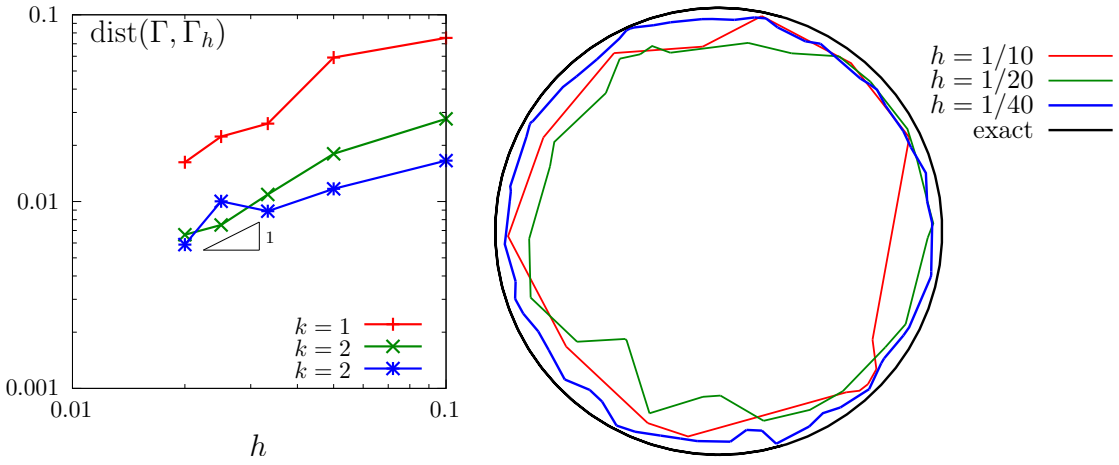


Figure 5.11: The augmented Lagrangian algorithm on the Mosolov problem with $Bi = 0.2$ and $n = 1/2$: (left) convergence of the yield surface versus mesh parameter h for various polynomial order k ; (right) visualization of the yield surface (P_1 approximation).

5.3 Viscoelastic fluids

5.3.1 A tensor transport equation

The aim of this chapter is to introduce to the numerical resolution of equations involving tensor derivatives by using discontinuous approximations. See Saramito [2013b, chap. 4] for an introduction to tensor derivatives and in section 4.1.1 of the present book, page 147, for discontinuous Galerkin methods.

The tensor derivative of a symmetric tensor $\boldsymbol{\sigma}$ is defined by:

$$\frac{\mathscr{D}_a\boldsymbol{\sigma}}{\mathscr{D}t} = \frac{\partial\boldsymbol{\sigma}}{\partial t} + (\mathbf{u}.\nabla)\boldsymbol{\sigma} + \boldsymbol{\sigma}\boldsymbol{g}_a(\mathbf{u}) + \boldsymbol{g}_a^T(\mathbf{u})\boldsymbol{\sigma}, \tag{5.10}$$

where $\mathbf{u}$ is a given velocity field,

$$\boldsymbol{g}_a(\mathbf{u}) = \left((1-a)\ \boldsymbol{\nabla}\mathbf{u} - (1+a)\ \boldsymbol{\nabla}\mathbf{u}^T\right)/2 \tag{5.11}$$

is a generalized velocity grandient and $a \in [-1,1]$ is the parameter of the tensorderivative. Problems involving tensor derivatives appear in viscoelasticity (polymer solution and polymer melt, see e.g. Saramito, 2016b), in fluid-particle suspension modeling (see e.g. Ozenda et al., 2018), in turbulence modeling ($R_{ij}-\epsilon$ models) or in liquid crystals modeling. Let $\Omega \subset \mathbb{R}^d$ be a bounded open domain.

The time-dependent tensor transport problem writes:

(P): *find* $\boldsymbol{\sigma}$, *defined in* $]0,T[\times\Omega$, *such that*

$$\begin{aligned}
\frac{\mathscr{D}_a\boldsymbol{\sigma}}{\mathscr{D}t} + \nu\boldsymbol{\sigma} &= \boldsymbol{\chi} \quad \text{in }]0,T[\times\Omega \\
\boldsymbol{\sigma} &= \boldsymbol{\sigma}_\Gamma \quad \text{on }]0,T[\times\partial\Omega_- \\
\boldsymbol{\sigma}(0) &= \boldsymbol{\sigma}_0 \quad \text{in } \Omega
\end{aligned}$$

where $\boldsymbol{\sigma}$ is the tensor valued unknown and $\nu > 0$ is a constant that represents the inverse of the Weissenberg number. Also $T > 0$ is a given final time, the data $\boldsymbol{\chi}$, $\boldsymbol{\sigma}_\Gamma$ and $\boldsymbol{\sigma}_0$ are known and $\partial\Omega_-$ denotes the upstream boundary (see section 4.1.1, page 147).

The steady version of the tensor transport problem writes:

(S): *find* $\boldsymbol{\sigma}$, *defined in* Ω, *such that*

$$\begin{aligned}
(\mathbf{u}.\nabla)\boldsymbol{\sigma} + \boldsymbol{\sigma}\boldsymbol{g}_a(\mathbf{u} + \boldsymbol{g}_a^T(\mathbf{u})\boldsymbol{\sigma}) + \nu\boldsymbol{\sigma} &= \boldsymbol{\chi} \text{ in } \Omega \\
\boldsymbol{\sigma} &= \boldsymbol{\sigma}_\Gamma \quad \text{on } \partial\Omega_-
\end{aligned}$$

A sufficient condition this problem to be well posed is [Saramito, 1994, 2013b]:

$$\mathbf{u} \in W^{1,\infty}(\Omega)^d \quad \text{and} \quad 2\nu - \|\operatorname{div}\mathbf{u}\|_{0,\infty,\Omega} - 2a\|D(\mathbf{u})\|_{0,\infty,\Omega} > 0$$

Note that this condition is always satisfied when $\operatorname{div}\mathbf{u} = 0$ and $a = 0$. We introduce the space:

$$X = \{\boldsymbol{\tau} \in L^2(\Omega)_s^{d\times d};\ (\mathbf{u}.\nabla)\boldsymbol{\tau} \in L^2(\Omega)_s^{d\times d}\}$$

and, for all $\boldsymbol{\sigma}, \boldsymbol{\tau} \in X$

$$\begin{aligned}
a(\boldsymbol{\sigma},\boldsymbol{\tau}) &= \int_\Omega \left((\mathbf{u}.\nabla)\boldsymbol{\sigma} + \boldsymbol{\sigma}\boldsymbol{g}_a(\mathbf{u}) + \boldsymbol{g}_a^T(\mathbf{u})\boldsymbol{\sigma} + \nu\,\boldsymbol{\sigma}\right) : \boldsymbol{\tau}\,\mathrm{d}x + \int_{\partial\Omega} \max(0, -\mathbf{u}.\mathbf{n})\,\boldsymbol{\sigma} : \boldsymbol{\tau}\,\mathrm{d}s \\
l(\boldsymbol{\tau}) &= \int_\Omega \boldsymbol{\chi} : \boldsymbol{\tau}\,\mathrm{d}x + \int_{\partial\Omega} \max(0, -\mathbf{u}.\mathbf{n})\,\boldsymbol{\sigma}_\Gamma : \boldsymbol{\tau}\,\mathrm{d}s
\end{aligned}$$

Then, the variational formulation of the steady problem writes:

(FV): *find* $\boldsymbol{\sigma} \in X$ *such that*

$$a(\boldsymbol{\sigma}, \boldsymbol{\tau}) \quad = \quad l(\boldsymbol{\tau}),\ \forall \boldsymbol{\tau} \in X$$

Note that the term $\max(0, -\mathbf{u}.\mathbf{n}) = (|\mathbf{u}.\mathbf{n}| - \mathbf{u}.\mathbf{n})/2$ is positive and vanishes everywhere except on $\partial\Omega_-$. Thus, the boundary condition $\phi = \phi_\Gamma$ is weakly imposed on $\partial\Omega_-$ via the integrals on the boundary. We aim at adapting the discontinuous Galerkin method to this problem. The *discontinuous* finite element space is defined by:

$$X_h = \{\boldsymbol{\tau}_h \in L^2(\Omega)^{d\times d}_s; \boldsymbol{\tau}_{h|K} \in P_k,\ \forall K \in \mathcal{T}_h\}$$

where $k \geqslant 0$ is the polynomial degree. Note that $X_h \not\subset X$ and that the $\nabla\boldsymbol{\tau}_h$ term has no more sense for discontinuous functions $\boldsymbol{\tau}_h \in X_h$. We introduce the *broken gradient* ∇_h as a convenient notation:

$$(\nabla_h \boldsymbol{\tau}_h)_{|K} = \nabla(\boldsymbol{\tau}_{h|K}),\ \forall K \in \mathcal{T}_h$$

Thus

$$\int_\Omega ((\mathbf{u}.\nabla_h)\boldsymbol{\sigma}_h) : \boldsymbol{\tau}_h \,\mathrm{d}x = \sum_{K\in\mathcal{T}_h} \int_K ((\mathbf{u}.\nabla)\boldsymbol{\sigma}_h) : \boldsymbol{\tau}_h \,\mathrm{d}x,\ \forall \boldsymbol{\sigma}_h, \boldsymbol{\tau}_h \in X_h$$

This leads to a discrete version a_h of the bilinear form a, defined for all $\boldsymbol{\sigma}_h, \boldsymbol{\tau}_h \in X_h$ by:

$$\begin{aligned} a_h(\boldsymbol{\sigma}_h, \boldsymbol{\tau}_h) &= t_h(\boldsymbol{u};\, \boldsymbol{\sigma}_h, \boldsymbol{\tau}_h) + \nu \int_\Omega \boldsymbol{\sigma}_h : \boldsymbol{\tau}_h \,\mathrm{d}x \\ t_h(\boldsymbol{u};\, \boldsymbol{\sigma}_h, \boldsymbol{\tau}_h) &= \int_\Omega \big((\mathbf{u}.\nabla_h)\boldsymbol{\sigma}_h + \boldsymbol{\sigma}\boldsymbol{g}_a(\mathbf{u}) + \boldsymbol{g}_a^T(\mathbf{u})\boldsymbol{\sigma}\big) : \boldsymbol{\tau}_h \,\mathrm{d}x + \int_{\partial\Omega} \max(0, -\mathbf{u}.\mathbf{n})\, \boldsymbol{\sigma}_h : \boldsymbol{\tau}_h \,\mathrm{d}s \\ &\quad + \sum_{S\in\mathscr{S}_h^{(i)}} \int_S \Big(-\mathbf{u}.\mathbf{n}\, [\![\boldsymbol{\sigma}_h]\!] : \{\!\{\boldsymbol{\tau}_h\}\!\} + \frac{\alpha}{2}|\mathbf{u}.\mathbf{n}|\, [\![\boldsymbol{\sigma}_h]\!] : [\![\boldsymbol{\tau}_h]\!] \Big) \,\mathrm{d}s \end{aligned} \tag{5.12}$$

$(FV)_h$: *find* $\boldsymbol{\sigma}_h \in X_h$ *such that*

$$a_h(\boldsymbol{\sigma}_h, \boldsymbol{\tau}_h) \quad = \quad l(\boldsymbol{\tau}_h),\ \forall \boldsymbol{\tau}_h \in X_h$$

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

File 5.19: transport_tensor_dg.cc

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
#include "transport_tensor_exact.icc"
int main(int argc, char**argv) {
  environment rheolef (argc, argv);
  geo omega (argv[1]);
  space Xh (omega, argv[2], "tensor");
  Float alpha = (argc > 3) ? atof(argv[3]) : 1;
  Float nu    = (argc > 4) ? atof(argv[4]) : 3;
  Float t0    = (argc > 5) ? atof(argv[5]) : acos(-1.)/8;
  Float a = 0;
  trial sigma (Xh); test tau (Xh);
  tensor ma = 0.5*((1-a)*grad_u - (1+a)*trans(grad_u));
  auto beta_a = sigma*ma + trans(ma)*sigma;
  form  ah = integrate (ddot(grad_h(sigma)*u + beta_a + nu*sigma,tau))
           + integrate ("boundary",
                max(0, -dot(u,normal()))*ddot(sigma,tau))
           + integrate ("internal_sides",
                - dot(u,normal())*ddot(jump(sigma),average(tau))
                + 0.5*alpha*abs(dot(u,normal()))
                           *ddot(jump(sigma),jump(tau)));
  field lh = integrate (ddot(chi(nu,t0),tau))
           + integrate ("boundary",
                max(0, -dot(u,normal()))*ddot(sigma_g(nu,t0),tau));
  field sigma_h(Xh);
  problem p (ah);
  p.solve (lh, sigma_h);
  dout << catchmark("nu")    << nu << endl
       << catchmark("t0")    << t0 << endl
       << catchmark("sigma") << sigma_h;
}
```

Running the program

Let $d = 2$ and $\Omega =]-1/2, 1/2[^2$. We consider the rotating field $\mathbf{u} = (-x_2, x_1)$. A particular solution of the time-dependent problem with zero right-hand side is given by:

$$\boldsymbol{\sigma}(x,t) = \frac{1}{2}\exp\left\{-\nu t - \frac{(x_1 - x_{1,c}(t))^2 + (x_2 - x_{2,c}(t))^2}{r_0^2}\right\} \times \begin{pmatrix} 1+\cos(2t) & \sin(2t) \\ \sin(2t) & 1-\cos(2t) \end{pmatrix}$$

where $x_{1,c}(t) = \bar{x}_{1,c}\cos(t) - \bar{x}_{2,c}\sin(t)$ and $x_{2,c}(t) = \bar{x}_{1,c}\sin(t) + \bar{x}_{2,c}\cos(t)$ with $r_0 > 0$ and $(\bar{x}_{1,c}, \bar{x}_{2,c}) \in \mathbb{R}^2$. The initial condition is chosen as $\boldsymbol{\sigma}_0(x) = \boldsymbol{\sigma}(0,x)$. This exact solution is implemented in the file '`transport_tensor_exact.icc`'. This file it is not listed here but is available in the **Rheolef** example directory. For the steady problem, the right-hand side could be chosen as $\boldsymbol{\chi} = -\dfrac{\partial \boldsymbol{\sigma}}{\partial t}$ and then $t = t_0$ is fixed. The numerical tests correspond to $\nu = 3$, $r_0 = 1/10$, $(\bar{x}_{1,c}, \bar{x}_{2,c}) = (1/4, 0)$ and a fixed time $t_0 = \pi/8$.

```
make transport_tensor_dg
mkgeo_grid -t 80 -a -0.5 -b 0.5 -c -0.5 -d 0.5 > square2.geo
./transport_tensor_dg square2 P1d > square2.field
field square2.field -comp 00 -elevation
field square2.field -comp 01 -elevation
```

The computation could also be performed with any `Pkd`, with $k \geqslant 0$.

Error analysis

The file '`transport_tensor_error_dg.cc`' implement the computation of the error between the approximate solution $\boldsymbol{\sigma}_h$ and the exact one $\boldsymbol{\sigma}$. This file it is not listed here but is available in the

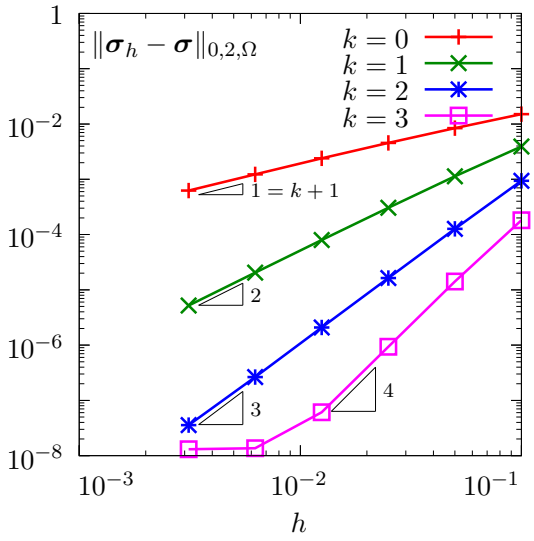


Figure 5.12: Tranport tensor problem: convergence versus mesh size.

Rheolef example directory. The computation of the error is obtained by:

```
make transport_tensor_error_dg
./transport_tensor_error_dg < square2.field
```

The error is plotted on Fig. 5.12 for various mesh size h and polynomial order k: observe the optimality of the convergence properties. For $k = 4$ and on the finest mesh, the error saturates at about 10^{-8}, due to finite machine precision effects. In the next chapter, transport tensor approximation are applied to viscoelastic fluid flow computations.

5.3.2 The Oldroyd model

We consider the following viscoelastic fluid flow problem (see e.g. Saramito, 2016b, chap. 4):
(P): find τ, $\boldsymbol{u}$ and p defined in $]0,T[\times\Omega$ such that

$$
\begin{aligned}
We\,\frac{\mathscr{D}_a\tau}{\mathscr{D}t} + \tau - 2\alpha D(\boldsymbol{u}) &= 0 \ \text{ in } \ ]0,T[\times\Omega && \text{(5.13a)}\\
Re\left(\frac{\partial \boldsymbol{u}}{\partial t} + \boldsymbol{u}.\nabla\boldsymbol{u}\right) - \mathbf{div}\,(\tau + 2(1-\alpha)D(\boldsymbol{u}) - p.I) &= 0 \ \text{ in } \ ]0,T[\times\Omega && \text{(5.13b)}\\
-\mathrm{div}\,\boldsymbol{u} &= 0 \ \text{ in } \ ]0,T[\times\Omega && \text{(5.13c)}\\
\tau &= \tau_\Gamma \ \text{ on } \ ]0,T[\times\partial\Omega_- && \text{(5.13d)}\\
\boldsymbol{u} &= \boldsymbol{u}_\Gamma \ \text{ on } \ ]0,T[\times\partial\Omega && \text{(5.13e)}\\
\tau(0) = \tau_0 \ \text{ and } \ \boldsymbol{u}(0) &= \boldsymbol{u}_0 \ \text{ in } \ \Omega && \text{(5.13f)}
\end{aligned}
$$

where τ_0, $\boldsymbol{u}_0$, τ_Γ and $\boldsymbol{u}_\Gamma$ are given. The first equation corresponds to a generalized Oldroyd model [Oldroyd, 1950]: when $a = -1$ we obtain the Oldroyd-A model, when $a = 1$, the Oldroyd-B model, and when $a \in]-1,1[$ a generalization of these two models. The dimensionless number $We \geqslant 0$ is the Weissenberg number: this is the main parameter for this problem. The dimensionless Reynolds number $Re \geqslant 0$ is often chosen small: as such fluids are usually slow, the $\boldsymbol{u}.\nabla\boldsymbol{u}$ inertia term is also neglected here for simplicity. The parameter $\alpha \in [0,1]$ represent a retardation. When $\alpha = 1$ we obtain the Maxwell model, that is a reduced version of the Oldroyd one. The total Cauchy

stress tensor is expressed by:

$$\boldsymbol{\sigma}_{\rm tot} \;=\; -p\,\boldsymbol{I} + 2(1-\alpha)D(\boldsymbol{u}) + \boldsymbol{\tau} \tag{5.14}$$

5.3.3 The θ-scheme algorithm

The θ-scheme is considered for the time discretization [Saramito, 1994] (see also Saramito, 2016b, chap. 4): this leads to a semi-implicit splitting algorithm that defines a sequence $(\tau^{(n)}, \boldsymbol{u}^{(n)}, p^n)_{n\geqslant 0}$ as

- $n=0$: set $(\tau^{(0)}, \boldsymbol{u}^{(0)}) = (\tau_0, \boldsymbol{u}_0)$ and p^0 arbitrarily chosen.
- $n \geqslant 0$: let $(\tau^{(n)}, \boldsymbol{u}^{(n)})$ being known, then $(\tau^{(n+1)}, \boldsymbol{u}^{(n+1)}, p^{(n+1)})$ is defined in three sub-steps.
  * sub-step 1: compute explicitly:

$$\begin{aligned}
\gamma &= \boldsymbol{u}^{(n)}.\nabla\tau^{(n)} + \tau^{(n)} M_a\left(\boldsymbol{u}^{(n)}\right) + M_a^T\left(\boldsymbol{u}^{(n)}\right)\tau^{(n)} \\
\tilde{\boldsymbol{f}} &= \lambda\boldsymbol{u}^{(n)} + \mathbf{div}\left(c_1\tau^{(n)} + c_2\gamma\right)
\end{aligned}$$

  where

$$c_1 = \frac{We}{We+\theta\Delta t} \quad\text{and}\quad c_2 = -\frac{We\theta\Delta t}{We+\theta\Delta t}$$

  Then determine $(\boldsymbol{u}^{(n+\theta)}, p^{(n+\theta)})$ such that

$$\lambda\boldsymbol{u}^{(n+\theta)} - \operatorname{div}\left(2\eta D\left(\boldsymbol{u}^{(n+\theta)}\right)\right) + \nabla p^{(n+\theta)} = \tilde{\boldsymbol{f}} \;\text{ in }\; \Omega \tag{5.15a}$$

$$-\operatorname{div}\boldsymbol{u}^{(n+\theta)} = 0 \;\text{ in }\; \Omega \tag{5.15b}$$

$$\boldsymbol{u} = \boldsymbol{u}_\Gamma((n+\theta)\Delta t) \;\text{ on }\; \partial\Omega \tag{5.15c}$$

  and finaly, compute explictely

$$\tau^{(n+\theta)} = c_1\tau^{(n)} + c_2\gamma + 2c_3 D\left(\boldsymbol{u}^{(n+\theta)}\right) \tag{5.15d}$$

  where

$$\lambda = \frac{Re}{\theta\Delta t},\quad \eta = \frac{(1-\alpha)We+\theta\Delta t}{We+\theta\Delta t} \quad\text{and}\quad c_3 = \frac{\alpha\theta\Delta t}{We+\theta\Delta t}$$

  * sub-step 2: $(\tau^{(n+\theta)}, \boldsymbol{u}^{(n+\theta)})$ being known, compute explictely

$$\begin{aligned}
\boldsymbol{u}^{(n+1-\theta)} &= \frac{1-\theta}{\theta}\boldsymbol{u}^{(n+\theta)} - \frac{1-2\theta}{\theta}\boldsymbol{u}^{(n)} \\
\xi &= c_4\tau_{n+\theta} + 2c_5 D\left(\boldsymbol{u}^{(n+\theta)}\right)
\end{aligned}$$

  and then find $\tau^{(n+1-\theta)}$ such that

$$\boldsymbol{u}^{(n+1-\theta)}.\nabla\tau^{(n+1-\theta)} + \tau^{(n+1-\theta)} M_a\left(\boldsymbol{u}^{(n+1-\theta)}\right) + M_a^T\left(\boldsymbol{u}^{(n+1-\theta)}\right)\tau^{(n+1-\theta)} + \nu\tau^{(n+1-\theta)} = \xi \;\text{ in }\; \Omega \tag{5.16a}$$

$$\tau^{(n+1-\theta)} = \tau_\Gamma((n+1-\theta)\Delta t) \;\text{ on }\; \partial\Omega_- \tag{5.16b}$$

  where

$$\nu = \frac{1}{(1-2\theta)\Delta t},\quad c_4 = \frac{1}{(1-2\theta)\Delta t} - \frac{1}{We} \quad\text{and}\quad c_5 = \frac{\alpha}{We}$$

* sub-step 3 is obtained by replacing n and $n+\theta$ by $n+1-\theta$ and $n+1$, respectively.

Thus, sub-step 1 and 2 reduces to two similar generalized Stokes problems while sub-step 3 involves a stress transport problem. Here $\Delta t > 0$ and $\theta \in]0, 1/2[$ are numerical parameters. A good choice is $\theta = 1 - 1/\sqrt{2}$ [Saramito, 1997]. This algorithm was first proposed by Saramito [1990] and extended [Saramito, 1995] to Phan-Thien and Tanner viscoelastic models. See also [Singh and Leal, 1993] for another similar approach in the context of FENE viscoelastic models. Scurtu [2005] presented some benchmarks of this algorithm while Chrispell et al. [2009] presented a numerical analysis of its convergence properties. The main advantage of this time-depend algorithm is its flexibility: while most time-dependent splitting algorithms for viscoelastic are limited to $\alpha \leqslant 1/2$ (see e.g. Pan et al., 2009), here the full range $\alpha \in]0, 1]$ is available.

Let us introduce the finite element spaces:

$$\begin{aligned} T_h &= \{\boldsymbol{\tau}_h \in (L^2(\Omega))^{d\times d}_s \ ; \ \boldsymbol{\tau}_{h|K} \in (P_1)^{d\times d}_s, \ \forall K \in \mathscr{T}_h\} \\ X_h &= \{\boldsymbol{v}_h \in (H^1(\Omega))^d \ ; \ \boldsymbol{\tau}_{h|K} \in (P_2)^d, \ \forall K \in \mathscr{T}_h\} \\ Q_h &= \{q_h \in L^2(\Omega) \ ; \ q_{h|K} \in P_1, \ \forall K \in \mathscr{T}_h\} \end{aligned}$$

Note the discontinuous approximation of pressure: it presents a major advantage, as $\operatorname{div}(X_h) \subset Q_h$, it leads to an exact *divergence-free* approximation of the velocity: for any field $\boldsymbol{v}_h \in X_h$ satisfying $\int_\Omega q_h \operatorname{div}(\boldsymbol{v}_h)\,\mathrm{d}x = 0$ for all $q_h \in Q_h$, we have $\operatorname{div} \mathbf{v}_h = 0$ point-wise, everywhere in Ω. The pair (X_h, Q_h) is known as the Scott and Vogelius [1985] lowest-order finite element approximation. This is a major advantage when dealing with a transport equation. The only drawback is that the pair (X_h, Q_h) does not satisfy the inf-sup condition for an arbitrary mesh. There exists a solution to this however: Arnold and Qin [1992] proposed a macro element technique applied to the mesh that allows satisfying the inf-sup condition: for any triangular finite element mesh, it is sufficient to split each triangle in three elements from its barycenter (see also Saramito, 2014). Note that the macro element technique extends to quadrilateral meshes [Arnold and Qin, 1992] and to the three-dimensional case [Zhang, 2005]. By this way, the approximate velocity field satisfies exactly the incompressibility constraint: this is an essential property for the operator splitting algorithm to behave correctly, combining stress transport equation with a divergence-free velocity approximation.

The tensor transport term is discretized as in the previous chapter, by using the t_h trilinear form introduced in (5.12) page 231. The bilinear forms b, c, d are defined by:

$$\begin{aligned} b(\boldsymbol{\sigma}_h, \boldsymbol{v}_h) &= \int_\Omega \boldsymbol{\sigma}_h : D(\boldsymbol{v}_h)\,\mathrm{d}x \\ c(\boldsymbol{u}_h, \boldsymbol{v}_h) &= \int_\Omega D(\boldsymbol{u}_h) : D(\boldsymbol{v}_h)\,\mathrm{d}x \\ d(\boldsymbol{u}_h, \boldsymbol{q}_h) &= \int_\Omega \operatorname{div}(\boldsymbol{u}_h)\, q_h\,\mathrm{d}x \end{aligned}$$

Let T, B, C, D and M be the discrete operators (i.e. the matrix) associated to the forms t_h, b, c, d and the L^2 scalar product in T_h. Assume that a stationnary state is reached for the discretized algorithm. Then, (5.15a)-(5.15d) writes

$$\begin{aligned} We\,T\tau_h + M\tau_h - 2\alpha B^T u_h &= 0 \\ B\left(c_1\tau_h + M^{-1}T\tau_h\right) + 2\eta C u_h + D^T p_h &= 0 \\ Du_h &= 0 \end{aligned}$$

Note that (5.16a)-(5.16b) reduces also to the first equation of the previous system. Expanding the coefficients, combining the two previous equations, and using $C = BM^{-1}B^T$, we obtain the system characterizing the stationary solution of the discrete version of the algorithm:

$$\begin{aligned} We\,T\tau_h + M\tau_h - 2\alpha B^T u_h &= 0 \\ B\tau_h + 2(1-\alpha) C u_h + D^T p_h &= 0 \\ Du_h &= 0 \end{aligned}$$

Note that this is exactly the discretized version of the stationary equation[1]. In order to check that the solution reaches a stationary state, the residual terms of this stationary equations are computed at each iteration, together with the relative error between two iterates.

File 5.20: `oldroyd_theta_scheme.h`

```
template<class Problem>
struct oldroyd_theta_scheme {
  oldroyd_theta_scheme();
  void initial (const geo& omega, field& tau_h, field& uh, field& ph,
                string restart);
  bool solve (field& tau_h, field& uh, field& ph);
protected:
  void step      (const field& tau_h0, const field& uh0, const field& ph0,
                  field& tau_h, field& uh, field& ph) const;
  void sub_step1 (const field& tau_h0, const field& uh0, const field& ph0,
                  field& tau_h, field& uh, field& ph) const;
  void sub_step2 (const field& uh0, const field& tau_h1, const field& uh1,
                  field& tau_h, field& uh) const;
  Float residue  (field& tau_h, field& uh, field& ph) const;
  void reset (const geo& omega);
  void update_transport_stress (const field& uh) const;
public:
  Float We, alpha, a, Re, delta_t, tol;
  size_t max_iter;
protected:
  space Th, Xh, Qh;
  form b, c, d, mt, inv_mt, mu, mp;
  mutable form th;
  mutable field thb;
  Float theta, lambda, eta, nu, c1, c2, c3, c4, c5;
  problem_mixed stokes;
};
#include "oldroyd_theta_scheme1.h"
#include "oldroyd_theta_scheme2.h"
#include "oldroyd_theta_scheme3.h"
```

[1]The original θ-scheme presented by Saramito [1994] has an additional relaxation parameter ω. The present version correspond to $\omega = 1$. When $\omega \neq 1$, the stationary solution still depends slightly upon Δt, as the stationary system do not simplifies completely.

File 5.21: `oldroyd_theta_scheme1.h`

```
template<class P>
oldroyd_theta_scheme<P>::oldroyd_theta_scheme()
: We(0), alpha(8./9), a(1), Re(1), delta_t(0.025), tol(1e-6), max_iter(500),
  Th(), Xh(), Qh(), b(), c(), d(), mt(), inv_mt(), mu(), mp(), th(), thb(),
  theta(), lambda(), eta(), nu(), c1(), c2(), c3(), c4(), c5(), stokes() {}
template<class P>
void oldroyd_theta_scheme<P>::reset(const geo& omega) {
  Th = space (omega, "P1d", "tensor");
  Xh = P::velocity_space (omega, "P2");
  Qh = space (omega, "P1d");
  theta = 1-1/sqrt(2.);
  lambda = Re/(theta*delta_t);
  eta = ((1 - alpha)*We + theta*delta_t)/(We + theta*delta_t);
  nu = 1/((1-2*theta)*delta_t);
  c1 = We/(We + theta*delta_t);
  c2 = - We*theta*delta_t/(We + theta*delta_t);
  c3 = alpha*theta*delta_t/(We + theta*delta_t);
  c4 = 1/((1-2*theta)*delta_t) - 1/We;
  c5 = alpha/We;
  trial u (Xh), tau(Th), p (Qh);
  test  v (Xh), xi (Th), q (Qh);
  mt = integrate (ddot(tau,xi));
  mu = integrate (dot(u,v));
  mp = integrate (p*q);
  integrate_option iopt;
  iopt.invert = true;
  inv_mt = integrate (ddot(tau,xi), iopt);
  b  = integrate (-ddot(tau,D(v)));
  c = integrate (lambda*dot(u,v) + 2*eta*ddot(D(u),D(v)));
  d  = integrate (-div(u)*q);
  stokes = problem_mixed (c, d);
  stokes.set_metric (mp);
}
```

File 5.22: oldroyd_theta_scheme2.h

```
template <class P>
bool oldroyd_theta_scheme<P>::solve(field& tau_h, field& uh, field& ph) {
  reset (uh.get_geo());
  field tau_h0 = tau_h, uh0 = uh, ph0 = ph;
  derr << "# n t rel_err residue lambda_min" << endl;
  Float r = residue (tau_h, uh, ph);
  Float rel_err = 0;
  derr << "0 0 0 " << r << endl;
  for (size_t n = 1; n <= max_iter; ++n) {
    step (tau_h0, uh0, ph0, tau_h, uh, ph);
    Float rel_err_prec = rel_err, r_prec = r;
    r = residue (tau_h, uh, ph);
    rel_err = field(tau_h-tau_h0).max_abs() + field(uh-uh0).max_abs();
    derr << n << " " << n*delta_t << " " << rel_err << " " << r << endl;
    if (rel_err < tol) return true;
    if (rel_err_prec != 0 && ((rel_err > 10*rel_err_prec && r > 10*r_prec) ||
                              (rel_err > 1e5 && r > 1e5) )) return false;
    tau_h0 = tau_h; uh0 = uh; ph0 = ph;
  }
  return (rel_err < sqrt(tol));
}
template <class P>
void oldroyd_theta_scheme<P>::initial (
    const geo& omega, field& tau_h, field& uh, field& ph, string restart) {
  reset (omega);
  ph = field(Qh,0);
  if (restart == "") {
    uh = P::velocity_field (Xh);
    trial u (Xh); test v (Xh), xi(Th);
    form c0  = integrate (2*ddot(D(u),D(v)));
    problem_mixed s0 (c0, d);
    s0.set_metric(mp);
    s0.solve (field(Xh,0), field(Qh,0), uh, ph);
    field Duh = inv_mt*integrate(ddot(D(uh),xi));
    tau_h = 2*alpha*Duh;
  } else {
    tau_h = field(Th);
    uh    = field(Xh);
    idiststream in (restart, "field");
    in >> catchmark("tau") >> tau_h
       >> catchmark("u")   >> uh
       >> catchmark("p")   >> ph;
  }
}
template <class P>
void oldroyd_theta_scheme<P>::step (
    const field& tau_h0, const field& uh0, const field& ph0,
    field& tau_h, field& uh, field& ph) const {
  field tau_h1 = tau_h0, uh1 = uh0, ph1 = ph0;
  sub_step1 (tau_h0, uh0, ph0, tau_h1, uh1, ph1);
  field tau_h2 = tau_h1, uh2 = uh1;
  sub_step2 (uh0, tau_h1, uh1, tau_h2, uh2);
  sub_step1 (tau_h2, uh2, ph1, tau_h, uh, ph);
}
template <class P>
Float
oldroyd_theta_scheme<P>::residue(field& tau_h, field& uh, field& ph) const{
  update_transport_stress (uh);
  test xi (Th);
  field Duh = inv_mt*integrate(ddot(D(uh),xi));
  field gh = 2*Duh;
  field rt = We*(th*tau_h-thb) + integrate (ddot(tau_h - alpha*gh, xi));
  field ru = b*(tau_h + (1-alpha)*gh) - d.trans_mult(ph);
  ru.set_b() = 0;
  field rp = d*uh;
  return rt.u().max_abs() + ru.u().max_abs() + rp.u().max_abs();
}
```

File 5.23: oldroyd_theta_scheme3.h

```
template <class P>
void oldroyd_theta_scheme<P>::sub_step1 (
  const field& tau_h0, const field& uh0, const field& ph0,
  field& tau_h, field& uh, field& ph) const
{
  update_transport_stress (uh0);
  field gamma_h = inv_mt*(th*tau_h0 - thb);
  test v (Xh), xi (Th);
  field lh = lambda*integrate (dot(uh0,v))
           + b*(c1*tau_h0 + c2*gamma_h);
  ph = ph0;
  uh.set_u() = uh0.u();
  stokes.solve (lh, field(Qh,0), uh, ph);
  field Duh = inv_mt*integrate(ddot(D(uh),xi));
  tau_h = c1*tau_h0 + c2*gamma_h + 2*c3*Duh;
}
template <class P>
void oldroyd_theta_scheme<P>::sub_step2 (
  const field& uh0,
  const field& tau_h1, const field& uh1,
  field& tau_h, field& uh) const
{
  uh = (1-theta)/theta*uh1 - (1-2*theta)/theta*uh0;
  test xi (Th);
  if (We == 0) {
    field Duh = inv_mt*integrate(ddot(D(uh),xi));
    tau_h = 2*alpha*Duh;
    return;
  }
  update_transport_stress (uh);
  form  th_nu = th + nu*mt;
  typename P::tau_upstream tau_up (Th.get_geo(), We, alpha);
  field lh = integrate (ddot(c4*tau_h1 + 2*c5*D(uh1),xi))
           + integrate ("boundary",
                        max(0, -dot(uh,normal()))*ddot(tau_up,xi));
  problem transport (th_nu);
  transport.solve (lh, tau_h);
}
template <class P>
void
oldroyd_theta_scheme<P>::update_transport_stress (const field& uh) const {
  typename P::tau_upstream tau_up (Th.get_geo(), We, alpha);
  trial tau (Th); test xi (Th);
  auto ma = 0.5*((1-a)*grad(uh) - (1+a)*trans(grad(uh)));
  auto beta_a = tau*ma + trans(ma)*tau;
  th  = integrate (ddot(grad_h(tau)*uh + beta_a,xi))
      + integrate ("boundary", max(0, -dot(uh,normal()))*ddot(tau,xi))
      + integrate ("internal_sides",
           - dot(uh,normal())*ddot(jump(tau),average(xi))
           + 0.5*abs(dot(uh,normal()))*ddot(jump(tau),jump(xi)));
  thb = integrate ("boundary", max(0, -dot(uh,normal()))*ddot(tau_up,xi));
}
```

5.3.4 Flow in an abrupt ontraction

Fig. 5.13 represents the contraction flow geometry. Let us denote by Γ_u, Γ_d, Γ_w and Γ_s the upstream, downstream, wall and symmetry axis boundary domains, respectively. This geometry has already been studied in section 2.2.2, in the context of a Newtonian fluid. Here, the fluid is more complex and additional boundary conditions on the upstream domain are required for the extra-stress tensor $\boldsymbol{\tau}$.

For the geometry, we assume that the lengths L_u and L_d are sufficiently large for the Poiseuille flows to be fully developped at upstream and downstream. Assuming also $a = 1$, i.e. the Oldroyd-B model, the boundary conditions at upstream are explicitely known as the solution of the fully

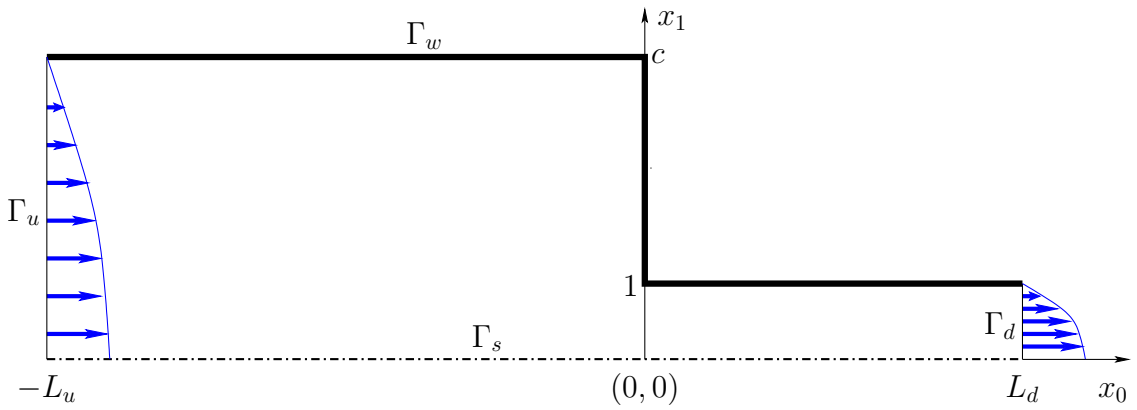


Figure 5.13: The Oldroyd problem in the abrupt contraction: shematic view of the flow domain.

Poiseuille flow for a plane pipe with half width c:

$$\begin{aligned}
u_0(x_1) &= \bar{u}\left(1-\left(\frac{x_1}{c}\right)^2\right) \\
\dot{\gamma}(x_1) &= u_0'(x_1) = -\frac{2\bar{u}\,x_1}{c^2} \\
\tau_{00}(x_1) &= 2\alpha We\,\dot{\gamma}^2(x_1) \\
\tau_{01}(x_1) &= \tau_{10}(x_1) = \alpha\,\dot{\gamma}(x_1) \\
\tau_{11}(x_1) &= 0
\end{aligned}$$

where $\bar{u}$ denotes the maximal velocity of the Poiseuille flow. Without loss of generality, thanks to a dimensional analysis, it can be adjused with the contraction ratio c for obtaining a flow rate equal to one:

$$\bar{u} = \begin{cases} 3/(2c) & \text{for a planar geometry} \\ 4/c^2 & \text{for an axisymmetric one} \end{cases}$$

File 5.24: `oldroyd_contraction.h`

```
#include "contraction.h"
struct oldroyd_contraction: contraction {
  struct tau_upstream: base {
    tau_upstream (geo omega, Float We1, Float alpha1)
      : base(omega), We(We1), alpha(alpha1) {}
    tensor operator() (const point& x) const {
      tensor tau;
      Float dot_gamma = - 2*base::umax*x[1]/sqr(base::c);
      tau(0,0) =     2*alpha*We*sqr(dot_gamma);
      tau(0,1) = tau(1,0) = alpha*dot_gamma;
      tau(1,1) = 0;
      return tau;
    }
    Float We, alpha;
  };
};
```

The class `contraction`, already used for Newtonian fluids, is here reused and extended with the boundary condition function `tau_upstream`, as a derived class `oldroyd_contraction`. We are now able to write the `main` program for solving a viscoelastic fluid flow problem in a contraction.

File 5.25: `oldroyd_contraction.cc`

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
#include "oldroyd_theta_scheme.h"
#include "oldroyd_contraction.h"
int main(int argc, char**argv) {
  environment rheolef (argc, argv);
  cin >> noverbose;
  oldroyd_theta_scheme<oldroyd_contraction> pb;
  geo omega (argv[1]);
  Float We_incr     = (argc > 2) ? atof(argv[2]) : 0.1;
  Float We_max      = (argc > 3) ? atof(argv[3]) : 0.1;
  Float delta_t0    = (argc > 4) ? atof(argv[4]) : 0.005;
  string restart    = (argc > 5) ?      argv[5]  : "";
  pb.tol            = 1e-3;
  pb.max_iter       = 50000;
  pb.alpha          = 8./9;
  pb.a              = 1;
  pb.delta_t        = delta_t0;
  Float delta_t_min = 1e-5;
  Float We_incr_min = 1e-5;
  dout << catchmark("alpha") << pb.alpha << endl
       << catchmark("a")     << pb.a     << endl;
  branch even ("We", "tau", "u", "p");
  field tau_h, uh, ph;
  pb.initial (omega, tau_h, uh, ph, restart);
  dout << even (pb.We, tau_h, uh, ph);
  bool ok = true;
  do {
    if (ok) pb.We += We_incr;
    derr << "# We = " << pb.We << " delta_t = " << pb.delta_t << endl;
    field tau_h0 = tau_h, uh0 = uh, ph0 = ph;
    ok = pb.solve (tau_h, uh, ph);
    if (ok) {
      dout << even (pb.We, tau_h, uh, ph);
    } else {
      pb.delta_t /= 2;
      tau_h = tau_h0; uh = uh0; ph = ph0;
      if (pb.delta_t < delta_t_min) {
        derr << "# solve failed: decreases We_incr and retry..." << endl;
        pb.delta_t = delta_t0;
        We_incr /= 2;
        pb.We -= We_incr;
        if (We_incr < We_incr_min) break;
      } else {
        derr << "# solve failed: decreases delta_t and retry..." << endl;
      }
    }
    derr << endl << endl;
  } while (true);
}
```

The splitting element technique for the Scott-Vogelius element is implemented as an option by the command `mkgeo_contraction`. This file it is not listed here but is available in the **Rheolef** example directory. The mesh generation for an axisymmetric contraction writes:

```
mkgeo_contraction 3 -c 4 -zr -Lu 20 -Ld 20 -split
geo contraction.geo
```

This command generates a mesh for the axisymmetric 4:1 abrupt contraction with upstream and downstream length $L_u = L_d = 20$. Such high lengths are required for the Poiseuille flow to be fully developped at upstream and downstream for large values of We. The 3 first argument of `mkgeo_contraction` is a number that characterizes the mesh density: when increasing, the average edge length decreases. Then, the program is started:

```
make oldroyd_contraction
```

```
mpirun -np 8 ./oldroyd_contraction contraction.geo 0.1 10 0.01 > contraction.branch
```

The program computes stationnary solutions by increasing We with the time-dependent algorithm. It performs a continuation algorithm, using solution at a lowest We as initial condition. The reccurence starts from an the Newtonian solution associated to $We = 0$. The others model parameters for this classical benchmark are fixed here as $\alpha = 8/9$ and $a = 1$ (Oldroyd-B model). The computation can take a while as there are two loops: one outer, on We, and the other inner on time, and there are two generalized Stokes subproblem and one tensorial transport one to solve at each time iteration. The inner time loop stops when the relative error is small enough. Thus, the parallel run, when available, is a major advantage: it is obtained by adding `mpirun -np 8` at the begining of the command line. Recall that the time scheme is conditionnaly stable: the time step should be small enough for the algorithm to converge to a stationnary solution. When the solver fails, it restarts with a smaller time step. Note that the present solver can be dramatically improved: by using a Newton method, as shown by Saramito [2014], it is possible to directly reach the stationnary solution, but such more sophisticated implementation is out of the scope of the present documentation.

The visualization of the stream function writes:

```
branch contraction.branch -toc
branch contraction.branch -extract 3 -branch > contraction-We-0.3.field
make streamf_contraction
field -mark u contraction-We-0.3.field -field | ./streamf_contraction > psi.field
field psi.field -n-iso 15 -n-iso-negative 10
field psi.field -n-iso 15 -n-iso-negative 10 -bw
```

The file '`streamf_contraction.cc`' has already been studied in section 2.2.2, in the context of a Newtonian fluid. The result is shown on Fig. 5.14. The vortex growths with We: this is the major effet observed on this problem. Observe the two color maps representation: positive values of the stream functions are associated to the vortex and negatives values to the main flow region. The last command is a black-and-white variant view.

The vortex activity is obtained by:

```
field psi.field -max
```

Recall that the minimal value of the stream function is -1, thanks to the dimensionless procedure used here.

Cuts along the axis of symmetry are obtained by:

```
field contraction-We-0.3.field -domain axis -mark u   -comp 0  -elevation -gnuplot
field contraction-We-0.3.field -domain axis -mark tau -comp 00 -elevation -gnuplot
field contraction-We-0.3.field -domain axis -mark tau -comp 11 -elevation -gnuplot
field contraction-We-0.3.field -domain axis -mark tau -comp 22 -elevation -gnuplot
```

These cuts are plotted on Fig. 5.16. Observe the overshoot of the velocity along the axis when $We > 0$ while this effect is not perceptible when $We = 0$. Also, the normal extra stress τ_{zz} growth dramatically in the entry region.

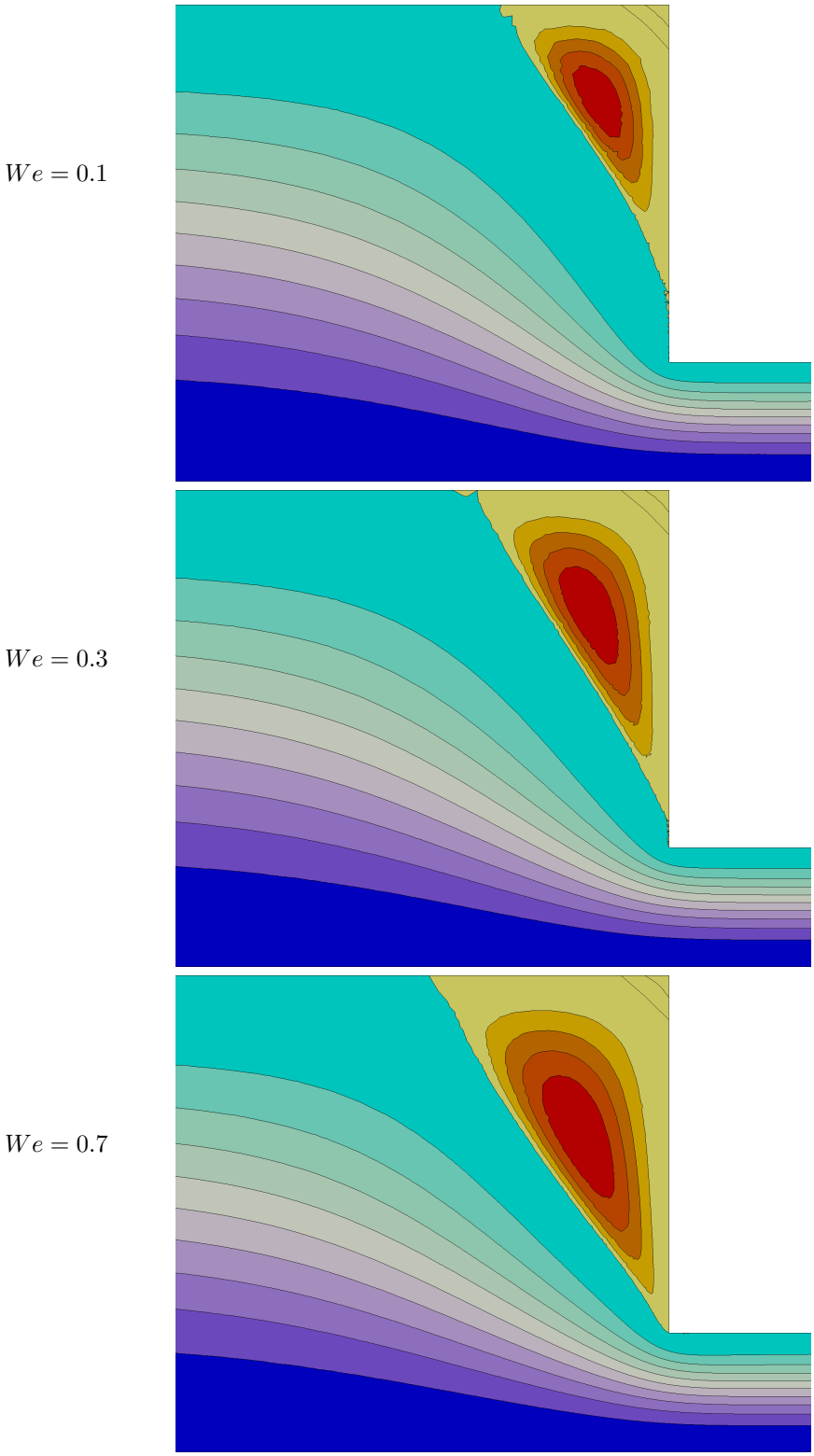


Figure 5.14: The Oldroyd problem in the axisymmetric contraction: stream function for $We = 0.1$, 0.3 and 0.7, from top to bottom.

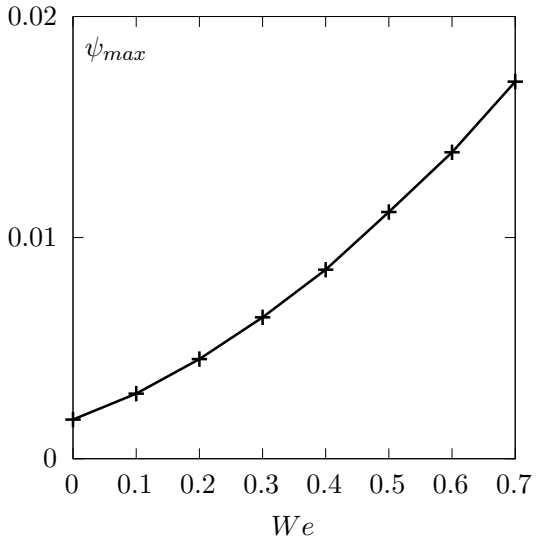


Figure 5.15: The Oldroyd-B problem in the axisymmetric contraction: vortex activity vs We ($\alpha = 8/9$).

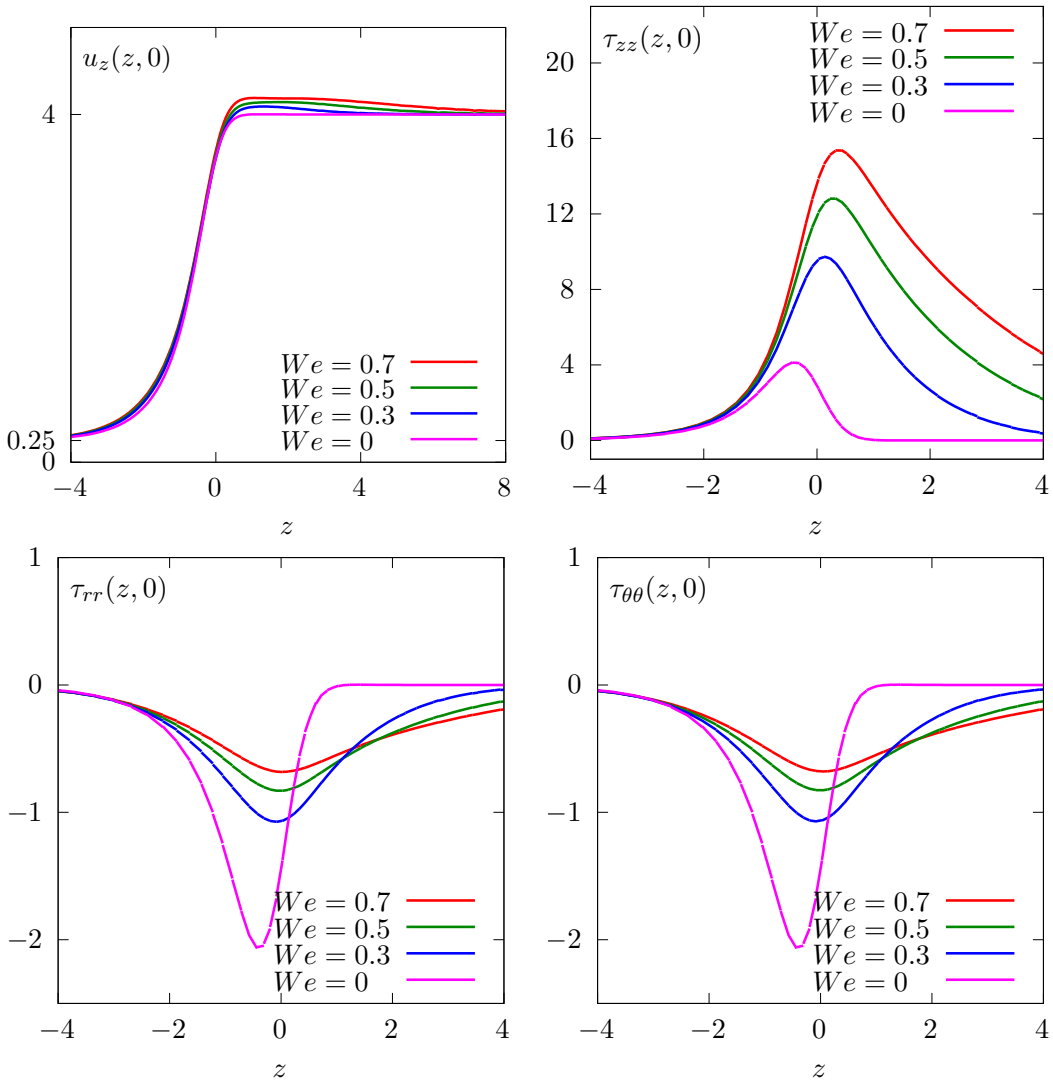


Figure 5.16: The Oldroyd problem in the axisymmetric contraction: velocity and stress components along the axis.

5.3.5 [New] Stress diffusion

The tensor diffusion variant of a viscoelastic model introduces an additional second order diffusive term in the differential equation for $\boldsymbol{\sigma}$, see section 5.3.1. It develops nice theoretical properties, see e.g. Lukáčová-Medvidová et al. [2017], Málek et al. [2018]. Let us consider the following variant of the tensor transport problem previously introduced in section 5.3.1, page 230, by adding a diffusion term:

(P): *find* $\boldsymbol{\sigma}$, *defined in* $]0,T[\times\Omega$, *such that*

$$\begin{aligned}\frac{\mathscr{D}_a\boldsymbol{\sigma}}{\mathscr{D}t} + \nu\boldsymbol{\sigma} - \varepsilon\Delta\boldsymbol{\sigma} &= \boldsymbol{\chi} \text{ in }]0,T[\times\Omega \\ \boldsymbol{\sigma} &= \boldsymbol{\sigma}_\Gamma \text{ on }]0,T[\times\partial\Omega_- \\ \boldsymbol{\sigma}(0) &= \boldsymbol{\sigma}_0 \text{ in } \Omega\end{aligned}$$

where $\varepsilon \geqslant 0$ is the diffusion parameter. Observe that when $\varepsilon = 0$, this problem reduces to the usual one, without diffusion, as introduced in section 5.3.1.

The steady version of the tensor transport problem writes:

(S): *find* $\boldsymbol{\sigma}$, *defined in* Ω, *such that*

$$\begin{aligned}\big(\mathbf{u}.\nabla\big)\boldsymbol{\sigma} + \boldsymbol{\sigma}\boldsymbol{g}_a(\mathbf{u} + \boldsymbol{g}_a^T(\mathbf{u})\boldsymbol{\sigma}\big) + \nu\boldsymbol{\sigma} - \varepsilon\Delta\boldsymbol{\sigma} &= \boldsymbol{\chi} \text{ in } \Omega \\ \boldsymbol{\sigma} &= \boldsymbol{\sigma}_\Gamma \text{ on } \partial\Omega_-\end{aligned}$$

We introduce the forms defined for all $\boldsymbol{\sigma},\boldsymbol{\tau} \in H^1(\Omega)_s^{d\times d}$ by

$$\begin{aligned}a(\boldsymbol{\sigma},\boldsymbol{\tau}) &= \int_\Omega \big((\mathbf{u}.\nabla)\boldsymbol{\sigma} + \boldsymbol{\sigma}\boldsymbol{g}_a(\mathbf{u}) + \boldsymbol{g}_a^T(\mathbf{u})\boldsymbol{\sigma} + \nu\,\boldsymbol{\sigma}\big) : \boldsymbol{\tau}\,\mathrm{d}x + \int_{\partial\Omega} \max\left(0,-\mathbf{u}.\mathbf{n}\right)\boldsymbol{\sigma} : \boldsymbol{\tau}\,\mathrm{d}s \\ &\quad + \int_\Omega \varepsilon\nabla\boldsymbol{\sigma} : \nabla\boldsymbol{\tau}\,\mathrm{d}x \\ l(\boldsymbol{\tau}) &= \int_\Omega \boldsymbol{\chi} : \boldsymbol{\tau}\,\mathrm{d}x + \int_{\partial\Omega} \max\left(0,-\mathbf{u}.\mathbf{n}\right)\boldsymbol{\sigma}_\Gamma : \boldsymbol{\tau}\,\mathrm{d}s\end{aligned}$$

Then, the variational formulation of the steady problem writes:

(FV): *find* $\boldsymbol{\sigma} \in X$ *such that*

$$a(\boldsymbol{\sigma},\boldsymbol{\tau}) = l(\boldsymbol{\tau}),\ \forall\boldsymbol{\tau} \in H^1(\Omega)_s^{d\times d}$$

The discontinuous finite element space is then defined by:

$$X_h = \{\boldsymbol{\tau}_h \in H^1(\Omega)_s^{d\times d};\boldsymbol{\tau}_{h|K} \in P_k,\ \forall K \in \mathcal{T}_h\}$$

where $k \geqslant 0$ is the polynomial degree. The discrete version a_h of the bilinear form a is defined for all $\boldsymbol{\sigma}_h,\boldsymbol{\tau}_h \in X_h$ by:

$$\begin{aligned}a_h(\boldsymbol{\sigma}_h,\boldsymbol{\tau}_h) &= t_h(\boldsymbol{u};\,\boldsymbol{\sigma}_h,\boldsymbol{\tau}_h) + \varepsilon d_h(\boldsymbol{u};\,\boldsymbol{\sigma}_h,\boldsymbol{\tau}_h) + \nu\int_\Omega \boldsymbol{\sigma}_h : \boldsymbol{\tau}_h\,\mathrm{d}x \\ t_h(\boldsymbol{u};\,\boldsymbol{\sigma}_h,\boldsymbol{\tau}_h) &= \int_\Omega \big((\mathbf{u}.\nabla_h)\boldsymbol{\sigma}_h + \boldsymbol{\sigma}\boldsymbol{g}_a(\mathbf{u}) + \boldsymbol{g}_a^T(\mathbf{u})\boldsymbol{\sigma}\big) : \boldsymbol{\tau}_h\,\mathrm{d}x + \int_{\partial\Omega} \max\left(0,-\mathbf{u}.\mathbf{n}\right)\boldsymbol{\sigma}_h : \boldsymbol{\tau}_h\,\mathrm{d}s \\ &\quad + \sum_{S\in\mathscr{S}_h^{(i)}} \int_S \Big(-\,\mathbf{u}.\mathbf{n}\,[\![\boldsymbol{\sigma}_h]\!] : \{\!\{\boldsymbol{\tau}_h\}\!\} + \frac{\alpha}{2}|\mathbf{u}.\mathbf{n}|\,[\![\boldsymbol{\sigma}_h]\!] : [\![\boldsymbol{\tau}_h]\!]\Big)\,\mathrm{d}s \\ d_h(\boldsymbol{u};\,\boldsymbol{\sigma}_h,\boldsymbol{\tau}_h) &= \int_\Omega \nabla_h\boldsymbol{\sigma}_h : \nabla_h\boldsymbol{\tau}_h\,\mathrm{d}x \\ &\quad + \sum_{S\in\mathscr{S}_h} \int_S \left(\kappa_s[\![\boldsymbol{\sigma}_h]\!] : [\![\boldsymbol{\tau}_h]\!] - [\![\boldsymbol{\sigma}_h]\!] : \{\!\{\nabla_h\boldsymbol{\tau}_h\boldsymbol{n}\}\!\} - [\![\boldsymbol{\tau}_h]\!] : \{\!\{\nabla_h\boldsymbol{\sigma}_h\boldsymbol{n}\}\!\}\right)\mathrm{d}s \\ l_h(\boldsymbol{\sigma}_h) &= \int_\Omega \boldsymbol{\chi} : \boldsymbol{\tau}\,\mathrm{d}x + \int_{\partial\Omega} \max\left(0,-\mathbf{u}.\mathbf{n}\right)\boldsymbol{\sigma}_\Gamma : \boldsymbol{\tau}\,\mathrm{d}s \\ &\quad + \varepsilon\int_{\partial\Omega} \left(\kappa_s\boldsymbol{\sigma}_\Gamma : \boldsymbol{\tau} - \boldsymbol{\sigma}_\Gamma : (\nabla_h\boldsymbol{\tau}\boldsymbol{n}) + \boldsymbol{\tau} : (\nabla\boldsymbol{\sigma}_\Gamma\boldsymbol{n})\right)\mathrm{d}s\end{aligned}$$

where $\boldsymbol{\sigma}_\Gamma$ is given and $\kappa_s = \beta\,\varpi_s$ is the stabilisation parameter, with $\beta = (k+1)(k+d)/d$ and ϖ_s is the penalisation, see section 4.3.1, page 167.

$(FV)_h$: *find* $\boldsymbol{\sigma}_h \in X_h$ *such that*

$$a_h(\boldsymbol{\sigma}_h, \boldsymbol{\tau}_h) \quad = \quad l_h(\boldsymbol{\tau}_h),\ \forall \boldsymbol{\tau}_h \in X_h$$

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

File 5.26: `diffusion_transport_tensor_dg.cc`

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
#include "diffusion_transport_tensor_exact.icc"
int main(int argc, char**argv) {
  environment rheolef (argc, argv);
  geo omega (argv[1]);
  space Xh (omega, argv[2], "tensor");
  Float alpha = (argc > 3) ? atof(argv[3]) : 1;
  Float nu    = (argc > 4) ? atof(argv[4]) : 3;
  Float t0    = (argc > 5) ? atof(argv[5]) : acos(-1.)/8;
  Float eps   = (argc > 6) ? atof(argv[6]) : 1e-2;
  Float a     = 0;
  size_t d = omega.dimension();
  size_t k = Xh.degree();
  Float eta = (k+1)*(k+d)/Float(d);
  trial sigma (Xh); test tau (Xh);
  tensor ma = 0.5*((1-a)*grad_u - (1+a)*trans(grad_u));
  auto beta_a = sigma*ma + trans(ma)*sigma;
  form  ah = integrate (ddot(grad_h(sigma)*u + beta_a + nu*sigma,tau)
                + eps*dddot(grad_h(sigma), grad_h(tau)))
           + integrate ("boundary",
                max(0, -dot(u,normal()))*ddot(sigma,tau))
           + integrate ("internal_sides",
                - dot(u,normal())*ddot(jump(sigma),average(tau))
                + 0.5*alpha*abs(dot(u,normal()))
                           *ddot(jump(sigma),jump(tau)))
           + integrate("sides",
                  eps*eta*penalty()*ddot(jump(sigma),jump(tau))
                - eps*ddot(jump(sigma), average(grad_h(tau  )*normal()))
                - eps*ddot(jump(tau)  , average(grad_h(sigma)*normal())));
  field lh = integrate (ddot(chi(eps,nu,t0),tau))
           + integrate ("boundary",
                  max(0, -dot(u,normal()))*ddot(sigma_g(nu,t0),tau)
                + eps*eta*penalty()*ddot(sigma_g(nu,t0),tau)
                - eps*ddot(sigma_g(nu,t0),grad_h(tau)*normal())
                + eps*ddot(grad_sigma_g(nu,t0)*normal(),tau));
  field sigma_h(Xh);
  problem p (ah);
  p.solve (lh, sigma_h);
  dout << catchmark("nu")    << nu << endl
       << catchmark("t0")    << t0 << endl
       << catchmark("sigma") << sigma_h;
}
```

Running the program

Let $d = 2$ and $\Omega =]-1/2, 1/2[^2$. We consider the rotating field $\mathbf{u} = (-x_2, x_1)$. A particular solution of the time-dependent problem with zero right-hand side is given by:

$$\boldsymbol{\sigma}(x,t) = \frac{1}{2}\exp\left\{-\nu t - \frac{(x_1 - x_{1,c}(t))^2 + (x_2 - x_{2,c}(t))^2}{r_0^2}\right\} \times \begin{pmatrix} 1+\cos(2t) & \sin(2t) \\ \sin(2t) & 1-\cos(2t) \end{pmatrix}$$

where $x_{1,c}(t) = \bar{x}_{1,c}\cos(t) - \bar{x}_{2,c}\sin(t)$ and $x_{2,c}(t) = \bar{x}_{1,c}\sin(t) + \bar{x}_{2,c}\cos(t)$ with $r_0 > 0$ and $(\bar{x}_{1,c}, \bar{x}_{2,c}) \in \mathbb{R}^2$. The form $_h$ is chosen with $\boldsymbol{\sigma}_\Gamma = \boldsymbol{\sigma}$. This exact solution is implemented in the file '`diffusion_transport_tensor_exact.icc`'. This file it is not listed here but is available

in the **Rheolef** example directory. For the steady problem, the right-hand side could be chosen as $\boldsymbol{\chi} = -\dfrac{\partial \boldsymbol{\sigma}}{\partial t}$ and then $t = t_0$ is fixed. The numerical tests correspond to $\nu = 3$, $r_0 = 1/10$, $(\bar{x}_{1,c}, \bar{x}_{2,c}) = (1/4, 0)$ and a fixed time $t_0 = \pi/8$.

```
make diffusion_transport_tensor_dg
mkgeo_grid -t 80 -a -0.5 -b 0.5 -c -0.5 -d 0.5 > square2.geo
./diffusion_transport_tensor_dg square2 P1d > square2.field
field square2.field -comp 00 -elevation
field square2.field -comp 01 -elevation
```

The computation could also be performed with any `Pkd`, with $k \geqslant 0$.

Error analysis

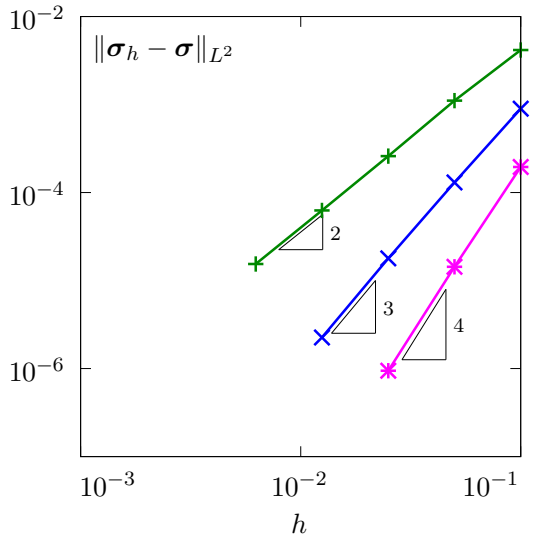


Figure 5.17: Diffusion-diffusion tensor problem: convergence versus mesh size.

The file '`diffusion_transport_tensor_error_dg.cc`' implement the computation of the error between the approximate solution $\boldsymbol{\sigma}_h$ and the exact one $\boldsymbol{\sigma}$. This file it is not listed here but is available in the **Rheolef** example directory. The computation of the error is obtained by:

```
make diffusion_transport_tensor_error_dg
./diffusion_transport_tensor_error_dg < square2.field
```

The error is plotted on Fig. 5.17 for various mesh size h and polynomial order k: observe the optimality of the convergence properties.

Chapter 6

[New] Hybrid discontinuous methods

This chapter is still in active development.

6.1 The Raviart-Thomas element

The aim of this chapter is to introduce to the Raviart-Thomas element Raviart and Thomas [1977] for building an approximation of the $H(\mathrm{div},\Omega)$ space.

There is a subtle issue. The **Rheolef** implementation choice for this element bases on internal interpolation nodes instead of moments. This choice leads to more efficient computation of degrees of freedom, but the standard Lagrange interpolation π_h no more satisfies the comutation diagram and optimal error in the $H(\mathrm{div},\Omega)$ norm. Instead of the Lagrange interpolation π_h, we propose a projection operator, that satisfies these desired properties. We start building this projection operator for a piecewise discontinuous version of this element: it allows one to build a projection that requires only local operations and converges optimaly. Moreover, the piecewise discontinuous Raviart-Thomas approximation is used during the post-processing stage of the hybrid discontinuous Galekin method, that will be developped in a forthcoming chapter.

Let

$$\begin{aligned} V_h &= \left\{ \boldsymbol{v}_h \in \left(L^2(\Omega)\right)^d ; \;\; \boldsymbol{v}_{h|K} \in RT_k(K), \; \forall K \in \mathcal{T}_h \right\} \\ Q_h &= \left\{ q_h \in L^2(\Omega) \; ; \;\; q_{h|K} \in P_k, \; \forall K \in \mathcal{T}_h \right\} \end{aligned}$$

Here, V_h represents the space of discontinuous and piecewise k-th order Raviart-Thomas RT_k vector-valued functions while Q_h is the space of piecewise discontinuous polynomials.

Let π_{Q_h} denote the L_2 projection from $L^2(\Omega)$ into Q_h. For all $p \in L^2(\Omega)$, it is defined as $\pi_{Q_h}(p) = p_h \in Q_h$, where q_h is the solution of the following quadratic minimization problem:

$$p_h = \inf_{q_h \in Q_h} \int_\Omega (p - q_h)^2 \mathrm{d}x$$

Its solution is characterized as the unique solution of the following linear system, expressed in variational form:

(P_1): *find* $p_h \in Q_h$ *such that*

$$\int_\Omega p_h \, q_h \, \mathrm{d}x = \int_\Omega p \, q_h \, \mathrm{d}x, \quad \forall q_h \in Q_h$$

Following Roberts and Thomas [Roberts and Thomas, 1991, p. 551-552], our aim is to define π_{V_h} as the L_2 projection from $H(\mathrm{div},\Omega)$ into V_h and satisfying the following *commuting* property:

$$\begin{array}{ccc} H(\mathrm{div},\Omega) & \xrightarrow{\mathrm{div}} & L^2(\Omega) \\ \downarrow{\scriptstyle \pi_{V_h}} & & \downarrow{\scriptstyle \pi_{Q_h}} \\ V_h & \xrightarrow{\mathrm{div}} & Q_h \end{array}$$

i.e.

$$\mathrm{div}(\pi_{V_h}(\boldsymbol{u})) \quad = \quad \pi_{Q_h}(\mathrm{div}\,\boldsymbol{u}), \quad \forall \boldsymbol{u} \in H(\mathrm{div},\Omega) \tag{6.1a}$$

In [Roberts and Thomas, 1991, p. 553], theorem 6.3, this projection operator then satisfies an optimal error bound in the $H(\mathrm{div},\Omega)$ norm, i.e.:

$$\|u - \pi_{V_h}(\boldsymbol{u})\|_{0,2,\Omega} + \|\mathrm{div}(u - \pi_{V_h}(\boldsymbol{u}))\|_{0,2,\Omega} \quad = \quad \mathcal{O}(h^{k+1}) \tag{6.1b}$$

Remark that the Lagrange interpolation operator π_h from $H(\mathrm{div},\Omega)$ to V_h do not necessarily satisfy the commuting property (6.1a). Indeed, this depends upon the way the Raviart-Thomas internal degrees of freedom are chosen and implemented. When *internal degrees of freedom* are chosen as integrals over polynomials of degree $\ell \leqslant k-1$, e.g. as in [Roberts and Thomas, 1991, p. 551], eqn (6.8), then the Lagrange interpolation π_h satisfies both (6.1a) and (6.1b), as shown Roberts and Thomas [1991], theorem 6.1 and 6.3.

In practice, it is more efficient to choose for all the internal degrees of freedom of the Raviart-Thomas some values of the function on a set of internal points: this implementation choice has been chosen in **Rheolef**. In that case, the Lagrange interpolation π_h neither satisfies the commutation (6.1a) nor the bound (6.1b). More precisely, the Lagrange interpolation error is optimal in L^2 norm only while its divergence converges sub-optimally. Thus, with the present choice of the internal degrees of freedom, there is a need to explicitly compute the projection π_{V_h} that satisfies both (6.1a) and (6.1b).

For all $\boldsymbol{u} \in H(\mathrm{div},\Omega)$, this projection is defined as the L^2 projection of $\boldsymbol{u}$ under the constraint (6.1a):

$$\begin{array}{rcl} \pi_{V_h}(\boldsymbol{u}) & = & \displaystyle\inf_{\boldsymbol{v}_h \in V_h} \|\boldsymbol{u} - \boldsymbol{v}_h\|_{0,2,\Omega}^2 \\ & & \text{subject to} \quad \mathrm{div}(\boldsymbol{v}_v) \;=\; \pi_{Q_h}(\mathrm{div}\,\boldsymbol{u}) \end{array}$$

Then, $\boldsymbol{u}_h = \pi_{V_h}(\boldsymbol{u}) \in V_h$ is equivalently characterized as the solution of the following saddle-point problem:

$$(\boldsymbol{u}_h, p_h) \quad = \quad \inf_{\boldsymbol{v}_h \in V_h} \sup_{q_h \in Q_h} L(\boldsymbol{v}_h, q_h)$$

where the Lagrangian L is defined, for all $(\boldsymbol{v}_h, q_h) \in V_h \times Q_h$, by

$$L(\boldsymbol{v}_h, q_h) \quad = \quad \int_\Omega \left(|\boldsymbol{u} - \boldsymbol{v}_h|^2 + q_h \mathrm{div}(\boldsymbol{u} - \boldsymbol{v}_h) \right) \mathrm{d}x$$

The saddle-point of L is characterized as the unique solution of a linear system, expressed in variational form. Moreover, since both V_h and Q_h are spaces of piecewise discontinuous functions, the linear system writes as a collection of local linear systems on each element.

(P): *find* $(\boldsymbol{u}_h, p_h) \in V_h \times Q_h$ *such that, on each element* $K \in \mathcal{T}_h$, *we have*

$$\int_K (\boldsymbol{u}_h.\boldsymbol{v}_h + p_h \mathrm{div}\,\boldsymbol{v}_h)\mathrm{d}x \quad = \quad \int_K \boldsymbol{u}.\boldsymbol{v}_h \,\mathrm{d}x \tag{6.2a}$$

$$\int_K q_h \mathrm{div}\,\boldsymbol{u}_h \,\mathrm{d}x \quad = \quad \int_K q_h \mathrm{div}\,\boldsymbol{u}\,\mathrm{d}x \tag{6.2b}$$

for all $(\boldsymbol{v}_h, q_h) \in V_h \times Q_h$. Observe that q_h represents the Lagrange multiplier associated to the commutation constraint (6.2b), which is equivalent to (6.1a).

Let us introduce the following forms:

$$\begin{aligned} a(\boldsymbol{u}_h, p_h;\ \boldsymbol{v}_h, q_h) &= \int_\Omega (\boldsymbol{u}_h.\boldsymbol{v}_h + p_h \mathrm{div}\, \boldsymbol{v}_h + q_h \mathrm{div}\, \boldsymbol{u}_h)\, \mathrm{d}x \\ \ell(\boldsymbol{v}_h, q_h) &= \int_\Omega (\boldsymbol{u}.\boldsymbol{v}_h + q_h \mathrm{div}\, \boldsymbol{u})\, \mathrm{d}x \end{aligned}$$

The previous problem writes equivalently in abstract form:

(P): *find* $(\boldsymbol{u}_h, p_h) \in V_h \times Q_h$ *such that*

$$a(\boldsymbol{u}_h, p_h;\ \boldsymbol{v}_h, q_h) = \ell(\boldsymbol{v}_h, q_h), \quad \forall (\boldsymbol{v}_h, q_h) \in V_h \times Q_h$$

Note that the matrix associated to the bilinear form a is symmetric and block-diagonal: it can thus be efficiently inverted on the fly at the element level during the assembly process. The following code implement this efficient approach.

File 6.1: `commute_rtd.cc`

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
#include "cosinus_vector.h"
int main(int argc, char**argv) {
  environment rheolef (argc, argv);
  geo omega (argv[1]);
  size_t d = omega.dimension(),
         k = (argc > 2) ? atoi(argv[2]) : 0;
  space Vh (omega, "RT"+to_string(k)+"d"),
        Lh (omega, "P" +to_string(k)+"d"),
        Xh = Vh*Lh;
  trial x (Xh); test y (Xh);
  auto u = x[0], lambda = x[1];
  auto v = y[0], mu     = y[1];
  integrate_option iopt;
  iopt.invert = true;
  form  inv_a = integrate (dot(u,v) + div_h(v)*lambda + div_h(u)*mu, iopt);
  field lh = integrate (dot(u_exact(d),v) + div_u_exact(d)*mu),
        xh = inv_a*lh,
        p_Vh_u = xh[0],
        pi_h_u = lazy_interpolate(Vh,u_exact(d));
  dout << catchmark("p_Vh_u") << p_Vh_u
       << catchmark("pi_h_u") << pi_h_u;
}
```

How to run the program ?

```
make commute_rtd commute_rtd_error
mkgeo_grid -t 10 > square.geo
./commute_rtd square.geo | ./commute_rtd_error
```

The program '`commute_rt.cc`' compute both the projection $\pi_{V_h}(u)$ and the standard Lagrange interpolation $\pi_h(u)$, while '`commute_rtd_error.cc`' performs the computation of the corresponding errors. The file '`cosinus_vector.h`' furnishes the function used for the present test:

$$\boldsymbol{u}(x) = \begin{pmatrix} \cos(x_0 + 2x_1) \\ \sin(x_0 - 2x_1) \end{pmatrix}$$

These two last files are not listed here but are available in the **Rheolef** example directory. Observe on Fig. 6.1 that the error for the projection $\pi_{V_h}(u)$ and its divergence behave as $\mathcal{O}(h^{k+1})$, which is optimal. Conversely, the error for the Lagrange interpolation $\pi_h(u)$ is sub-optimal for the divergence.

In conclusion, the projection π_{V_h} should be used instead of the interpolation π_h when we want to build an optimal Raviart-Thomas approximation.

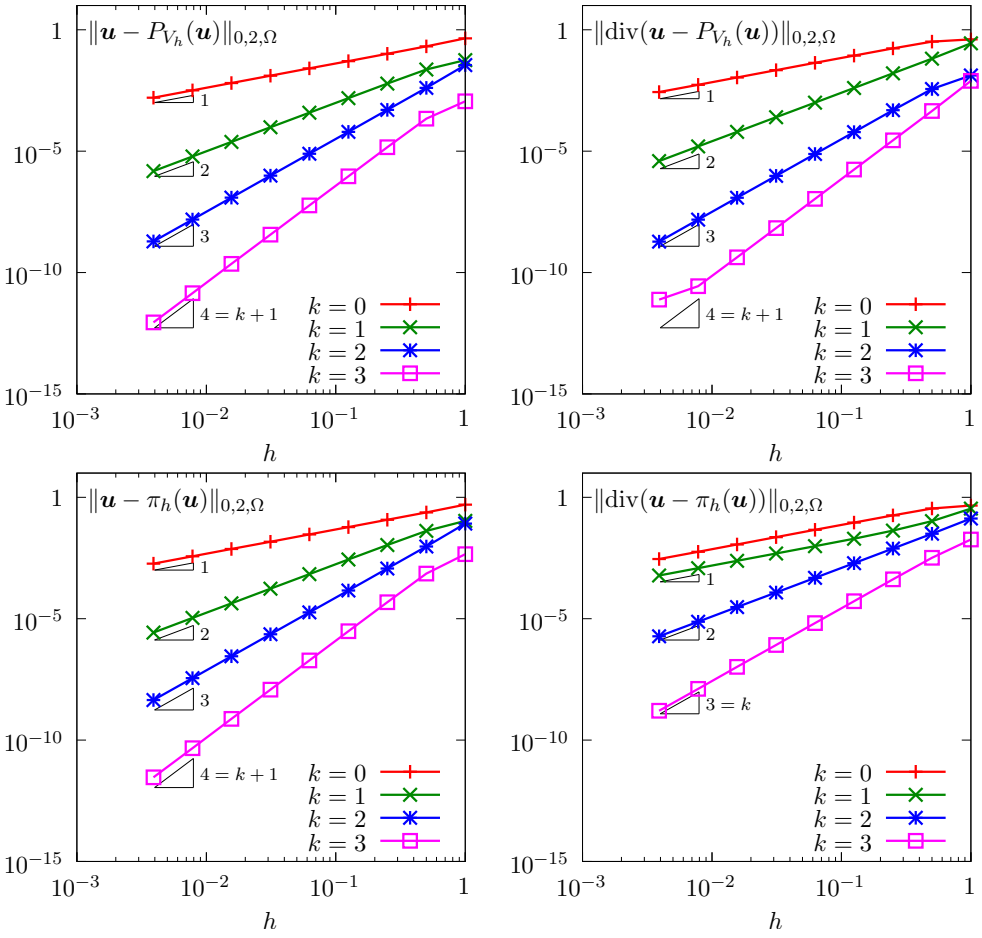


Figure 6.1: Raviart-Thomas approximation: π_{V_h} projection (top) and π_h interpolation (bottom) errors in L^2 norm for the approximation and its divergence.

6.2 Hybrid discontinuous Galerkin (HDG) methods

The aim of this chapter is to introduce to hybridization of discontinuous Galerkin methods within the **Rheolef** environment. For a review of hybridizable discontinuous Galerkin methods, see Nguyen et al. [2011]. The hybridization technique allows an efficient finite element implementation of many problems of importance, such as the Navier-Stokes one. Let us start by some model problems.

6.2.1 The Poisson problem

Let us consider the Poisson problem with mixed Dirichlet and Neumann boundary conditions:

(P): *find* u, *defined in* Ω, *such that*

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

where $\partial\Omega = \Gamma_d \cup \Gamma_n$, and the interior of the two boundary domains Γ_d and Γ_n are disjoints. The

data f, g_d and g_n are given. Let us introduce the gradient $\boldsymbol{\sigma} = \nabla u$ as an independent variable. The problem writes equivalently (see e.g. Nguyen et al. [2011]):

(P): *find* $\boldsymbol{\sigma}$ *and* u, *defined in* Ω, *such that*

$$\begin{cases} \boldsymbol{\sigma} - \nabla u = 0 \ \text{ in } \ \Omega & (6.3a) \\ \operatorname{div}(\boldsymbol{\sigma}) = -f \ \text{ in } \ \Omega & (6.3b) \\ u = g_d \ \text{ on } \ \Gamma_d & (6.3c) \\ \boldsymbol{\sigma}.\boldsymbol{n} = g_n \ \text{ on } \ \Gamma_n & (6.3d) \end{cases}$$

Let us multiply (6.3a) by a test function $\boldsymbol{\tau}$ and integrate by parts on any element K:

$$\int_K (\boldsymbol{\sigma}.\boldsymbol{\tau} + \operatorname{div}(\boldsymbol{\tau})\, u)\ \mathrm{d}x - \int_{\partial K} (\boldsymbol{\tau}.\boldsymbol{n})\, u\, \mathrm{d}s \quad = \quad 0$$

In the discontinuous Galerkin method, the trace of u on ∂K will be discontinuous across the inter-element boundaries. The hybridization replaces this trace by a new independent variable, denoted as λ, that is only definied on sides of the mesh:

$$\int_K (\boldsymbol{\sigma}.\boldsymbol{\tau} + \operatorname{div}(\boldsymbol{\tau})\, u)\ \mathrm{d}x - \int_{\partial K} (\boldsymbol{\tau}.\boldsymbol{n})\, \lambda\, \mathrm{d}s \quad = \quad 0 \tag{6.4a}$$

The λ variable will be uni-valued on boundary inter-elements and accounts strongly for the boundary condition: $\lambda = g_d$ on Γ_d. Then, let us multiply (6.3b) by a test function v with a zero average value on K:

$$\int_K \operatorname{div}(\boldsymbol{\sigma})\, v\, \mathrm{d}x \quad = \quad -\int_K f\, v\, \mathrm{d}x$$

In order to weakly impose the condition $u = \lambda$ on ∂K, we add a penalization term:

$$\int_K \operatorname{div}(\boldsymbol{\sigma})\, v\, \mathrm{d}x - \beta h^n \int_{\partial K} h^n (u - \lambda)\, v\, \mathrm{d}s \quad = \quad -\int_K f\, v\, \mathrm{d}x \tag{6.4b}$$

Here, h denotes the local mesh size in the element K. The two constants $\beta > 0$ and $n \in \mathbb{R}$ are respectively a penalization coefficient and power index. Following Cockburn et al. [2009], we consider the three cases $n = 0$, 1 and -1. We have three unknowns $\boldsymbol{\sigma}$, u and λ and only the two equations (6.4a) and (6.4b). For the problem to be complete, we add an equation for λ. Observe that (6.4b) writes equivalently, after a second integration by part on K:

$$-\int_K \boldsymbol{\sigma}.\nabla v\, \mathrm{d}x + \int_{\partial K} (\boldsymbol{\sigma}.\boldsymbol{n} - \beta h^n (u - \lambda))\, v\, \mathrm{d}s \quad = \quad -\int_K f\, v\, \mathrm{d}x$$

The quantity involved in the integral on ∂K is denoted as $\widehat{\boldsymbol{\sigma}} = \boldsymbol{\sigma} - \beta h^n (u - \lambda)\, \boldsymbol{n}$ and refered as the *numerical flux* across inter-element boundaries. As an additional equation, we impose the continuity of the normal component of this numerical flux. On the Γ_n boundary domain, the normal component of the numerical flux is imposed to be the prescribed flux g_n, while the Dirichlet condition $\lambda = g_d$ is precribed on the Γ_d boundary domain:

$$\begin{aligned} [\![\boldsymbol{\sigma} - \beta h^n (u - \lambda)\, \boldsymbol{n}]\!].\boldsymbol{n} &= 0 \ \text{ on } \ S,\ \forall S \in \mathscr{S}_h^{(i)} && (6.5a) \\ \boldsymbol{\sigma}.\boldsymbol{n} - \beta h^n (u - \lambda) &= g_n \ \text{ on } \ \Gamma_n && (6.5b) \\ \lambda &= g_d \ \text{ on } \ \Gamma_d && (6.5c) \end{aligned}$$

For an internal side $S = \partial K_+ \cap \partial K_- \in \mathscr{S}_h^{(i)}$ between two elements $K_1, K_2 \in \mathscr{T}_h$, relation (6.5a) writes:

$$\boldsymbol{\sigma}_+.\boldsymbol{n}_+ - \beta h^n (u_+ - \lambda) + \boldsymbol{\sigma}_-.\boldsymbol{n}_- - \beta h^n (u_- - \lambda) \quad = \quad 0 \ \text{ on } \ S$$

where $\boldsymbol{\sigma}_\pm$ and $u_\pm$ are the trace on S of $\boldsymbol{\sigma}$ and u in $K_\pm$ and $\boldsymbol{n}_\pm$ are the outer normal of $K_\pm$ on S. Since S is oriented, let us choose without loss of generality $\boldsymbol{n} = \boldsymbol{n}_- = -\boldsymbol{n}_+$. Then, the previous relation writes:

$$\begin{array}{rl} & (\boldsymbol{\sigma}_- - \boldsymbol{\sigma}_+).\boldsymbol{n} - \beta h^n (u_- + u_+) + 2\beta h^n \lambda \;=\; 0 \;\text{ on }\; S \\ \Longleftrightarrow & [\![\boldsymbol{\sigma}]\!].\boldsymbol{n} - 2\beta h^n (\{\!\{u\}\!\} - \lambda) \;=\; 0 \;\text{ on }\; S \end{array}$$

where we have used the jump and average across the internal side S. The previous relation, together with (6.5c) and (6.5c), writes in variational form:

$$\int_{\mathscr{S}_h^{(i)}} ([\![\boldsymbol{\sigma}]\!].\boldsymbol{n} - 2\beta h^n (\{\!\{u\}\!\} - \lambda))\, \mu \,\mathrm{d}s + \int_{\partial\Omega} (\boldsymbol{\sigma}.\boldsymbol{n} - \beta h^n (u - \lambda))\, \mu \,\mathrm{d}s \;=\; \int_{\Gamma_n} g_n \,\mu \,\mathrm{d}s \quad (6.6)$$

for all test function μ, defined on all internal sides of $\mathscr{S}_h^{(i)}$ and on all sides of the boundary domain Γ_n and that vanishes on all sides of the boundary domain Γ_d.

Grouping (6.4a), (6.4b) and (6.6), we obtain the discrete variational formulation:

$(FV)_h$: *find* $(\boldsymbol{\sigma}_h, u_h, \lambda_h) \in T_h \times X_h \times M_h(g_d)$ *such that*

$$\begin{array}{rcl} \displaystyle\int_\Omega \boldsymbol{\sigma}.\boldsymbol{\tau}\,\mathrm{d}x + \int_\Omega \mathrm{div}_h(\boldsymbol{\tau})\, u \,\mathrm{d}x - \int_{\mathscr{S}_h^{(i)}} ([\![\boldsymbol{\tau}]\!].\boldsymbol{n})\,\lambda \,\mathrm{d}s - \int_{\partial\Omega} (\boldsymbol{\tau}.\boldsymbol{n})\,\lambda \,\mathrm{d}s &=& 0 \\ \displaystyle\int_\Omega \mathrm{div}_h(\boldsymbol{\sigma})\, v \,\mathrm{d}x - \beta \sum_{K\in\mathscr{T}_h} \int_{\partial K} h^n u\, v \,\mathrm{d}s + \int_{\mathscr{S}_h^{(i)}} 2\beta h^n \{\!\{v\}\!\}\,\lambda \,\mathrm{d}s + \int_{\partial\Omega} \beta h^n v\,\lambda \,\mathrm{d}s &=& -\displaystyle\int_\Omega f\, v \,\mathrm{d}x \\ \displaystyle\int_{\mathscr{S}_h^{(i)}} (2\beta h^n \{\!\{u\}\!\} - [\![\boldsymbol{\sigma}]\!].\boldsymbol{n})\,\mu \,\mathrm{d}s + \int_{\partial\Omega} (\beta h^n u - \boldsymbol{\sigma}.\boldsymbol{n})\,\mu \,\mathrm{d}s - \int_{\mathscr{S}_h^{(i)}} 2\beta h^n \lambda\,\mu \,\mathrm{d}s - \int_{\partial\Omega} \beta h^n \lambda\,\mu \,\mathrm{d}s &=& -\displaystyle\int_{\Gamma_n} g_n\,\mu \,\mathrm{d}s \end{array}$$

for all $(\boldsymbol{\tau}_h, v_h, \tau_h) \in T_h \times X_h \times M_h(0)$, where

$$\begin{array}{rcl} T_h &=& \left\{ \boldsymbol{\tau} \in \left(L^2(\Omega)\right)^d \;;\; \boldsymbol{\tau}_{|K} \in (P_k)^d,\ \forall K \in \mathscr{T}_h \right\} \\ X_h &=& \left\{ v \in L^2(\Omega) \;;\; v_{|K} \in P_k,\ \forall K \in \mathscr{T}_h \right\} \\ M_h &=& \left\{ \mu \in L^2(\mathscr{S}_h) \;;\; \mu_{|S} \in P_k,\ \forall S \in \mathscr{S}_h \right\} \\ M_h(g_d) &=& \left\{ \mu \in M_h \;;\; \mu_{|S} = \pi_h(g_d),\ \forall S \subset \Gamma_d \right\} \end{array}$$

and $k \geqslant 0$ is the polynomial degree. This is a symmetric system with a mixed structure. Let:

$$\begin{array}{rcl} a_h((\boldsymbol{\sigma},u),(\boldsymbol{\tau},v)) &=& \displaystyle\int_\Omega (\boldsymbol{\sigma}.\boldsymbol{\tau} + \mathrm{div}_h(\boldsymbol{\tau})\, u + \mathrm{div}_h(\boldsymbol{\sigma})\, v)\ \mathrm{d}x - \beta \sum_{K\in\mathscr{T}_h} \int_{\partial K} h^n u\, v \,\mathrm{d}s \\ b_h((\boldsymbol{\tau},v),\mu) &=& \displaystyle\int_{\mathscr{S}_h^{(i)}} (-[\![\boldsymbol{\tau}]\!].\boldsymbol{n} + 2\beta h^n \{\!\{v\}\!\})\,\mu \,\mathrm{d}s + \int_{\partial\Omega} (-\boldsymbol{\tau}.\boldsymbol{n} + \beta h^n \{\!\{v\}\!\})\,\mu \,\mathrm{d}s \\ c_h(\lambda,\mu) &=& \displaystyle\int_{\mathscr{S}_h^{(i)}} 2\beta h^n \lambda\,\mu \,\mathrm{d}s + \int_{\partial\Omega} \beta h^n \lambda\,\mu \,\mathrm{d}s \\ \ell_h(\boldsymbol{\tau},v) &=& \displaystyle -\int_\Omega f\, v \,\mathrm{d}x \\ k_h(\mu) &=& \displaystyle -\int_{\Gamma_n} g_n\,\mu \,\mathrm{d}s \end{array}$$

Then, the variational formulation writes equivalently:

$(FV)_h$: *find* $(\boldsymbol{\sigma}_h, u_h, \lambda_h) \in T_h \times X_h \times M_h(g_d)$ *such that*

$$\begin{array}{rcl} a_h((\boldsymbol{\sigma},u),(\boldsymbol{\tau},v)) + b_h((\boldsymbol{\tau},v),\lambda) &=& \ell_h(\boldsymbol{\tau},v) \\ b_h((\boldsymbol{\sigma},u),\mu) - c_h(\lambda,\mu) &=& k_h(\mu) \end{array}$$

for all $(\boldsymbol{\tau}_h, v_h, \tau_h) \in T_h \times X_h \times M_h(0)$. Let $\chi_h = (\boldsymbol{\sigma}_h, u_h)$. This linear symmetric system admits the following matrix structure:

$$\begin{pmatrix} A & B^T \\ B & -C \end{pmatrix} \begin{pmatrix} \boldsymbol{\chi}_h \\ \lambda_h \end{pmatrix} = \begin{pmatrix} F \\ G \end{pmatrix}$$

A carreful study shows that the A matrix has a block-diagonal structure at the element level and can be easily inverted on the fly during the assembly process. The matrix structure writes equivalently:

$$\begin{aligned} &\iff \begin{cases} A\boldsymbol{\chi}_h + B^T\lambda_h &= F \\ B\boldsymbol{\chi}_h - C\lambda_h &= G \end{cases} \\ &\iff \begin{cases} \boldsymbol{\chi}_h = A^{-1}(F - B^T\lambda_h) \\ (C + BA^{-1}B^T)\lambda_h = BA^{-1}F - G \end{cases} \end{aligned}$$

The second equation is solved first: the only remaining unknown is the Lagrange multiplier λ_h, in a linear system involving the Schur complement matrix $S = C + BA^{-1}B^T$. Then, the two variables $\boldsymbol{\chi}_h = (\boldsymbol{\sigma}_h, u_h)$ are simply obtained by a direct computation.

File 6.2: `dirichlet_hdg.cc`

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
#include "sinusprod_dirichlet.h"
int main(int argc, char**argv) {
  environment rheolef (argc, argv);
  geo omega (argv[1]);
  string approx = (argc > 2) ?      argv[2]  : "P1d";
  Float  n      = (argc > 3) ? atof(argv[3]) : 1;
  Float  beta   = (argc > 4) ? atof(argv[4]) : 1;
  space Th (omega, approx, "vector"),
        Xh (omega, approx),
        Yh = Th*Xh,
        Mh (omega["sides"], approx);
  Mh.block("boundary");
  space Wh(Mh.get_geo()["boundary"],approx);
  size_t d = omega.dimension();
  size_t k = Xh.degree();
  trial x(Yh), lambda(Mh);
  test  y(Yh), mu(Mh);
  auto sigma = x[0], u = x[1];
  auto tau   = y[0], v = y[1];
  integrate_option iopt;
  iopt.invert = true;
  auto coef = beta*pow(h_local(),n);
  form inv_a = integrate(dot(sigma,tau) + u*div_h(tau) + v*div_h(sigma)
                         - on_local_sides(coef*u*v), iopt);
  form b = integrate("internal_sides",
             (-dot(jump(sigma),normal()) + 2*coef*average(u))*mu)
         + integrate("boundary", (-dot(sigma,normal()) + coef*u)*mu);
  form c = integrate("internal_sides", 2*coef*lambda*mu)
         + integrate("boundary",          coef*lambda*mu);
  field lh = integrate (-f(d)*v);
  field kh(Mh,0), lambda_h(Mh,0);
  lambda_h ["boundary"] = lazy_interpolate (Wh, g(d));
  form s = c + b*inv_a*trans(b);
  field rh = b*(inv_a*lh) - kh;
  problem p (s);
  p.solve (rh, lambda_h);
  field xh = inv_a*(lh - b.trans_mult(lambda_h));
  dout << catchmark("n")      << n << endl
       << catchmark("beta")   << beta << endl
       << catchmark("u")      << xh[1]
       << catchmark("lambda") << lambda_h
       << catchmark("sigma")  << xh[0];
}
```

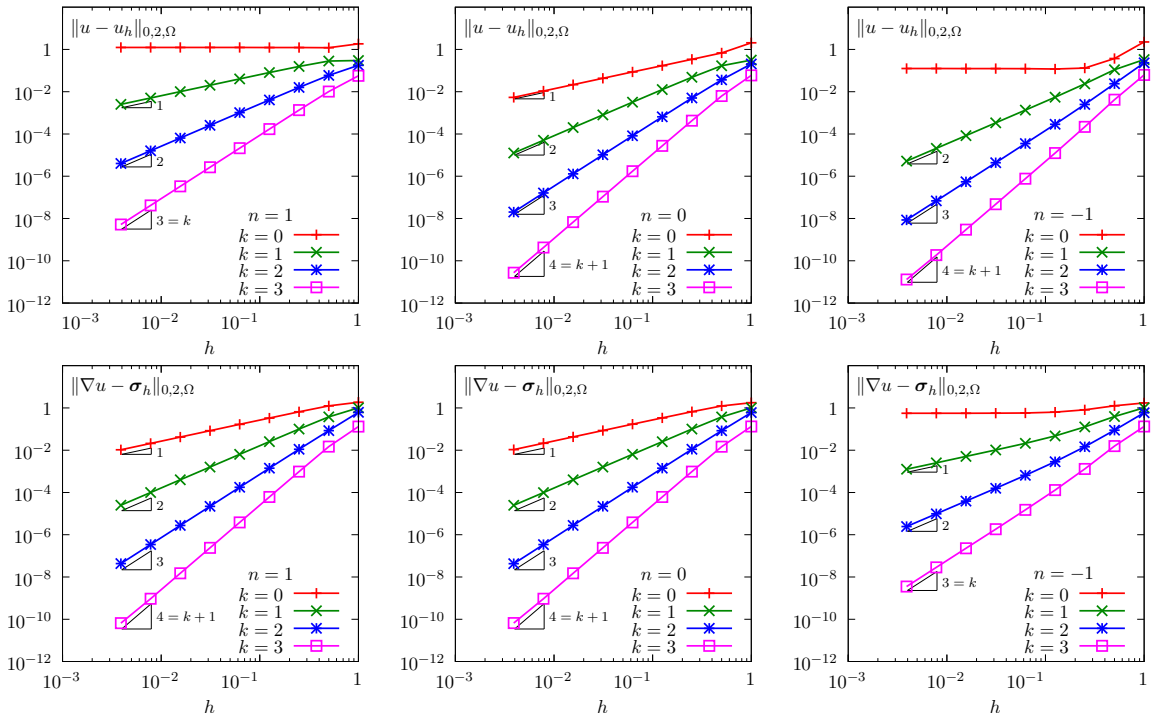


Figure 6.2: Hybrid discontinuous Galerkin method: Poisson problem with Dirichlet boundary conditions in 2D geometry. Convergence vs h and k for the approximation of the solution u_h and of its gradient $\boldsymbol{\sigma}_h$. (left) $n=1$; (center) $n=0$; (right) $n=-1$.

The right-hand-side f and the Dirichlet boundary condition g has been chosen such that the exact solution is given by:

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

The files '`sinusprod_dirichlet.h`', that defines the data f and g, and '`sinusprod_error_hdg.cc`', that compute the error in various norms, are not listed here but are available in the **Rheolef** example directory.

How to run the program

The compilation and run write:

```
make dirichlet_hdg
mkgeo_grid -t 10 > square.geo
./dirichlet_hdg square.geo P1d > square.field
field square.field -elevation
field square.field -elevation -mark lambda
field square.field -velocity  -mark sigma
make sinusprod_error_hdg
./sinusprod_error_hdg < square.field
```

Fig. 6.2 plots the errors vs h and k for a two dimensional geometry. Observe that the approximation u_h converges optimaly, as h^{k+1}, in the L^2 norm for any $k \geqslant 0$ when the power index $n=1$. When $n=1$, the convergence is suboptimal, as h^k only and note that the lowest order approximation $k=0$ is not convergent. When $n=-1$, the convergence is optimal, as h^{k+1}, only when $k \geqslant 1$,

while the lowest order approximation $k = 0$ is not convergent. The approximation $\boldsymbol{\sigma}_h$ of the gradient converges optimaly, as h^{k+1}, for both $n = 0$ and 1, while it is suboptimal, as h^k, when $n = -1$. All these results are consistent with the approxiamtion theory of the HDG method, see Cockburn et al. [2009] and tables 2, 3, 6.

6.2.2 Superconvergence of the Lagrange multiplier

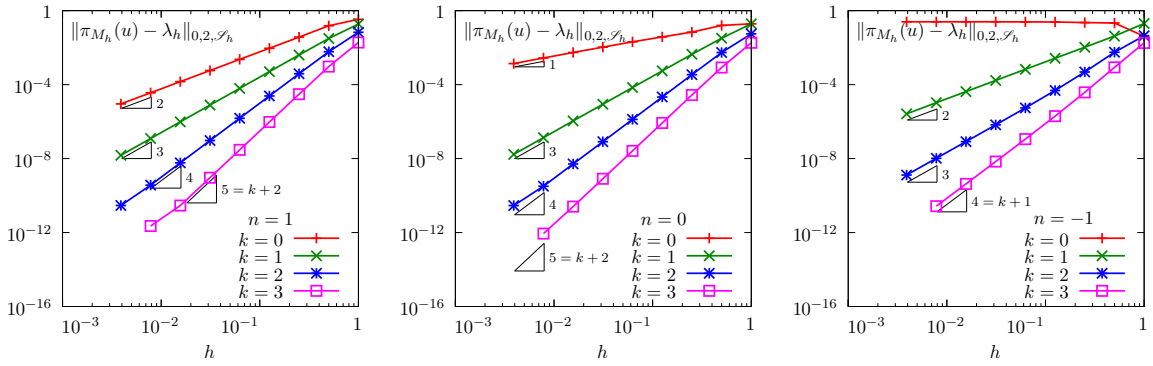


Figure 6.3: Hybrid discontinuous Galerkin method: Poisson problem with Dirichlet boundary conditions in 2D geometry. Super-convergence vs h and k for the Lagrange multiplier λ_h to the L^2 projection on M_h of the exact solution. (left) $n = 1$; (center) $n = 0$; (right) $n = -1$.

Let π_{M_h} denote the L_2 projection from $L^2(\mathscr{S}_h)$ into M_h. We consider the special case of computing the projection $\pi_{M_h}(u) \in M_h$ of the restriction to $\mathscr{S}_h$ of an element $u \in L^2(\mathscr{S}_h)$. It is defined as

$$\pi_{M_h}(u) \quad = \quad \inf_{\mu_h \in M_h} \frac{1}{2} \sum_{S \in \mathscr{S}_h} \int_S (u - \mu_h)^2 \mathrm{d}s$$

The following bilinear forms are introduced:

$$m_s(\lambda, \mu) \quad = \quad \sum_{S \in \mathscr{S}_h} \int_S \lambda \, \mu \, \mathrm{d}s$$

$$k_h(\mu) \quad = \quad \sum_{S \in \mathscr{S}_h} \int_S u \, \mu \, \mathrm{d}s$$

Then, the projection reduces to:

find $\bar{\lambda}_h = \pi_{M_h}(u) \in M_h$ *such that*

$$m_s(\bar{\lambda}_h, \mu_h) \quad = \quad k_h(\mu_h), \quad \forall \mu_h \in M_h$$

The result of projection operator can be obtained by a resolution of a linear system. Following [Cockburn et al., 2008, p. 1896], in order to measure the error on the set of sides $\mathscr{S}_h$ of the mesh, we introduce the mesh-dependent norm $\|.\|_{0,2,\mathscr{S}_h}$, defined for all $\mu \in L^2(\mathscr{S}_h)$ by

$$\|\mu\|^2_{0,2,\mathscr{S}_h} = \sum_{S \in \mathscr{S}_h} \int_S h_S \mu^2 \, \mathrm{d}s$$

where h_S is a characteristic length on the side $S \in \mathscr{S}_h$. Fig. 6.3 plots the difference $\pi_{M_h}(u) - \lambda_h$ in this mesh-dependent norm. In agrement with the theoretical results [Cockburn et al., 2008, p. 1896], we observe the superconvergence of the multiplier λ_h to the L^2 projection $\pi_{M_h}(u)$. More precisely, when $n = 1$ the order of convergence is $k + 2$ for any $k \geqslant 0$. When $n = 0$, the superconvergence occurs only when $k \geqslant 1$ and when $n = -1$ there is no superconvergence. This error is also computed by the code '`sinusprod_error_hdg.cc`'.

6.2.3 Superconvergence of the piecewise averaged solution

File 6.3: `dirichlet_hdg_average.icc`

```
field dirichlet_hdg_average (field uh, field lambda_h) {
  size_t k = uh.get_space().degree();
  size_t d = uh.get_geo().dimension();
  space Zh (uh.get_geo(), "P0");
  trial zeta(Zh); test xi(Zh);
  integrate_option iopt;
  iopt.invert = true;
  if (k >= 1) {
    form inv_mz = integrate (zeta*xi, iopt);
    iopt.set_order(2*k+2);
    field lh = integrate (uh*xi, iopt);
    return inv_mz*lh;
  }
  const space& Mh = lambda_h.get_space();
  trial lambda (Mh); test mu (Mh);
  form inv_ms = integrate("sides", lambda*mu, iopt);
  field inv_sh = inv_ms*field(Mh,1);
  field lh = integrate (on_local_sides(inv_sh*lambda_h*xi));
  return (1./(d+1))*lh;
}
```

File 6.4: `dirichlet_hdg_average.cc`

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
#include "dirichlet_hdg_average.icc"
int main(int argc, char**argv) {
  environment rheolef (argc, argv);
  field uh, lambda_h;
  din >> catchmark("u")      >> uh
      >> catchmark("lambda") >> lambda_h;
  field bar_uh = dirichlet_hdg_average (uh, lambda_h);
  dout << catchmark("u")      << uh
       << catchmark("bar_u") << bar_uh;
}
```

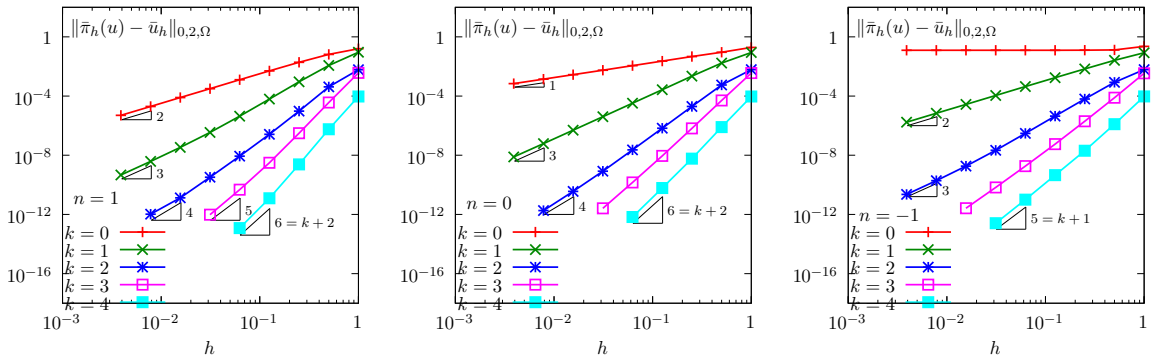


Figure 6.4: Hybrid discontinuous Galerkin method: Poisson problem with Dirichlet boundary conditions in 2D geometry. Super-convergence vs h and k for the piecewise average $\bar{u}_h$ of the approximate solution. (left) $n = 1$; (center) $n = 0$; (right) $n = -1$.

The superconvergence of the piecewise average values of the solution obtained by the hybrid discontinuous Galerkin method was first observed in Cockburn et al. [2008, 2009] and then exploited for many postprocessing application (see e.g. Nguyen et al. [2011]). The file '`dirichlet_hdg_average.icc`' compute $\bar{u}_h = \bar{\pi}_h(u_h, \lambda_h)$, the averaged solution, defined by

(see Cockburn et al. [2008], eqn (2.9b)):

$$\bar{\pi}_h(u_h,\lambda_h) = \begin{cases} \dfrac{1}{d}\displaystyle\sum_{S\subset\partial K}\lambda_h & \text{when } k=0 \\ \dfrac{1}{\mathrm{meas}(K)}\displaystyle\int_K u_h\,\mathrm{d}x & \text{when } k\geqslant 1 \end{cases}$$

The file '`sinusprod_error_hdg_average.cc`' that compute the error $\bar{u}_h - \bar{\pi}_h(u)$ is not listed here but is available in the **Rheolef** example directory. The computation of the error is obtained by:

```
make dirichlet_hdg dirichlet_hdg_average sinusprod_error_hdg_average
mkgeo_grid -t 10 > square.geo
./dirichlet_hdg square.geo P1d | dirichlet_hdg_average | \
    ./sinusprod_error_hdg_average
```

Observe on Fig. 6.4 that, when $n=1$ and for any $k\geqslant 0$, we obtain $\bar{u}_h - \bar{\pi}_h(u)$ of order $k+2$. This is a remarkable result since u_h is piecewise polynomial of order k and is expected to converge at order $k+1$. When $n=0$, for any $k\geqslant 1$ we also observe this superconvergence while, when $k=0$, the average value converges only at first order. Finaly, when $n=-1$, for any $k\geqslant 1$ we do no more observe These observations are consistent with those of [Cockburn et al., 2009, p. 16] (see also [Cockburn et al., 2008, p. 1613]).

6.2.4 Improving the solution by local postprocessing

By combining the approximation $\boldsymbol{\sigma}_h$ of the gradient ∇u, that converges at rate $k+1$ (see Fig. 6.2), with the average $\bar{u}_h$ that super-converges at rate $k+2$ (see Fig. 6.4), it is then possible, by a local integration inside each element, to obtain a new approximation u_h^*, that is piecewise discontinuous polynomial of order $k+1$, and that converges at rate $k+2$.

The postprocess step is nothing than the resolution of the following local Neumann problem in any element $K\in\mathcal{T}_h$ (see [Cockburn et al., 2010b, p. 1360], eqn (5.1)):

(P^*): *find* u^*, *defined in* K, *such that*

$$\begin{cases} -\Delta u^* = f \ \text{ in } K \\ \dfrac{\partial u^*}{\partial n} = \boldsymbol{\sigma}_h.\boldsymbol{n} \ \text{ on } \partial K \\ \displaystyle\int_K u^*\,\mathrm{d}x = \int_K \bar{u}_h\,\mathrm{d}x \end{cases}$$

where $\boldsymbol{\sigma}_h$ and $\bar{u}_h$ are given by the previous resolution.

For any $\beta\in\mathbb{R}$, let us introduce the following functional space:

$$X_K^*(\beta) \quad = \quad \left\{ v\in H^1(K)\ ;\ \int_K v\,\mathrm{d}x = \beta \right\}$$

Then, the variational formulation of the problem writes:

(FV^*): *find* $u^*\in X_K\left(\int_K \bar{u}_h\,\mathrm{d}x\right)$ *such that*

$$\int_K \nabla u^*.\nabla v^*\,\mathrm{d}x \quad = \quad \int_K f\,v^*\,\mathrm{d}x + \int_{\partial K}\boldsymbol{\sigma}_h.\boldsymbol{n}\,v^*\,\mathrm{d}s, \quad \forall v^*\in X_K(0)$$

The linear constraint for the imposition of the average value is not easy to impose in $X_K(\beta)$. Indeed, $X_K(\beta)$ is not a vectorial space when $\beta\neq 0$. Following the methodology previously introduced for the Neumann boundary conditions for the Laplace operator, we introduce a Lagrange multiplier denoted here as ζ, which is constant inside each element K.

Finally, the postprocessing step compute the approximation u_h^*, when $\boldsymbol{\sigma}_h$ and u_h are known,as:
$(FV^*)_h$: *find* $(u_h^*, \zeta_h) \in X_h^* \times Z_h$ *such that, on each element* $K \in \mathscr{T}_h$, *we have*

$$\begin{cases} \displaystyle\int_K \nabla u_h^* . \nabla v_h^* \,\mathrm{d}x + \int_K v_h^* \,\zeta_h \,\mathrm{d}x = \int_K f\, v_h^* \,\mathrm{d}x + \int_{\partial K} \boldsymbol{\sigma}_h . \boldsymbol{n}\, v_h^* \,\mathrm{d}s, \quad \forall v_h^* \in X_h^* \\ \displaystyle\int_K u_h^* \,\xi_h \,\mathrm{d}x \qquad\qquad\qquad\qquad = \int_K \bar{u}_h \,\xi_h \,\mathrm{d}x, \quad \forall \xi_h \in X_h^* \end{cases}$$

where

$$\begin{aligned} X_h^* &= \left\{ v \in L^2(\Omega) \,;\, v_{|K} \in P_{k+1},\ \forall K \in \mathscr{T}_h \right\} \\ Z_h &= \left\{ \xi \in L^2(\Omega) \,;\, \xi_{|K} \in P_0,\ \forall K \in \mathscr{T}_h \right\} \end{aligned}$$

Let

$$\begin{aligned} a_h^*(u, \zeta;\, v, \xi) &= \int_\Omega (\nabla_h u . \nabla_h v + v\,\zeta + u\,\xi) \,\mathrm{d}x \\ \ell_h^*(v, \xi) &= \int_\Omega (f\, v + \bar{u}_h \,\xi) \,\mathrm{d}x + \sum_{K \in \mathscr{T}_h} \int_{\partial K} (\boldsymbol{\sigma}_h . \boldsymbol{n})\, v \,\mathrm{d}s \end{aligned}$$

The second step writes in this abstract setting:
$(FV^*)_h$: *find* $(u_h^*, \zeta_h^*) \in X_h^* \times Z_h$ *such that*

$$a_h^*(u_h^*, \zeta_h;\, v_h^*, \xi_h) = \ell_h^*(v_h^*, \xi_h), \quad \forall (v_h^*, \xi_h) \in X_h^* \times Z_h$$

Note that the matrix associated to the bilinear form a_h^* is symmetric and block-diagonal: it can thus be easily be inverted on the fly at the element level. The present postprocessing stage was first introduced in [Cockburn et al., 2010b, p. 1360], eqn (5.1) as a replacement and a simplification of those previously introduced in Cockburn et al. [2008, 2009]. The following code implement this postprocessing.

File 6.5: dirichlet_hdg_post.cc

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
#include "sinusprod_dirichlet.h"
#include "dirichlet_hdg_average.icc"
int main(int argc, char**argv) {
  environment rheolef (argc, argv);
  Float n, beta;
  field uh, lambda_h, sigma_h;
  din >> catchmark("n")      >> n
      >> catchmark("beta")   >> beta
      >> catchmark("u")      >> uh
      >> catchmark("lambda") >> lambda_h
      >> catchmark("sigma")  >> sigma_h;
  field bar_uh = dirichlet_hdg_average (uh, lambda_h);
  const geo& omega = uh.get_geo();
  size_t d = omega.dimension();
  size_t k = uh.get_space().degree();
  space Xhs (omega, "P"+to_string(k+1)+"d"),
        Zhs (omega, "P0"),
        Yhs = Xhs*Zhs;
  trial x(Yhs); test y(Yhs);
  auto us = x[0], zeta = x[1];
  auto vs = y[0], xi   = y[1];
  integrate_option iopt;
  iopt.invert = true;
  form inv_ahs = integrate(dot(grad_h(us),grad_h(vs)) + zeta*vs + xi*us, iopt);
  field lhs = integrate (f(d)*vs + xi*bar_uh
                         + on_local_sides(dot(sigma_h,normal())*vs));
  field xhs = inv_ahs*lhs;
  dout << catchmark("n")      << n << endl
       << catchmark("beta")   << beta << endl
       << catchmark("u")      << xhs[0]
       << catchmark("lambda") << lambda_h
       << catchmark("sigma")  << sigma_h
       << catchmark("zeta")   << xhs[1];
}
```

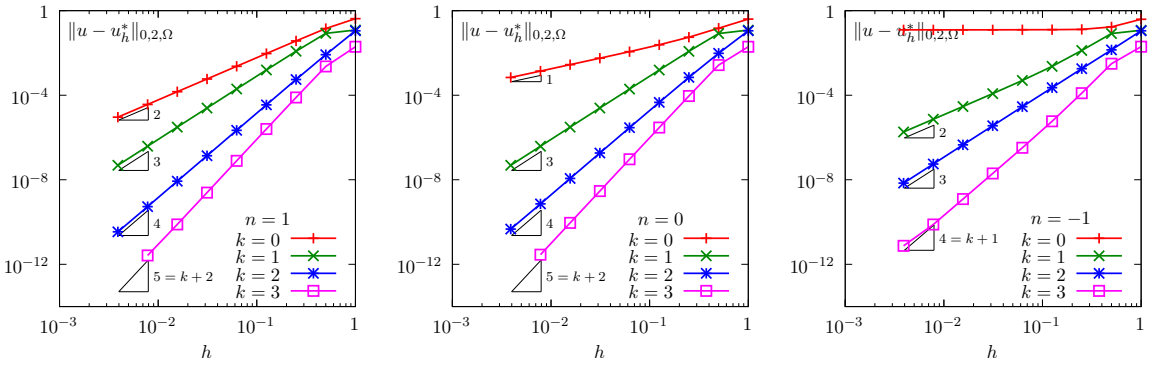


Figure 6.5: Solution post-processing of the hybrid discontinuous Galerkin method: Poisson problem with Dirichlet boundary conditions in 2D geometry. Convergence vs h and k for the post-treated approximate solution. (left) $n = 1$; (center) $n = 0$; (right) $n = -1$.

How to run the program

The compilation and run write:

```
make dirichlet_hdg dirichlet_hdg_post ./sinusprod_error_hdg
mkgeo_grid -t 10 > square.geo
```

```
./dirichlet_hdg square.geo P1d > square.field
./dirichlet_hdg_post < square.field > square-post.field
field square-post.field -elevation
```

Note the dramatic improvement of the post-treated solution, when compared with the original one, that could be observed with:

```
field square.field -elevation
```

This improvement could be accurately quantified by the error analysis:

```
./sinusprod_error_hdg < square.field
./sinusprod_error_hdg < square-post.field
```

The results are shown on Fig. 6.5. When $n = 1$, observe that, for any $k \geqslant 0$, the error for the post-treated solution u_h^* converges to zero at rate $k+2$ in L^2 norm, which is optimal, since u_h^* is a piecewise $k+1$ degree polynomial. When $n = 0$, this result is obtained only for $k \geqslant 1$ while, when $n = -1$, the convergence is suboptimal.

6.2.5 Improving the gradient with the Raviart-Thomas element

Let $(\boldsymbol{\sigma}_h, u_h, \lambda_h)$ be the solution of problem $(FV)_h$. This solution is considered here as known. Observe that the normal component of the numerical flux $\widehat{\boldsymbol{\sigma}}_h.\boldsymbol{n} = \boldsymbol{\sigma}_h.\boldsymbol{n} - \beta h^n(u_h - \lambda_h)$ is continuous across any inter-element boundaries. Indeed, relation (6.5a) is imposed weakely in the variational formulation $(FV)_h$, with a Lagrange multiplier $\mu_h \in M_h$ which is piecewise polynomial of degree k. Since $\widehat{\boldsymbol{\sigma}}_h$ is also piecewise polynomial of degree k, the discrete version of (6.5a) is also satisfied strongly:

$$\widehat{\boldsymbol{\sigma}}_h.\boldsymbol{n} \;=\; [\![\boldsymbol{\sigma}_h - \beta h^n(u_h - \lambda_h)\,\boldsymbol{n}]\!].\boldsymbol{n} \;=\; 0 \;\text{ on }\; S,\; \forall S \in \mathscr{S}_h^{(i)}$$

Since any element of $H(\mathrm{div},\Omega)$ presents the continuity of its normal component across any inter-element boundaries, is possible to define a new $H(\mathrm{div},\Omega)$ conform approximation of $\boldsymbol{\sigma}$, denoted as $\tilde{\boldsymbol{\sigma}}_h$, which satisfies $\tilde{\boldsymbol{\sigma}}_h.\boldsymbol{n} = \widehat{\boldsymbol{\sigma}}_h.\boldsymbol{n}$ on any internal sides. Similarly to (6.5a), $\tilde{\boldsymbol{\sigma}}_h$ also should satisfy a discrete version of (6.5c)-(6.5c) i.e. $\widehat{\boldsymbol{\sigma}}_h.\boldsymbol{n}$ is equals to g_n on Γ_n and to $\boldsymbol{\sigma}_h.\boldsymbol{n} - \beta h^n(u_h - g_d)$ on Γ_d. It is characterized as the unique element $\tilde{\boldsymbol{\sigma}}_h \in \tilde{T}_h$ satisfying in any element $K \in \mathscr{T}_h$ the following local variational formulation [Cockburn et al., 2010b, p. 1360]:

$$\begin{aligned}
\int_K \tilde{\boldsymbol{\sigma}}_h.\tilde{\boldsymbol{\gamma}}_h \,\mathrm{d}x &= \int_K \boldsymbol{\sigma}_h.\tilde{\boldsymbol{\gamma}}_h \,\mathrm{d}x, \quad \forall \tilde{\boldsymbol{\gamma}}_h \in \tilde{G}_h \\
\int_{\partial K} (\tilde{\boldsymbol{\sigma}}_h.\boldsymbol{n})\,\tilde{\mu}_h \,\mathrm{d}s &= \int_{\partial K} (\boldsymbol{\sigma}_h.\boldsymbol{n} - \beta h^n(u_h - \lambda_h))\,\tilde{\mu}_h \,\mathrm{d}s, \quad \forall \tilde{\mu}_h \in \tilde{M}_h
\end{aligned}$$

where we have introduced the finite dimensional spaces

$$\begin{aligned}
\tilde{T}_h &= \left\{ \tilde{\boldsymbol{\tau}}_h \in \left(L^2(\Omega)\right)^d ;\; \tilde{\boldsymbol{\tau}}_{h|K} \in RT_k(K),\; \forall K \in \mathscr{T}_h \right\} \\
\tilde{W}_h &= \left\{ \tilde{\gamma}_h \in \left(L^2(\Omega)\right)^d ;\; \tilde{\gamma}_{h|K} \in P_{k-1}(K),\; \forall K \in \mathscr{T}_h \right\} \\
\tilde{M}_h &= \prod_{K\in\mathscr{T}_h} \left\{ \tilde{\boldsymbol{\tau}}_{|\partial K}.\boldsymbol{n} \;;\; \tilde{\boldsymbol{\tau}} \in RT_k(K) \right\} \;=\; \prod_{K\in\mathscr{T}_h} \prod_{S\subset\partial K} P_k(S)
\end{aligned} \tag{6.7}$$

The space $\tilde{T}_h$ contains discontinuous and piecewise k-th order Raviart-Thomas RT_k polynomial vector-valued functions. The space $\tilde{M}_h$ contains, inside each element, the normal trace of functions of $\tilde{T}_h$. Note that elements of $\tilde{M}_h$ are piecewise P_k on all sides $S \subset \partial K$. Moreover, for all internal side $S \in \mathscr{S}_h^{(i)}$, the elements of $\tilde{M}_h$ are bi-valued, due to the discontinuity of elements of $\tilde{\boldsymbol{\tau}} \in \tilde{T}_h$.

Finally, $\tilde{W}_h$ contains piecewise discontinuous polynomial of degree k–1. By convention, when $k=0$, then the space $\tilde{W}_h$ is empty. Observe that $\dim(\tilde{T}_h) = \dim(\tilde{W}_h) + \dim(\tilde{M}_h)$ and that the previous local variational problem is well-posed. Let

$$\begin{aligned}
\tilde{a}_h(\tilde{\boldsymbol{\sigma}}_h;\ [\tilde{\boldsymbol{\gamma}}_h, \tilde{\mu}_h]) &= \int_\Omega \tilde{\boldsymbol{\sigma}}_h.\tilde{\boldsymbol{\gamma}}_h \,\mathrm{d}x + \sum_{K\in\mathscr{T}_h}\sum_{S\subset\partial K}\int_S (\tilde{\boldsymbol{\sigma}}_h.\boldsymbol{n})\,\tilde{\mu}_h \,\mathrm{d}s \\
\tilde{\ell}_h([\tilde{\boldsymbol{\gamma}}_h, \tilde{\mu}_h]) &= \int_\Omega \boldsymbol{\sigma}_h.\tilde{\boldsymbol{\gamma}}_h \,\mathrm{d}x + \sum_{K\in\mathscr{T}_h}\int_{\partial K} (\boldsymbol{\sigma}_h.\boldsymbol{n} - \beta h^n(u_h-\lambda_h))\,\tilde{\mu}_h \,\mathrm{d}s
\end{aligned}$$

For known $(\boldsymbol{\sigma}_h, u_h, \lambda_h)$, the postprocessing of the gradient writes:

$(\widetilde{FV})_h$: *find* $\tilde{\boldsymbol{\sigma}}_h \in \tilde{T}_h$ *such that* $\tilde{a}_h(\tilde{\boldsymbol{\sigma}}_h, [\tilde{\boldsymbol{\gamma}}_h, \tilde{\mu}_h]) = \tilde{\ell}_h([\tilde{\boldsymbol{\gamma}}_h, \tilde{\mu}_h])$ for all $(\tilde{\boldsymbol{\gamma}}_h, \tilde{\mu}_h) \in \tilde{W}_h \times \tilde{M}_h$.

A carreful study shows that the matrix associated with the $\tilde{a}_h$ bilinear form has a block-diagonal structure at the element level and can be efficiently inverted on the fly during the assembly process. This property is due to the usage of discontinuous Raviart-Thomas approximation space $\tilde{T}_h$. Moreover, since the normal component of $\tilde{\boldsymbol{\sigma}}_h$ is equal to the numerical flux $\hat{\boldsymbol{\sigma}}_h.\boldsymbol{n}$ which is continuous across inter-element boundaries this is also the case for $\tilde{\boldsymbol{\sigma}}_h.\boldsymbol{n}$ and finally $\tilde{\boldsymbol{\sigma}}_h \in H(\mathrm{div},\Omega)$.

File 6.6: `dirichlet_hdg_post_rt.cc`

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
int main(int argc, char**argv) {
  environment rheolef (argc, argv);
  Float n, beta;
  field sigma_h, uh, lambda_h;
  din >> catchmark("n")      >> n
      >> catchmark("beta")   >> beta
      >> catchmark("u")      >> uh
      >> catchmark("lambda") >> lambda_h
      >> catchmark("sigma")  >> sigma_h;
  const geo& omega = uh.get_geo();
  size_t d = omega.dimension();
  size_t k = uh.get_space().degree();
  string approx = (k == 0) ? "empty" : "P"+to_string(k-1)+"d";
  space Tht(omega, "RT"+to_string(k)+"d");
  space Wht(omega, approx, "vector");
  space Mht(omega, "trace_n(RT"+to_string(k)+"d)");
  space Sht = Wht*Mht;
  trial sigma_t (Tht); test tau (Sht);
  auto tau_internal = tau[0], tau_n = tau[1];
  auto coef = beta*pow(h_local(),n);
  form  aht = integrate (dot(sigma_t, tau_internal)
                + on_local_sides (dot(sigma_t,normal())*tau_n));
  field lht = integrate(dot(sigma_h, tau_internal)
                + on_local_sides((dot(sigma_h,normal())
                                + coef*(lambda_h - uh))*tau_n));
  field sigma_ht (Tht);
  problem p (aht);
  p.solve (lht, sigma_ht);
  dout << catchmark("n")      << n << endl
       << catchmark("beta")   << beta << endl
       << catchmark("u")      << uh
       << catchmark("lambda") << lambda_h
       << catchmark("sigma")  << sigma_h
       << catchmark("sigmat") << sigma_ht;
}
```

How to run the program

The compilation and run write:

```
make dirichlet_hdg dirichlet_hdg_post_rt
```

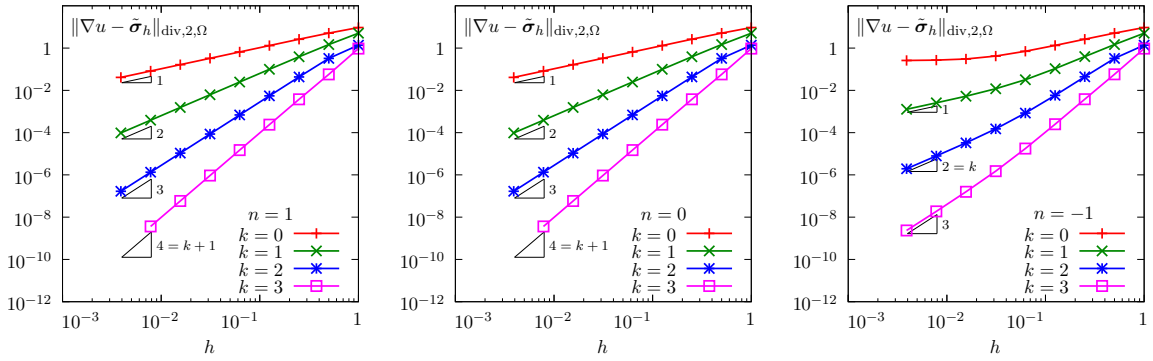


Figure 6.6: Solution post-processing of the hybrid discontinuous Galerkin method: Poisson problem with Dirichlet boundary conditions in 2D geometry. Convergence vs h and k for the post-treated gradient $\tilde{\boldsymbol{\sigma}}_h$ approximate solution. (left) $n = 1$; (center) $n = 0$; (right) $n = -1$.

```
mkgeo_grid -t 10 > square.geo
./dirichlet_hdg square.geo P1d > square.field
./dirichlet_hdg_post_rt < square.field > square-rt.field
field square-rt.field -velocity -mark sigmat
make sinusprod_error_hdg_post_rt
./sinusprod_error_hdg_post_rt < square-rt.field
```

Fig. 6.6 plots the errors vs h and k for a two dimensional geometry. Observe that, when $n = 1$, the approximation $\tilde{\boldsymbol{\sigma}}_h$ of the gradient converges to ∇u with a $k + 1$ rate in $H(\mathrm{div}$ norm, with:

$$\|\boldsymbol{\tau}\|^2_{\mathrm{div},2,\Omega} = \|\boldsymbol{\tau}\|^2_{0,2,\Omega} + \|\mathrm{div}(\boldsymbol{\tau})\|^2_{0,2,\Omega}$$

for any $\boldsymbol{\tau} \in H(\mathrm{div}, \Omega)$. It means that $\mathrm{div}(\tilde{\boldsymbol{\sigma}}_h)$ converges to Δu with a $k + 1$ rate in both L^2 norm. This is optimal with respect to the classical interpolation results for the k-th order Raviart-Thomas element (see section 6.1, page 249). When $n = 0$, the convergence is also optimal while, when $n = -1$ it is suboptimal.

n	u_h	$\boldsymbol{\sigma}_h$	u_h^*	$\tilde{\boldsymbol{\sigma}}_h$
1	k	$k+1$	$k+2$	$k+1$
0	$k+1$	$k+1$	$\begin{cases} 1, & k=0 \\ k+2, & k \geqslant 0 \end{cases}$	$k+1$
-1	$\begin{cases} 0, & k=0 \\ k+1, & k \geqslant 0 \end{cases}$	k	$\begin{cases} 0, & k=0 \\ k+1, & k \geqslant 0 \end{cases}$	k

Table 6.1: Hybrid discontinuous Galerkin method: convergence order versus k for $n \in \{1, 0, -1\}$.

Fig. 6.1 summarizes the convergence orders versus the mesh size h in terms of k and n. Observe that $n = 1$ allows one to obtain an optimal convergence of both the the post-processed solution u_h^* and its gradient $\tilde{\boldsymbol{\sigma}}_h$ for any polynomial degree $k \geqslant 0$. When $n = 0$, it is optimal only for $k \geqslant 1$ while when $n = -1$ it is suboptimal. Thus the HDG method and its postprocessed stage could be considered as a whole. It allows, by solving a linear system for the k-th piecewise polynomial approximation of the Lagrange multiplier only, to obtain a k+2 order approximation of the solution in L^2 and a k+1 approximation of its gradient in $H(\mathrm{div})$.

6.3 Hybrid high order (HHO) methods

The aim of this chapter is to introduce to hybrid high order (HHO) methods within the **Rheolef** environment. For a review of hybrid high order methods, and its link to hybrid discontinuous Galerkin (HDG) methods, see Cockburn et al. [2016]. Let us start by some model problems.

6.3.1 The diffusion problem

Let us consider an anisotropic diffusion problem with homogeneous Dirichlet conditions:

(P): *find u, defined in Ω, such that*

$$\begin{cases} -\mathrm{div}(\boldsymbol{A}\nabla u) = f \quad \text{in} \quad \Omega & \text{(6.8a)} \\ u = 0 \quad \text{on} \quad \partial\Omega & \text{(6.8b)} \end{cases}$$

where the right-hand-side f and the diffusion $\boldsymbol{A}$ are given functions. The diffusion is assumed to be bounded, symmetric, uniformly positive definite matrix-valued function. Observe that when $\boldsymbol{A} = \boldsymbol{I}$, the problem reduces to the usual Poisson problem with homogeneous Dirichlet conditions.

6.3.2 The reconstruction operator

The cornerstone of the HHO method is the reconstruction operator. Let $k \geqslant 0$ and $\ell \in \{k-1, k, k+1\}$ be two integers and let us introduce the following finite element spaces:

$$\begin{aligned} X_h &= \left\{ v_h \in L^2(\Omega) \ ; \ v_{h|K} \in P_\ell, \ \forall K \in \mathscr{T}_h \right\} \\ M_h &= \left\{ \mu_h \in L^2(\mathscr{S}_h) \ ; \ \mu_{h|S} \in P_k, \ \forall S \in \mathscr{S}_h \right\} \\ X_h^* &= \left\{ v_h^* \in L^2(\Omega) \ ; \ v_{h|K}^* \in P_{k+1}, \ \forall K \in \mathscr{T}_h \right\} \end{aligned}$$

The *reconstruction operator* [Cockburn et al., 2016, p. 637] is defined by

$$\begin{array}{rccc} r_h : & X_h \times M_h & \longrightarrow & X_h^* \\ & (u_h, \lambda_h) & \longmapsto & u_h^* = r_h(u_h, \lambda_h) \end{array}$$

where $u_h^* \in X_h^*$ is characterized by a collection of local constrained minimization problems, associated to local Neumann problems similar to those previously introduced for the HDG method, in subsection 6.2.4, page 259:

$$\begin{aligned} u_h^* &= \arg\inf_{v_h^* \in X_h^*} \frac{1}{2} \int_\Omega |\nabla_h (v_h^* - u_h)|_{\boldsymbol{A}}^2 \, \mathrm{d}x - \sum_{K \in \mathscr{T}_h} \int_{\partial K} (\lambda_h - u_h) \ (\boldsymbol{A} \nabla v_h^*) . \boldsymbol{n} \, \mathrm{d}s \\ &\text{subject to} \int_K v_h^* \, \mathrm{d}x = \int_K u_h \, \mathrm{d}x, \quad \forall K \in \mathscr{T}_h \end{aligned}$$

For any symmetric definite positive matrix $\boldsymbol{c}$, we denote by $|.|_{\boldsymbol{c}}$ the anisotropic norm in $\mathbb{R}^d$ defined by $|\boldsymbol{\xi}|_{\boldsymbol{c}}^2 = \boldsymbol{\xi}.(\boldsymbol{c}\boldsymbol{\xi})$ for all $\boldsymbol{\xi} \in \mathbb{R}^d$. The constraint corresponds to the closure for the local Neumann problems, as otherwise the solution would be defined up to a constant in each element $\forall K \in \mathscr{T}_h$. This constrained minimization problem is not convenient for the numerical resolution: we prefer to express it as a saddle-point problem by introducing a Lagrange multiplier $\zeta_h \in Z_h$ with

$$Z_h = \left\{ \xi_h \in L^2(\Omega) \ ; \ \xi_{h|K} \in P_0, \ \forall K \in \mathscr{T}_h \right\}$$

The corresponding Lagrangian writes

$$\begin{aligned} L_\gamma(u^*, \zeta) &= \frac{1}{2} \int_\Omega |\nabla_h (u^* - u_h)|_{\boldsymbol{A}}^2 \, \mathrm{d}x - \sum_{K \in \mathscr{T}_h} \int_{\partial K} (\lambda_h - u_h) \ (\boldsymbol{A} \nabla u^*) . \boldsymbol{n} \, \mathrm{d}s \\ &\quad + \int_\Omega (u^* - u_h) \, \zeta \, \mathrm{d}x - \int_\Omega \frac{\gamma}{2} \zeta^2 \, \mathrm{d}x \end{aligned} \tag{6.9}$$

Observe that the first two terms of the Lagrangian correspond to the minimization functional. The third term is associated to the constraint with its associated Lagrangian multiplier. When $\gamma = 0$, the fourth term vanishes and thus, the saddle-point of the Lagrangian is equivalent to the previous constrained minimization problem. The fourth term, when the augmentation parameter $\gamma > 0$ will be discussed later: in that case, L_γ is related to the augmented Lagrangian. The saddle-point problem expresses:

$$(u_h^*, \zeta_h) = \arg\inf_{v_h^* \in X_h^*} \sup_{\xi_h Z_h} L_\gamma(v_h^*, \xi_h)$$

Since L_γ is differentiable, its saddle point is also characterized as the unique solution of the following collection of local variational problems:

$(FV^*)_h$: *let* $(u_h, \lambda_h) \in X_h \times M_h$ *being given, find* $(u_h^*, \zeta_h) \in X_h^* \times Z_h$ *such that*

$$\begin{cases} \displaystyle\int_K \nabla u_h^* . (\boldsymbol{A} \nabla v_h^*) \, \mathrm{d}x + \int_K v_h^* \zeta_h \, \mathrm{d}x = \int_K \nabla u_h . (\boldsymbol{A} \nabla v_h^*) \, \mathrm{d}x + \int_{\partial K} (\lambda_h - u_h) \ (\boldsymbol{A} \nabla v_h^*) . \boldsymbol{n} \, \mathrm{d}s \\ \displaystyle\int_K u_h^* \xi_h \, \mathrm{d}x - \int_K \gamma \zeta_h \xi_h \, \mathrm{d}x = \int_K u_h \xi_h \, \mathrm{d}x \end{cases}$$

for all $K \in \mathscr{T}_h$ and all $(v_h^*, \xi_h) \in X_h^* \times Z_h$. Observe that, for any $K \in \mathscr{T}_h$, we can choose the test function v_h^* as equal to 1 in K and zero elsewhere in Ω. Then, the first equation leads to $\zeta_h = 0$ in K. Since this could be repeated for any $K \in \mathscr{T}_h$, we obtain $\zeta_h = 0$. As a consequence, the term factored by $\gamma \geqslant 0$ has no influence upon the solution (u_h^*, ζ_h) which is independent upon γ: in this sense, this is not a *penalization* parameter, but an *augmented Lagrangian* parameter. Up to now, we assume $\gamma > 0$: we will show how it facilitates the numerical resolution of the system.

Let, for all $u^*, v \in X_h^*$, $\zeta, \xi \in Z_h^*$, $u \in X_h$ and $\lambda \in M_h$:

$$a_*(u^*, v^*) = \int_\Omega \nabla_h u^* . (\boldsymbol{A} \nabla_h v^*) \, \mathrm{d}x \tag{6.10a}$$

$$b_*(u^*, \xi) = \int_\Omega u^* \xi \, \mathrm{d}x \tag{6.10b}$$

$$c_*(\zeta, \xi) = \int_\Omega \gamma \, \zeta \, \xi \, \mathrm{d}x \tag{6.10c}$$

$$\check{a}(u, v^*) = \int_\Omega \nabla_h u . (\boldsymbol{A} \nabla_h v^*) \, \mathrm{d}x - \sum_{K \in \mathscr{T}_h} \int_{\partial K} u \ (\boldsymbol{A} \nabla_h v^*) . \boldsymbol{n} \, \mathrm{d}s$$

$$b(u, \xi) = \int_\Omega u \, \xi \, \mathrm{d}x$$

$$e_*(u^*, \mu) = \sum_{K \in \mathscr{T}_h} \int_{\partial K} \mu \ (\boldsymbol{A} \nabla_h u^*) . \boldsymbol{n} \, \mathrm{d}s$$

Then, the previous reconstruction problem writes:

$(FV^*)_h$: *let* $(u_h, \lambda_h) \in X_h \times M_h$ *being given, find* $(u_h^*, \zeta_h) \in X_h^* \times Z_h$ *such that*

$$\begin{aligned} a_*(u_h^*, v_h^*) + b_*(v_h^*, \zeta_h) &= \check{a}(u_h, v_h^*) + e_*(v_h^*, \lambda_h), & \forall v_h^* \in X_h^* \\ b_*(u_h^*, \xi_h) - c_*(\zeta_h, \xi_h) &= b(u_h, \xi_h), & \forall \xi_h \in Z_h \end{aligned}$$

By identifying the bilinear forms and the associated matrix, the previous linear problem also expreses in matrix-vector form:

$$\begin{pmatrix} a_* & b_*^T \\ b_* & -c_* \end{pmatrix} \begin{pmatrix} u_h^* \\ \zeta_h \end{pmatrix} = \begin{pmatrix} \check{a} & e_* \\ b & 0 \end{pmatrix} \begin{pmatrix} u_h \\ \lambda_h \end{pmatrix}$$

A carreful study shows that the matrix on the left-hand-side has a block-diagonal structure at the element level and its inverse can be easily constructed explicitly by inverting on the fly the

element matrix during the assembly process. Then, the result of reconstruction operator can be obtained at low computational cost, by simple matrix-vector multiplication.

A further improvement can be obtained. Observe that the matrix on the left-hand-side is symmetric but definite: at the element level, it admits both positive and negative eigenvalues, due to the saddle point structure.

Expanding the previous linear system, we obtain

$$\begin{cases} a_* u_h^* + b_*^T \zeta_h = \check{a} u_h + e_*^T \lambda_h \\ b_* u_h^* - c_* \zeta_h = b u_h \end{cases}$$

Since the c_* matrix is diagonal, when choosing $\gamma > 0$, the ζ unknown could be eliminated at the element level:

$$\begin{aligned} \zeta_h &= c_*^{-1} \left(b_* u_h^* - b u_h \right) \\ u_h^* &= S_*^{-1} \left(S_h u_h + e_*^T \lambda_h \right) \\ \text{with } S_* &= a_* + b_*^T c_*^{-1} b_* \\ S_h &= \check{a} + b_*^T c_*^{-1} b \end{aligned} \tag{6.11}$$

Since we known that $\zeta_h = 0$, the corresponding computation has no importance, expect for checking purpose, and let us turn to those of u_h^*. The Schur complement S_* is still block-diagonal at the element level and, moreover, it is then symmetric definite positive. This is of major importance when considering iterative solvers, e.g. for tridimensional geometries. Let us turn to the practical choice of $\gamma > 0$. An investigation of (6.10a) shows that entries of a_* scales as $\mathcal{O}(h^{d-2})$. The h^d comes from the integration measure $\mathrm{d}x$ and the h^{-2},from the two derivations ∇_h. Conversely, assuming γ independent upon the mesh size h, the entries for b_* and c_*, from (6.10b) and (6.10c), respectively, scales as $\mathcal{O}(h^d)$ since there is no derivation in b_* and c_*, Then, $b_*^T c_*^{-1} b_*$ scales as $\mathcal{O}(h^d)$. Then, the expression (6.11) for s_s is not equilibrated. Choosing $\gamma = \alpha h^2$, with $\alpha > 0$ independent upon h, leads to $b_*^T c_*^{-1} b_*$ that scales as $\mathcal{O}(h^{d-2})$, i.e. as a_*. This h-dependent choice for γ dramatically improves the condition number for S_*. Note that [Pietro and Droniou, 2020, p. 454], eqn (B.7), proposed a similar implementation, despites it based on different notations and concepts.

File 6.7: reconstruction_hho.cc

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
#include "sinusprod.h"
#include "diffusion_isotropic.h"
int main(int argc, char**argv) {
  environment rheolef (argc, argv);
  geo omega (argv[1]);
  string Pkd = (argc > 2) ? argv[2] : "P0",
         Pld = (argc > 3) ? argv[3] :  Pkd;
  space  Xh (omega, Pld),
         Mh (omega["sides"], Pkd);
  size_t l = Xh.degree(), k = Xh.degree(), d = omega.dimension();
  check_macro(l == k-1 || l == k || l == k+1,
    "invalid (k,l) = ("<<k<<","<<l<<")");
  space Xhs(omega, "P"+to_string(k+1)+"d"),
        Zh (omega, "P0");
  trial u(Xh), lambda(Mh), us(Xhs), zeta (Zh);
  test  v(Xh), mu    (Mh), vs(Xhs), xi   (Zh);
  auto as = lazy_integrate (dot(grad_h(us),A(d)*grad_h(vs)));
  auto bs = lazy_integrate (us*xi);
  auto cs = lazy_integrate (pow(h_local(),2)*zeta*xi);
  auto m  = lazy_integrate (u*v);
  form ms = lazy_integrate (lambda*mu);
  auto inv_cs = inv(cs);
  auto inv_m  = inv(m);
  auto inv_S  = inv(as + trans(bs)*inv_cs*bs);
  auto llh = lazy_integrate (u_exact(d)*v);
  field rhs = lazy_integrate (u_exact(d)*mu);
  field pi_Xh_u = inv_m*llh;
  field pi_Mh_lambda(Mh);
  problem pms (ms);
  pms.solve (rhs, pi_Mh_lambda);
  field lh = lazy_integrate (dot(grad_h(pi_Xh_u),A(d)*grad_h(vs))
                    + on_local_sides((pi_Mh_lambda-pi_Xh_u)
                                     *dot(normal(),A(d)*grad_h(vs))));
  auto kh = lazy_integrate (pi_Xh_u*xi);
  auto rh = lh + bs.trans_mult(inv_cs*kh);
  field us_h = inv_S*rh;
  field zeta_h = inv_cs*(bs*us_h - kh);
  dout << catchmark("us")   << us_h
       << catchmark("zeta") << zeta_h;
}
```

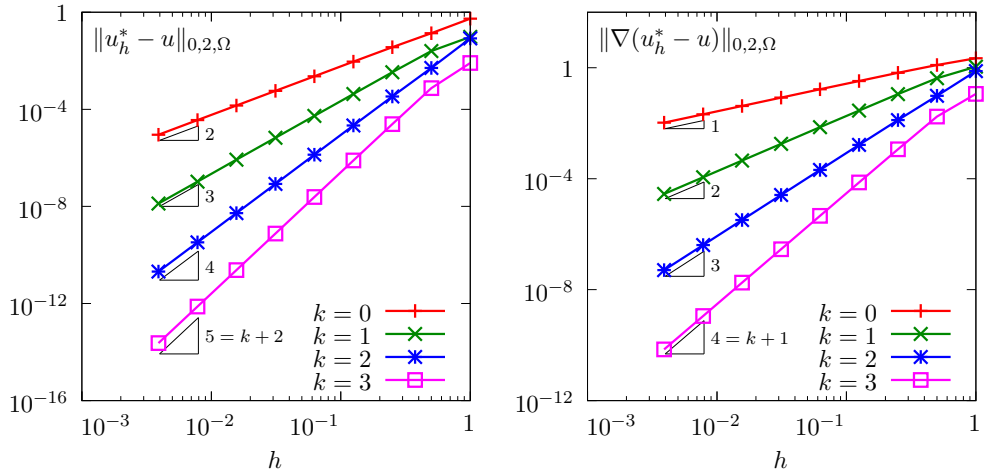


Figure 6.7: Hybrid hight-order (HHO) method: error vs h and k for the reconstruction operator.

How to run the program

The compilation and run write:

```
make reconstruction_hho sinusprod_error_hho_reconstruction
mkgeo_grid -t 10 > square.geo
./reconstruction_hho square.geo P1d P1d | ./sinusprod_error_hho_reconstruction
```

The code '`reconstruction_hho.cc`' first computes u_h and λ_h as L^2 projections on X_h and M_h, respectively, of a given function:

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

Then, it builds the reconstruction $u_h^* = r_h(u_h, \lambda_h)$ and output the result. Next, the code '`sinusprod_error_hho_reconstruction.cc`' shows the L^2 and H^1 error for the reconstruction u_h^*. The files '`sinusprod.h`', '`diffusion_isotropic.h`', and '`sinusprod_error_hho_reconstruction.cc`', that defines u and $\boldsymbol{A}$, and that compute the errors, respectively, are not listed here but is available in the **Rheolef** example directory. Fig. 6.7 plots the error versus mesh size h and the polynomial order k: the convergence order is $k+2$ in the L^2 norm and $k+1$ in the H^1 one, as expected (see Cockburn et al. [2016], eqn (2.9) or di Pietro and Ern [2015], eqn (3)).

6.3.3 The projections

Let π_{X_h} denotes the L^2 projection from $L^2(\Omega)$ into X_h. For all $u^* \in L^2(\Omega)$, it is defined as $\pi_{X_h}(u^*) = \check{u}_h \in X_h$, where $\check{u}_h$ is the solution of the following quadratic minimization problem:

$$\check{u}_h \quad = \quad \inf_{\check{v}_h \in X_h} \frac{1}{2} \int_\Omega (u^* - \check{v}_h)^2 \mathrm{d}x$$

Let us consider the special case of computing the projection $\check{u}_h = \pi_{X_h}(u_h^*) \in X_h$ of an element $u_h^* \in X_h^*$. The following bilinear forms are introduced:

$$\begin{aligned} m(\check{u}, \check{v}) &= \int_\Omega \check{u}\, \check{v}\, \mathrm{d}x \\ d_*(u^*, \check{v}) &= \int_\Omega u^*\, \check{v}\, \mathrm{d}x \end{aligned}$$

Then, the projection reduces to:

let $u_h^* \in X_h^*$ being given, find $\check{u}_h \in X_h$ such that

$$m(\check{u}_h, \check{v}_h) \quad = \quad d_*(u_h^*, \check{v}_h), \quad \forall \check{v}_h \in X_h$$

Note that the matrix associated to the bilinear form m from $X_h \times X_h$ to $\mathbb{R}$ has a block-diagonal structure at the element level and can be easily inverted on the fly during the assembly process. Then, the result of projection operator can be obtained by simple matrix-vector multiplication.

Let us reintroduce the following space $\tilde{M}_h$, already defined in (6.7), page 262, in the context of the HDG method:

$$\tilde{M}_h \quad = \quad \prod_{K \in \mathscr{T}_h} \{ \tilde{\boldsymbol{\tau}}_{|\partial K} . \boldsymbol{n} \ ; \ \tilde{\boldsymbol{\tau}} \in RT_k(K) \} \quad = \quad \prod_{K \in \mathscr{T}_h} \prod_{S \subset \partial K} P_k(S)$$

Let $\pi_{\tilde{M}_h}$ denote the L^2 projection into $\tilde{M}_h$. We consider the special case of computing the projection $\tilde{\delta}_h = \pi_{\tilde{M}_h}(u_h^*) \in \tilde{M}_h$ of an element $u_h^* \in X_h^*$. It is defined as

$$\tilde{\delta}_h \quad = \quad \inf_{\tilde{\varphi}_h \in \tilde{M}_h} \frac{1}{2} \sum_{K \in \mathscr{T}_h} \int_{\partial K} (u_h^* - \tilde{\varphi}_h)^2 \mathrm{d}s$$

Note that $u_h^* \in X_h^*$ is bi-valued and P_{k+1} on all internal side while its projection $\tilde{\delta}_h = \pi_{\tilde{M}_h}(u_h^*) \in \tilde{M}_h$ is also bi-valued but only P_k. The following bilinear forms are introduced:

$$\begin{aligned}\tilde{m}(\tilde{\delta},\tilde{\varphi}) &= \sum_{K\in\mathscr{T}_h}\int_{\partial K}\tilde{\delta}\,\tilde{\varphi}\,\mathrm{d}s\\ \tilde{d}_*(u^*,\tilde{\varphi}) &= \sum_{K\in\mathscr{T}_h}\int_{\partial K}u^*\,\tilde{\varphi}\,\mathrm{d}s\end{aligned}$$

Then, the projection reduces to:

let $u_h^* \in X_h^*$ being given, find $\tilde{\delta}_h \in \tilde{M}_h$ such that

$$\tilde{m}(\tilde{\delta}_h,\tilde{\varphi}_h) = \tilde{d}(u_h^*,\tilde{\varphi}_h),\quad \forall\tilde{\varphi}_h \in \tilde{M}_h$$

Here also, the matrix associated to the bilinear form m_s from $\tilde{M}_h \times \tilde{M}_h$ to $\mathbb{R}$ has a block-diagonal structure at the element level and can be easily inverted on the fly during the assembly process. Then, the result of projection operator can be obtained by simple matrix-vector multiplication.

6.3.4 The discrete problem statement

The discretization of the diffusion problem (6.8a)-(6.8b) by the hybrid high-order (HHO) method [Cockburn et al., 2016, p. 639] is expressed here as a minimization problem:

$$(u_h,\lambda_h) = \arg\min_{v_h\in X_h}\max_{\mu_h\in M_h(0)} L_1(v_h,\mu_h)$$

with

$$\begin{aligned}L_1(u,\lambda) &= \int_\Omega\left(\frac{1}{2}|\nabla_h r_h(u,\lambda)|_{\boldsymbol{A}}^2 - f\,u\right)\mathrm{d}x\\ &\quad+\frac{1}{2}\sum_{K\in\mathscr{T}_h}\sum_{S\subset\partial K}\int_S \beta\,h_S^{-1}\left(\pi_{\tilde{M}_h}\{u-\lambda+(I-\pi_{X_h})\,r_h(u,\lambda)\}\right)^2\mathrm{d}s\\ M_h(0) &= \left\{\mu_h\in M_h\ ;\ \mu_{h|S}=0,\ \forall S\subset\partial\Omega\right\}\end{aligned}$$

In the definition of L_1, the first term is related to the consistency and the second one to the stabilization, with a $\beta\,h_S^{-1}$ penalization factor (see [Cockburn et al., 2016, p. 637], eqns (2.4)-(2.5)), while h_S denotes a local mesh characteristic length in the side $S\in\mathscr{S}_h$. Here $\beta>0$ is the *penalization* parameter, which is a constant independent of the mesh size h. The space $M_h(0)$ for the Lagrange multiplier λ_h is introduced in order to impose the homogeneous Dirichlet condition (6.8b) as a constraint.

The expression of L_1 is not convenient for practical implementation, as it involves both the reconstruction operator r_h and the projections π_{X_h} and $\pi_{\tilde{M}_h}$. Let us introduce the reconstruction $u_h^* = r_h(u_h,\mu_h)\in X_h^*$ as an auxilliary variable. The constraint $u_h^* - r_h(u_h,\mu_h)=0$ is imposed via a Lagrange multiplier $v_h^*\in X_h^*$. Both the two additional variables u_h^* and v_h^* have a constrained average value and we also introduce two associated Lagrangian multipliers: one is denoted by $\zeta_{1,h}\in Z_h$, associated to the imposition of a zero element-wise average value for $u_h^*-u_h$. the second, $\zeta_{2,h}\in Z_h$, is associated to the imposition of a zero element-wise average value for v_h^*.

Similarly to the augmented Lagrangian (6.9), we introduce a new Lagrangian:

$$\begin{aligned} L_2(u,u^*;\ \lambda,v^*,\zeta_1,\zeta_2) \quad = \quad & \int_\Omega \left(\frac{1}{2}|\nabla_h u^*|^2_{\boldsymbol{A}} - f\,u\right)\,\mathrm{d}x \\ & + \frac{1}{2}\sum_{K\in\mathscr{T}_h}\sum_{S\subset\partial K}\int_S \beta\, h_S^{-1}\left(\pi_{\tilde{M}_h}\{u-\lambda+u^*-\pi_{X_h}(u^*)\}\right)^2\,\mathrm{d}s \\ & + \int_\Omega \nabla_h(u^*-u).\left(\boldsymbol{A}\nabla_h v^*\right)\,\mathrm{d}x \ + \sum_{K\in\mathscr{T}_h}\int_{\partial K}(u-\lambda)\ \left(\boldsymbol{A}\nabla v^*\right).\boldsymbol{n}\,\mathrm{d}s \\ & + \int_\Omega \left\{(u^*-u)\,\zeta_1 + v^*\zeta_2 - \frac{\alpha\,h^{-2}}{2}\left(\zeta_1^2-\zeta_2^2\right)\right\}\,\mathrm{d}x \end{aligned}$$

In the last term, also have introduced the augmented Lagrangian term with coefficient $\alpha>0$, as introduced in the previous paragraph on the reconstruction operator. The previous problem in terms of L_1 can be equivalently expressed as a saddle point for L_2, with six unknowns:

$$(u_h,u_h^*;\ \lambda_h,v_h^*,\zeta_{1,h},\zeta_{2,h}) \quad = \quad \underset{\substack{v_h\in X_h\\ v_h^*\in X_h^*}}{\arg\inf}\ \underset{\substack{\mu_h\in M_h(0)\\ w_h^*\in X_h^*\\ \xi_{1,h}\in Z_h\\ \xi_{2,h}\in Z_h}}{\sup}\ L_2(v_h,v_h^*;\ \mu_h,w_h^*,\xi_{1,h},\xi_{2,h})$$

The reconstruction operator r_h has been eliminated, but the previous expression of L_2 still involves the two projectors π_{X_h} and $\pi_{\tilde{M}_h}$. These explicit calls to the projectors are not convenient for the practical implementation. Let us introduce the two auxiliary variables $\check{u}_h=\pi_{X_h}(u_h^*)\in X_h$ and $\tilde{\delta}_h=\pi_{\tilde{M}_h}(u-\lambda+u_h^*-\check{u}_h)\in\tilde{M}_h$ as two constraints and the two corresponding Lagrangian multipliers, $\check{v}_h\in X_h$ and $\tilde{\gamma}_h\in\tilde{M}_h$, respectively. Then, we obtain the following Lagrangian:

$$\begin{aligned} L(u,u^*,\check{u},\tilde{\delta};\ \lambda,v^*,\zeta_1,\zeta_2,\check{v},\tilde{\gamma}) \quad = \quad & \int_\Omega \left(\frac{1}{2}|\nabla_h u^*|^2_{\boldsymbol{A}} - f\,u\right)\,\mathrm{d}x + \frac{1}{2}\sum_{K\in\mathscr{T}_h}\sum_{S\subset\partial K}\int_S \beta\, h_S^{-1}\,\tilde{\delta}^2\,\mathrm{d}s \\ & + \int_\Omega \nabla_h(u^*-u).\left(\boldsymbol{A}\nabla_h v^*\right)\,\mathrm{d}x + \sum_{K\in\mathscr{T}_h}\int_{\partial K}(u-\lambda)\ \left(\boldsymbol{A}\nabla v^*\right).\boldsymbol{n}\,\mathrm{d}s \\ & + \int_\Omega \left\{(u^*-u)\,\zeta_1 + v^*\zeta_2 - \frac{\alpha\,h^{-2}}{2}\left(\zeta_1^2-\zeta_2^2\right)\right\}\,\mathrm{d}x \\ & + \int_\Omega (u^*-\check{u})\,\check{v}\,\mathrm{d}x + \sum_{K\in\mathscr{T}_h}\sum_{S\subset\partial K}\int_S (u-\lambda+u^*-\check{u}-\tilde{\delta})\,\tilde{\gamma}\,\mathrm{d}s \end{aligned}$$

and the previous minimization problem can be equivalently expressed as a saddle point one with ten unknowns:

$$\begin{aligned} & (u_h,u_h^*,\check{u}_h,\tilde{\delta}_h;\ \lambda_h,v_h^*,\zeta_{1,h},\zeta_{2,h},\check{v}_h,\tilde{\gamma}_h) \\ & = \underset{\substack{w_h\in X_h\\ w_h^*\in X_h^*\\ \check{w}_h\in X_h\\ \tilde{\varphi}_h\in\tilde{M}_h}}{\arg\inf}\ \underset{\substack{\mu_h\in M_h(0)\\ y_h^*\in X_h^*\\ \xi_{1,h}Z_h\\ \xi_{2,h}Z_h\\ \check{y}_h\in X_h\\ \tilde{\kappa}_h\in\tilde{M}_h}}{\sup}\ L(w_h,w_h^*,\check{w}_h,\tilde{\varphi}_h;\ \mu_h,y_h^*,\xi_{1,h},\xi_{2,h},\check{y}_h,\tilde{\kappa}_h) \end{aligned}$$

Since L is differentiable and quadratic, its saddle point is characterized as the unique solution of a variational and linear problem:

$(FV)_h$: *find* $(u_h^*,\zeta_{1,h},\tilde{\delta}_h;\ v_h^*,\zeta_{2,h},\tilde{\gamma}_h;\ \check{u}_h,\check{v}_h,u_h;\ \lambda)\in\left(X_h^*\times Z_h\times\tilde{M}_h\right)^2\times X_h^3\times M_h(0)$ *such*

that

$$
\begin{aligned}
&\int_\Omega \nabla_h u^*.(\boldsymbol{A}\nabla_h w^*)\,\mathrm{d}x - \int_\Omega f\,w\,\mathrm{d}x + \sum_{K\in\mathscr{T}_h}\sum_{S\subset\partial K}\int_S \beta\, h_S^{-1}\,\tilde{\delta}\,\tilde{\varphi}\,\mathrm{d}s \\
&+ \int_\Omega \{\nabla_h(u^*-u).(\boldsymbol{A}\nabla_h y^*) + \nabla_h(w^*-w).(\boldsymbol{A}\nabla_h v^*)\}\,\mathrm{d}x \\
&+ \sum_{K\in\mathscr{T}_h}\sum_{S\subset\partial K}\int_S \{u\,(\boldsymbol{A}\nabla y^*).\boldsymbol{n} + w\,(\boldsymbol{A}\nabla v^*).\boldsymbol{n}\}\,\mathrm{d}s \\
&+ \int_\Omega \left\{(u^*-u)\xi_1 + (w^*-w)\zeta_1 + v^*\xi_2 + y^*\zeta_2 - \alpha\, h^{-2}(\zeta_1\xi_1 - \zeta_2\xi_2)\right\}\,\mathrm{d}x \\
&+ \int_\Omega \{(u^*-\check{u})\check{y} + (w^*-\check{w})\check{v}\}\,\mathrm{d}x \\
&+ \sum_{K\in\mathscr{T}_h}\sum_{S\subset\partial K}\int_S \left\{(u+u^*-\check{u}-\tilde{\delta})\tilde{\kappa} + (w+w^*-\check{w}-\tilde{\varphi})\tilde{\gamma}\right\}\,\mathrm{d}x \\
&- \sum_{K\in\mathscr{T}_h}\sum_{S\subset\partial K}\int_S \{\lambda\,(\boldsymbol{A}\nabla y^*).\boldsymbol{n} + \mu\,(\boldsymbol{A}\nabla v^*).\boldsymbol{n}\}\,\mathrm{d}s \\
&- \sum_{K\in\mathscr{T}_h}\sum_{S\subset\partial K}\int_S (\lambda\,\tilde{\kappa} + \mu\,\tilde{\gamma})\,\mathrm{d}s \\
=\;& 0
\end{aligned}
$$

for all $(w_h^*, \xi_{1,h}, \tilde{\varphi}_h;\ y_h^*, \xi_{2,h}, \tilde{\kappa}_h;\ \check{w}_h, \check{y}_h, w_h;\ \mu) \in \left(X_h^* \times Z_h \times \tilde{M}_h\right)^2 \times X_h^3 \times M_h(0)$. The h subscripts for the unknowns and test functions have been omitted for simplicity in the variational expression. This system could appear as complex. But observe that all discretization are piecewise discontinuous at the element level except those for $\lambda_h \in M_h(0)$. Then, all variables except this one could be eliminated at the element level. In order to process to this elimination, the previous variational formulation is rearranged as a matrix block structure:

$$
\left(\begin{array}{c||cccc|cccc|cc}
 & u^* & \zeta_1 & v^* & \zeta_2 & \tilde{\delta} & \tilde{\gamma} & \check{u} & \check{v} & u & \lambda \\
\hline\hline
w^* & a_* & b_*^T & a_* & & & \tilde{d}_*^T & & d_*^T & & \\
\xi_1 & b_* & -c_* & & & & & & & -d & \\
y^* & a_* & & & b_*^T & & & & & -\check{a} & -e_*^T \\
\xi_2 & & & b_* & c_* & & & & & & \\
\hline\hline
\tilde{\varphi} & & & & & \tilde{c} & -\tilde{m} & & & & \\
\tilde{\kappa} & \tilde{d}_* & & & & -\tilde{m} & & -\tilde{d} & & \tilde{d} & -\tilde{e}^T \\
\check{w} & & & & & & -\check{d}^T & & -m & & \\
\check{y} & d_* & & & & & & -m & & & \\
\hline\hline
w & & -d^T & -\check{a}^T & & & \tilde{d}^T & & & & \\
\hline\hline
\mu & & & -e_* & & & -\tilde{e} & & & &
\end{array}\right)
\left(\begin{array}{c}
u_h^* \\ \hline \zeta_{1,h} \\ \hline v_h^* \\ \hline \zeta_{2,h} \\ \hline\hline \tilde{\delta}_h \\ \hline \tilde{\gamma}_h \\ \hline \check{u}_h \\ \hline \check{v}_h \\ \hline\hline u_h \\ \hline\hline \lambda_h
\end{array}\right)
=
\left(\begin{array}{c}
0 \\ \hline 0 \\ \hline 0 \\ \hline 0 \\ \hline\hline 0 \\ \hline 0 \\ \hline 0 \\ \hline 0 \\ \hline\hline \ell_h \\ \hline\hline 0
\end{array}\right)
$$

The matrix are identified to the corresponding forms and we have introduce thirteen bilinear forms and a linear form. A first group of five bilinear forms is associated to symmetric positive matrix plus the right-hand-side ℓ_h:

$$
\begin{aligned}
a_*(u^*, w^*) &= \int_\Omega \nabla_h u^*.\,(\boldsymbol{A}\nabla_h w^*)\,\mathrm{d}x & \ell_h(w) &= \int_\Omega f\,w\,\mathrm{d}x \\
m(u, w) &= \int_\Omega u\,w\,\mathrm{d}x & \tilde{m}(\tilde{\delta}, \tilde{\varphi}) &= \sum_{K\in\mathscr{T}_h}\int_{\partial K} \tilde{\delta}\,\tilde{\varphi}\,\mathrm{d}s \\
c_*(\zeta, \xi) &= \int_\Omega \alpha\, h^{-2}\zeta\,\xi\,\mathrm{d}x & \tilde{c}(\tilde{\delta}, \tilde{\varphi}) &= \sum_{K\in\mathscr{T}_h}\int_{\partial K} \beta\, h_S^{-1}\tilde{\delta}\,\tilde{\varphi}\,\mathrm{d}s
\end{aligned}
$$

All the associated matrix are symmetric definite positive, and then invertible, except a_*, which is only symmetric positive but has a zero eigenvalue. The second group corresponds to rectangular matrix:

$$b_*(u^*,\xi) = \int_\Omega u^*\xi \,\mathrm{d}x$$

$$d(u,\xi) = \int_\Omega u\,\xi \,\mathrm{d}x \qquad \tilde{d}(u,\tilde{\varphi}) = \sum_{K\in\mathscr{T}_h}\int_{\partial K} u\,\tilde{\varphi}\,\mathrm{d}s$$

$$d_*(u^*,w) = \int_\Omega u^*\,w\,\mathrm{d}x \qquad \tilde{d}_*(u^*,\tilde{\varphi}) = \sum_{K\in\mathscr{T}_h}\int_{\partial K} u^*\,\tilde{\varphi}\,\mathrm{d}s$$

$$\check{a}(u,w^*) = \int_\Omega \nabla_h u.\left(\boldsymbol{A}\nabla_h w^*\right)\,\mathrm{d}x - \sum_{K\in\mathscr{T}_h}\int_{\partial K} u\,\left(\boldsymbol{A}\nabla_h w^*\right).\boldsymbol{n}\,\mathrm{d}s$$

These two groups admit a block-diagonal structure at the element level, due to the piecewise discontinuous character of the trial and test functions. In the following third group, bilinear forms corresponds to rectangular matrix with non-block-diagonal structure, due to the inter-element continuity of $\mu \in Mh$:

$$e_*(v^*,\mu) = \sum_{K\in\mathscr{T}_h}\int_{\partial K} \mu\,\left(\boldsymbol{A}\nabla_h v^*\right).\boldsymbol{n}\,\mathrm{d}s \qquad \tilde{e}(\tilde{\gamma},\mu) = \sum_{K\in\mathscr{T}_h}\int_{\partial K} \tilde{\gamma}\,\mu\,\mathrm{d}s$$

Let us process to the elimination of all the unknowns except λ_h. For simplicity, the matrix subequations are designated by the corresponding test-function surrounded by parenthesis, such as (w^*) for the first one. Let us start by eliminating some variables related to projections: from $(\check{y})$, $(\tilde{w})$ and $(\tilde{\varphi})$, we get, respectively:

$$\check{u} = \quad m^{-1}d_*u^* \tag{6.12a}$$

$$\check{v} = -m^{-1}\tilde{d}^T\tilde{\gamma} \tag{6.12b}$$

$$\tilde{\gamma} = \tilde{m}^{-1}\tilde{c}\,\tilde{\delta} \tag{6.12c}$$

The following subexpression will be frequently involved:

$$d_*^T\check{v} + \tilde{d}_*^T\tilde{\gamma} = D^T\tilde{\delta}$$
$$\text{with} \quad D = \tilde{c}\,\tilde{m}^{-1}\tilde{D}_*$$
$$\tilde{D}_* = \tilde{d}_* - \tilde{d}\,m^{-1}d_*$$

Next, let us turn to variables related to the reconstruction. from (ξ_1) and (w^*), we get: we obtain successively:

$$\zeta_1 = c_*^{-1}\left(b_*u^* - du\right) \tag{6.12d}$$

$$u_* = S_*^{-1}\left(-a_*v^* - D^T\tilde{\delta} + b_*^Tc_*^{-1}d\,u\right) \tag{6.12e}$$

$$\text{with} \quad S_* = a_* + b_*^Tc_*^{-1}b_*$$

Conversely, from (ξ_2), (y^*) and (6.12e)

$$\zeta_2 = -c_*^{-1}b_*v^* \tag{6.12f}$$

$$v^* = T^{-1}\left(-a_*S_*^{-1}D^T\tilde{\delta} + Ru - e_*^T\lambda\right) \tag{6.12g}$$

$$\text{with} \quad T = a_*S_*^{-1}a_* + b_*^Tc_*^{-1}b_*$$
$$R = a_*S_*^{-1}b_*^Tc_*^{-1}d - \check{a}$$

Back-substituting in (6.12d)-(6.12e):

$$u_* = -\left(S_*^{-1} - S_*^{-1}a_*T^{-1}a_*S_*^{-1}\right)D^T\tilde{\delta} + S_*^{-1}\left(b_*^T c_*^{-1}d - a_*T^{-1}R\right)u + S_*^{-1}a_*T^{-1}e_*^T\lambda \tag{6.12h}$$

$$\begin{aligned}\zeta_1 = &-c_*^{-1}b_*\left(S_*^{-1} - S_*^{-1}a_*T^{-1}a_*S_*^{-1}\right)D^T\tilde{\delta}\\ &+ c_*^{-1}\left[b_*S_*^{-1}\left(b_*^T c_*^{-1}d - a_*T^{-1}R\right) - d\right]u\\ &+ c_*^{-1}b_*S_*^{-1}a_*T^{-1}e_*^T\lambda\end{aligned} \tag{6.12i}$$

Then, replacing (6.12c), (6.12g) and (6.12i) in (w), we obtain:

$$A_c u + \tilde{E}\tilde{\delta} - R^T T^{-1}e_*^T\lambda = \ell \tag{6.12j}$$

$$\begin{aligned}\text{with}\quad A_c &= R^T T^{-1}R + d^T\left(c_*^{-1} - c_*^{-1}b_*S_*^{-1}b_*^T c_*^{-1}\right)d\\ &= R^T T^{-1}R\\ \text{since}\quad c_* &= b_*S_*^{-1}b_*^T\\ \tilde{E} &= \tilde{d}^T\tilde{m}^{-1}\tilde{c} + \breve{a}^T T^{-1}a_*S_*^{-1}D^T + d^T c_*^{-1}b_*\left(S_*^{-1} - S_*^{-1}a_*T^{-1}a_*S_*^{-1}\right)D^T\end{aligned}$$

It remains to express $\tilde{\delta}$ in terms of u and λ. From $(\tilde{\kappa})$ and (6.12a):

$$\tilde{m}\tilde{\delta} - \tilde{D}_* u^* - \tilde{d}u + \tilde{e}^T\lambda = 0$$

Left-multiplying by $\tilde{c}\tilde{m}^{-1}$ and substituting (6.12h), we get

$$\tilde{\delta} = \tilde{M}^{-1}\left(\tilde{E}^T u + F^T\lambda\right) \tag{6.12k}$$

$$\begin{aligned}\tilde{M} &= \tilde{c} + D\left(S_*^{-1} - S_*^{-1}a_*T^{-1}a_*S_*^{-1}\right)D^T\\ F &= e_*T^{-1}a_*S_*^{-1}D^T - \tilde{e}\,\tilde{m}^{-1}\tilde{c}\end{aligned}$$

Replacing in (6.12j), we get

$$Au + B^T\lambda = \ell \tag{6.12l}$$

$$\begin{aligned}\text{with}\quad A &= A_c + A_s\\ A_s &= \tilde{E}\tilde{M}^{-1}\tilde{E}^T\\ B &= F\tilde{M}^{-1}\tilde{E}^T - e_*T^{-1}R\end{aligned}$$

Here, A_c represents the consistency part of the operator A while A_s is the stabilization part, i.e. the penalization. Finally, (μ) becomes

$$Bu + C\lambda = 0 \tag{6.12m}$$

$$\text{with}\quad C = e_*T^{-1}e_*^T + FM^{-1}F^T \tag{6.12n}$$

From (6.12l)-(6.12m), we get

$$\begin{aligned}u &= A^{-1}(\ell - B^T\lambda)\\ S\lambda &= f\\ \text{with}\quad S &= C - BA^{-1}B^T\\ f &= -BA^{-1}\ell\end{aligned}$$

The following code implements this technique:

File 6.8: dirichlet_hho.cc

```
#include "rheolef.h"
using namespace rheolef;
using namespace std;
#include "sinusprod_dirichlet.h"
#include "diffusion_isotropic.h"
int main(int argc, char**argv) {
  environment rheolef (argc, argv);
  geo omega (argv[1]);
  string Pkd = (argc > 2) ? argv[2] : "P1d",
         Pld = (argc > 3) ? argv[3] :  Pkd;
  space  Xh (omega, Pld),
         Mh (omega["sides"], Pkd);
  Mh.block("boundary");
  size_t k = Mh.degree(), l = Xh.degree(), dim = omega.dimension();
  Float beta  = (argc > 4) ? atof(argv[4]) : 10*(k+1)*(k+dim)/Float(dim);
  check_macro(l == k-1 || l == k || l == k+1, "invalid (k,l)");
  space Xhs(omega, "P"+to_string(k+1)+"d"),
        Zh (omega, "P0"),
        Mht(omega, "trace_n(RT"+to_string(max(k,l))+"d)");
  trial us(Xhs), u(Xh), zeta(Zh), deltat(Mht), lambda(Mh);
  test  ws(Xhs), w(Xh),   xi(Zh),   phit(Mht),     mu(Mh);
  auto lh = lazy_integrate (f(dim)*w);
  auto m  = lazy_integrate (u*w);
  auto as = lazy_integrate (dot(grad_h(us),A(dim)*grad_h(ws)));
  auto cs = lazy_integrate (pow(h_local(),2)*zeta*xi);
  auto mt = lazy_integrate (on_local_sides(deltat*phit));
  auto ct = lazy_integrate (on_local_sides(beta*pow(h_local(),-1)*deltat*phit));
  auto bs = lazy_integrate (us*xi);
  auto d  = lazy_integrate (u*xi);
  auto ds = lazy_integrate (us*w);
  auto dt = lazy_integrate (on_local_sides(u*phit));
  auto dst= lazy_integrate (on_local_sides(us*phit));
  auto ac = lazy_integrate (dot(grad_h(u),A(dim)*grad_h(ws))
                - on_local_sides(u*dot(A(dim)*grad_h(ws),normal())));
  auto et = lazy_integrate (on_local_sides(mu*deltat));
  auto es = lazy_integrate (on_local_sides(mu*dot(A(dim)*grad_h(us),normal())));
  auto inv_cs = inv(cs);
  auto inv_Ss = inv(as + trans(bs)*inv_cs*bs);
  auto inv_T  = inv(as*inv_Ss*as + trans(bs)*inv_cs*bs);
  auto R = as*inv_Ss*trans(bs)*inv_cs*d - ac;
  auto Ac  = trans(R)*inv_T*R;
  auto D = ct*inv(mt)*(dst - dt*inv(m)*ds);
  auto M0 = inv_Ss - inv_Ss*as*inv_T*as*inv_Ss;
  auto inv_M = inv(ct + D*M0*trans(D));
  auto E = trans(dt)*inv(mt)*ct
         + trans(ac)*inv_T*as*inv_Ss*trans(D)
         + trans(d)*inv_cs*bs*M0*trans(D);
  auto As = E*inv_M*trans(E);
  auto inv_A = inv(Ac + As);
  auto F = es*inv_T*as*inv_Ss*trans(D)
         - et*inv(mt)*ct;
  auto C = es*inv_T*trans(es) + F*inv_M*trans(F);
  auto B = F*inv_M*trans(E) - es*inv_T*R;
  form S = C - B*inv_A*trans(B);
  problem pS (S);
  field rhs = -B*(inv_A*lh);
  field lambda_h(Mh, 0);
  pS.solve (rhs, lambda_h);
  auto uh = inv_A*(lh - B.trans_mult(lambda_h));
  auto deltat_h = inv_M*(E.trans_mult(uh) + F.trans_mult(lambda_h));
  auto vs_h = inv_T*(-as*inv_Ss*D.trans_mult(deltat_h) + R*uh - es.trans_mult(lambda_h));
  field us_h = inv_Ss*(-as*vs_h - D.trans_mult(deltat_h) + trans(bs)*inv_cs*d*uh);
  dout << catchmark("us")     << us_h
       << catchmark("u")      << field(uh)
       << catchmark("lambda") << lambda_h;
}
```

File 6.9: dirichlet_homogeneous.h

```
struct f {
  Float operator() (const point& x) const { return 1; }
  f(size_t=0) {}
};
struct g {
  Float operator() (const point& x) const { return 0; }
  g(size_t=0) {}
};
```

As usual, the right-hand-side f and the Dirichlet boundary condition g has been chosen such that the exact solution is given by:

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

How to run the program

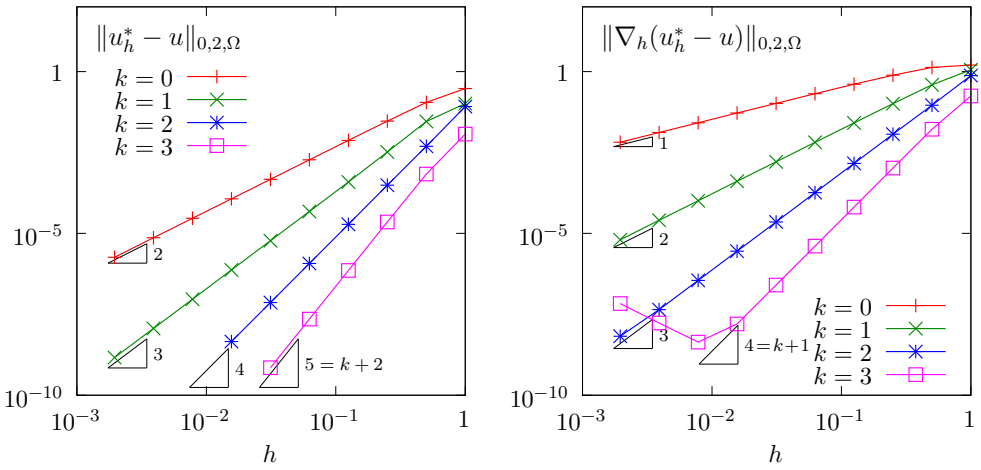


Figure 6.8: Hybrid hight-order (HHO) method: error vs h and k for the Poisson problem.

The compilation and run write:

```
make dirichlet_hho
mkgeo_grid -t 10 > square.geo
./dirichlet_hho square.geo P1d > square.field
field square.field -elevation
field square.field -elevation -mark u
field square.field -elevation -mark lambda
```

Observe the better behavior of the u_h^* approximate solution, obtained by reconstruction, when compared with the original one u. The error versus the mesh refinement h and the polynomial orer k is plotted on Fig. 6.8. These numerical experiments confirm the theoretical predictions (see e.g. Pietro and Droniou [2020], pages 65 and 68):

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

Fig. 6.9 plots the L^2 error for u_h^* versus the CPU time required to its computation for various mesh refinement h and polynomial order k. It compares it with the L^2 error for obtaining the P_{k+1}

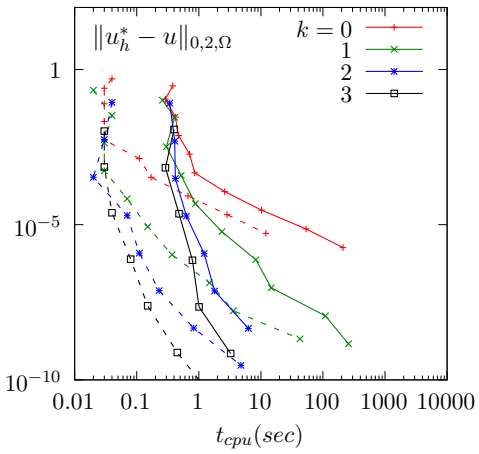


Figure 6.9: Comparison of hybrid hight-order (HHO) method (continuous lines) with classical Lagrange continuous approximation (dotted lines). Error vs CPU time for various h and k for the Poisson problem. The test bases on a bidimensional triangular mesh and a direct sequential solver (no parallel computation).

continuous Lagrange solution. While increasing the polynomial order k, the slope of convergence of the error versus h appears to be higher than its counterpart with the classical Lagrange continuous approximation (dotted lines). Thus, the HHO method appears to be asymptotically more efficient when $h \to 0$ i.e. large meshes and high CPU times. Nevertheless, the constant in front of this asymptotical HHO method is higher than its counterpart with the classical Lagrange continuous approximation, and thus, when the mesh is medium or small, the classical method is still faster. The constant of the HHO method is due to algebraic manipulations, such as assembly and matrix-matrix products at the element level: it could certainly be decreased by a more efficient implementation of the HHO method, and this perspective will be explorated in the future.

Appendix A

Technical appendices

A.1 How to write a variational formulation ?

The major key-point for using **Rheolef** is to put the problem in variational form. Then this variational form can be efficiently translated into `C++` language. This appendix is dedicated to readers who are not fluent with variational formulations and some related functional analysis tools.

A.1.1 The Green formula

Let us come back to the model problem presented in section 1.1.1, page 14, equations (1.1)-(1.2) and details how this problem is transformed into (1.3).

Let $H_0^1(\Omega)$ the space of functions whose gradient square has a finite sum over Ω and that vanishes on $\partial\Omega$:

$$H_0^1(\Omega) = \{v \in L^2(\Omega);\ \nabla v \in L^2(\Omega)^d \text{ and } v = 0 \text{ on } \partial\Omega\}$$

We start by multiplying (1.1) by an arbitrarily test-function $v \in H_0^1(\Omega)$ and then integrate over Ω :

$$-\int_\Omega \Delta u\, v \,\mathrm{d}x = \int_\Omega f\, v \,\mathrm{d}x, \quad \forall v \in H_0^1(\Omega)$$

The next step is to invoke an integration by part, the so-called Green formula:

$$\int_\Omega \Delta u\, v \,\mathrm{d}x + \int_\Omega \nabla u.\nabla v \,\mathrm{d}x = \int_{\partial\Omega} \frac{\partial u}{\partial n}\, v \,\mathrm{d}s, \quad \forall u, v \in H^1(\Omega)$$

Since our test-function v vanishes on the boundary, the integral over $\partial\Omega$ is zero and the problem becomes:

$$\int_\Omega \nabla u.\nabla v \,\mathrm{d}x = \int_\Omega f\, v \,\mathrm{d}x, \quad \forall v \in H_0^1(\Omega)$$

This is exactly the variational formulation (1.3), page 14.

A.1.2 The vectorial Green formula

In this section, we come back to the linear elasticity problem presented in section 2.1.1, page 43, equations (2.1)-(2.2) and details how this problem is transformed into (2.3).

Let Γ_d (resp. Γ_n) denotes the parts of the boundary $\partial\Omega$ related to the homogeneous Dirichlet boundary condition $\mathbf{u} = 0$ (resp. the homogeneous Neumann boundary condition $\sigma(\mathbf{u})\,\mathbf{n} = 0$). We suppose that $\partial\Omega = \overline{\Gamma}_d \cap \overline{\Gamma}_n$. Let us introduce the following functional space:

$$\mathbf{V} = \{\mathbf{v} \in H^1(\Omega)^d;\ \mathbf{v} = 0 \text{ on } \Gamma_d\}$$

Then, multiplying the first equation of (2.2) by an arbitrarily test-function $\mathbf{v} \in \mathbf{V}$ and then integrate over Ω :

$$-\int_\Omega \mathbf{div}(\sigma(\mathbf{u})).\mathbf{v}\,\mathrm{d}x = \int_\Omega \mathbf{f}.\mathbf{v}\,\mathrm{d}x, \quad \forall \mathbf{v} \in \mathbf{V}$$

The next step is to invoke an integration by part:

$$\int_\Omega \mathbf{div}\,\tau.\mathbf{v}\,\mathrm{d}x + \int_\Omega \tau : D(\mathbf{v})\,\mathrm{d}x = \int_{\partial\Omega} \tau : (\mathbf{v}\otimes\mathbf{n})\,\mathrm{d}s, \quad \forall \tau \in L^2(\Omega)^{d\times d},\ \forall \mathbf{v} \in \mathbf{V}$$

Recall that $\mathbf{div}\,\tau$ denotes $\left(\sum_{j=0}^{d-1} \partial_j \tau_{i,j}\right)_{0\leqslant i<d}$, i.e. the vector whose component are the divergence of each row of τ. Also, $\sigma : \tau$ denote the double contracted product $\sum_{i,j=0}^{d-1} \sigma_{i,j}\tau_{i,j}$ for any tensors σ and τ, and that $\mathbf{u}\otimes\mathbf{v}$ dotes the $\tau_{i,j} = u_i\, v_j$ tensor, vectors $\mathbf{u}$ and $\mathbf{v}$. Remark that $\tau : (\mathbf{u}\otimes\mathbf{v}) = (\tau\,\mathbf{v}).\mathbf{u} = \sum_{i,j=0}^{d-1} \tau_{i,j}\, u_i\, v_j$. Choosing $\tau = \sigma(\mathbf{u})$ in the previous equation leads to:

$$\int_\Omega \sigma(\mathbf{u}) : D(\mathbf{v})\,\mathrm{d}x = \int_{\partial\Omega} (\sigma(\mathbf{u})\,\mathbf{n}).\mathbf{v}\,\mathrm{d}s + \int_\Omega \mathbf{f}.\mathbf{v}\,\mathrm{d}x, \quad \forall \mathbf{v} \in \mathbf{V}$$

Since our test-function v vanishes on Γ_d and the solution satisfies the homogeneous Neumann boundary condition $\sigma(\mathbf{u})\,\mathbf{n} = 0$ on Γ_n, the integral over $\partial\Omega$ is zero and the problem becomes:

$$\int_\Omega \sigma(\mathbf{u}) : D(\mathbf{v})\,\mathrm{d}x = \int_\Omega \mathbf{f}.\mathbf{v}\,\mathrm{d}x, \quad \forall \mathbf{v} \in \mathbf{V}$$

From the definition of $\sigma(\mathbf{u})$ in (2.1) page 43 we have:

$$\begin{aligned} \sigma(\mathbf{u}) : D(\mathbf{v}) &= \lambda\,\mathrm{div}(\mathbf{u})\,(I : D(\mathbf{v})) + 2\mu D(\mathbf{u}) : D(\mathbf{v}) \\ &= \lambda\,\mathrm{div}(\mathbf{u})\,\mathrm{div}(\mathbf{v}) + 2\mu D(\mathbf{u}) : D(\mathbf{v}) \end{aligned}$$

and the previous relation becomes:

$$\int_\Omega \lambda \mathrm{div}(\mathbf{u})\,\mathrm{div}(\mathbf{v})\,\mathrm{d}x + \int_\Omega 2\mu D(\mathbf{u}) : D(\mathbf{v})\,\mathrm{d}x = \int_\Omega \mathbf{f}.\mathbf{v}\,\mathrm{d}x, \quad \forall \mathbf{v} \in \mathbf{V}$$

This is exactly the variational formulation (2.3), page 43.

A.1.3 The Green formula on a surface

Let Γ a closed and orientable surface of $\mathbb{R}^d$, $d = 2, 3$ and $\mathbf{n}$ its unit normal. From [Laadhari et al., 2010], appendix C we have the following integration by part:

$$\int_\Gamma \mathrm{div}_s\mathbf{v}\,\xi\,\mathrm{d}s + \int_\Gamma \mathbf{v}.\nabla_s\xi\,\mathrm{d}s = \int_\Gamma \mathbf{v}.\mathbf{n}\,\xi\,\mathrm{div}\,\mathbf{n}\,\mathrm{d}s$$

for all $\xi \in H^1(\Gamma)$ and $\mathbf{v} \in H^1(\Gamma)^d$. Note that $\mathrm{div}\,\mathbf{n}$ represent the surface curvature. Next, we choose $\mathbf{v} = \nabla_s\varphi$, for any $\varphi \in H^2(\Gamma)$. Remaking that $\mathbf{v}.\mathbf{n} = 0$ and that $\mathrm{div}_s\mathbf{v} = \Delta_s\varphi$. Then:

$$\int_\Gamma \Delta_s\,\xi\,\mathrm{d}s + \int_\Gamma \nabla_s\varphi.\nabla_s\xi\,\mathrm{d}s = 0$$

This formula is the starting point for all variational formulations of problems defined on a surface (see chapter 3.1).

A.2 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 Hecht [2006], and the `gmsh` mesh generator, suitable when $d = 1, 2$ and 3, and written by Geuzaine and Remacle [2009].

A.2.1 Bidimensional mesh with `bamg`

We first create a '`square.bamgcad`' file:

```
MeshVersionFormatted
  0
Dimension
  2
Vertices
  4
  0  0     1
  1  0     2
  1  1     3
  0  1     4
Edges
  4
  1  2     101
  2  3     102
  3  4     103
  4  1     104
hVertices
  0.1 0.1 0.1 0.1
```

This is an uniform mesh with element size $h = 0.1$. We refer to the `bamg` documentation Hecht [2006] for the complete file format description. Next, enter the mesh generator commands:

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

Then, create the file '`square.dmn`' that associate names to the four boundary domains of the mesh. Here, there is four boundary domains:

```
EdgeDomainNames
  4
  bottom
  right
  top
  left
```

and enter the translation command:

```
bamg2geo square.bamg square.dmn > square.geo
```

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

```
geo square
```

This presents the mesh it in a graphical form, usually with `paraview`. You can switch to the `gnuplot` render:

```
geo square -gnuplot
```

A finer mesh could be generated by:

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

A.2.2 Unidimensional mesh with `gmsh`

The simplest unidimensional mesh is a line:

```
h_local = 0.1;
Point(1) = {0, 0, 0, h_local};
Point(2) = {1, 0, 0, h_local};
Line(3) = {1,2};
Physical Point("left")  = {1};
Physical Point("right") = {2};
Physical Point("boundary") = {1,2};
Physical Line("interior") = {3};
```

The mesh generation command writes:

```
gmsh -1 line.mshcad -format msh2 -o line.msh
```

Then, the conversion to '`.geo`' format and the visualization:

```
msh2geo line.msh > line.geo
geo line
```

A.2.3 Bidimensional mesh with `gmsh`

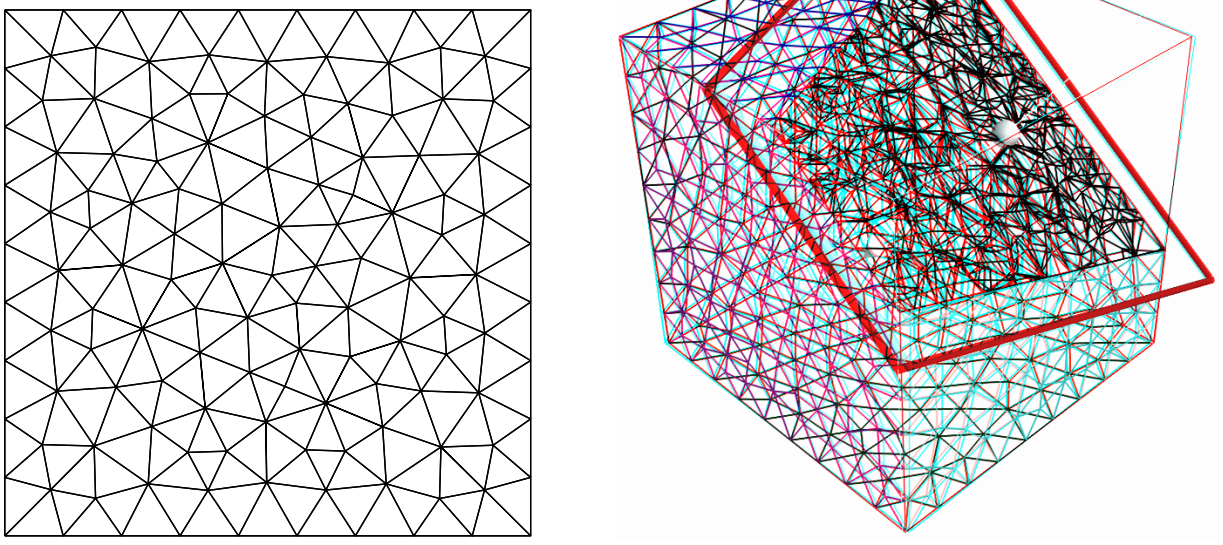


Figure A.1: Visualization of the `gmsh` meshes '`square.geo`' and '`cube.geo`'.

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

```
n = 10.0;
hloc = 1.0/n;
Point(1) = {0, 0, 0, hloc};
Point(2) = {1, 0, 0, hloc};
```

```
Point(3) = {1, 1, 0, hloc};
Point(4) = {0, 1, 0, hloc};
Line(1) = {1,2};
Line(2) = {2,3};
Line(3) = {3,4};
Line(4) = {4,1};
Line Loop(5) = {1,2,3,4};
Plane Surface(6) = {5} ;
Physical Point("left_bottom")  = {1};
Physical Point("right_bottom") = {2};
Physical Point("right_top")    = {3};
Physical Point("left_top")     = {4};
Physical Line("boundary") = {1,2,3,4};
Physical Line("bottom") = {1};
Physical Line("right")  = {2};
Physical Line("top")    = {3};
Physical Line("left")   = {4};
Physical Surface("interior") = {6};
```

This is an uniform mesh with element size $h = 0.1$. We refer to the `gmsh` documentation Geuzaine and Remacle [2009] for the complete file format description. Next, enter the mesh generator commands:

```
gmsh -2 square.mshcad -format msh2 -o square.msh
```

Then, enter the translation command:

```
msh2geo square.msh > square.geo
```

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

```
geo square
```

Remark that the domain names, defined in the .mshcad file, are included in the gmsh .msh input file and are propagated in the .geo by the format conversion.

A.2.4 Tridimensional mesh with `gmsh`

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

```
Mesh.Algorithm   = 7; // bamg
Mesh.Algorithm3D = 7; // mmg3d
a = 0; c = 0; f = 0;
b = 1; d = 1; g = 1;
n = 10;
hloc = 1.0/n;
Point(1) = {a, c, f, hloc};
Point(2) = {b, c, f, hloc};
Point(3) = {b, d, f, hloc};
Point(4) = {a, d, f, hloc};
Point(5) = {a, c, g, hloc};
Point(6) = {b, c, g, hloc};
Point(7) = {b, d, g, hloc};
Point(8) = {a, d, g, hloc};
Line(1) = {1,2};
```

```
Line(2) = {2,3};
Line(3) = {3,4};
Line(4) = {4,1};
Line(5) = {5,6};
Line(6) = {6,7};
Line(7) = {7,8};
Line(8) = {8,5};
Line(9)  = {1,5};
Line(10) = {2,6};
Line(11) = {3,7};
Line(12) = {4,8};
Line Loop(21) = {-1,-4,-3,-2};
Plane Surface(31) = {21} ;
Line Loop(22) = {5,6,7,8};
Plane Surface(32) = {22} ;
Line Loop(23) = {1,10,-5,-9};
Plane Surface(33) = {23} ;
Line Loop(24) = {12,-7,-11,3};
Plane Surface(34) = {24} ;
Line Loop(25) = {2,11,-6,-10};
Plane Surface(35) = {25} ;
Line Loop(26) = {9,-8,-12,4};
Plane Surface(36) = {26} ;
Surface Loop(41) = {31,32,33,34,35,36};
Volume(51) = {41};
Physical Surface("bottom") = {31};
Physical Surface("top") = {32};
Physical Surface("left") = {33};
Physical Surface("front") = {35};
Physical Surface("right") = {34};
Physical Surface("back") = {36};
Physical Volume("internal") = {51};
```

Next, enter the mesh generator commands:

```
gmsh -3 cube.mshcad -format msh2 -o cube.msh
```

Then, enter the translation command:

```
msh2geo cube.msh > cube.geo
```

This command creates a '`cube.geo`' file. Look at the mesh via the command:

```
geo cube
geo cube.geo -cut
```

The second command allows one to see inside the mesh.

Appendix B

GNU Free Documentation License

Version 1.1, March 2000

Copyright © 2000 Free Software Foundation, Inc.
59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

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

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

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

Applicability and Definitions

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

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

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

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

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

Verbatim Copying

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

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

Copying in Quantity

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

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

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you

use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

Modifications

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
- State on the Title page the name of the publisher of the Modified Version, as the publisher.
- Preserve all the copyright notices of the Document.
- Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- Include an unaltered copy of this License.
- Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

- Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.
- Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties – for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

Combining Documents

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

Collections of Documents

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

Aggregation With Independent Works

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

Translation

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

Termination

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

Future Revisions of This License

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

*

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with the

> Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being LIST"; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Bibliography

E. M. Abdalass. *Résolution performante du problème de Stokes par mini-éléments, maillages auto-adaptatifs et méthodes multigrilles – applications*. PhD thesis, Thèse de l'école centrale de Lyon, 1987.

L. Abouorm. Méthodes mathématiques pour les écoulements sur des surfaces. Master's thesis, M2R Université J. Fourier, Grenoble, 2010. URL http://www-ljk.imag.fr/membres/Pierre.Saramito/Lara-Abouorm-m2r-2010.pdf.

P. Alart. Méthode de Newton généralisée en mécanique du contact. *Journal de Mathématiques Pures et Appliquées*, 76(1):83–108, 1997.

P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, and J. Koster. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM J. Matrix Anal. Appl.*, 23(1):15–41, 2001. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.40.4181&rep=rep1&type=pdf.

P. R Amestoy, A. Guermouche, J.-Y. L'Excellent, and S. Pralet. Hybrid scheduling for the parallel solution of linear systems. *Parallel Comput.*, 32(2):136–156, 2006. URL http://hal.inria.fr/inria-00070599/PDF/RR-5404.pdf.

D. N. Arnold and J. Qin. Quadratic velocity/linear pressure Stokes elements. *Adv. Comput. Meth. Partial Diff. Eqn.*, 7:28–34, 1992.

D. N. Arnold, F. Brezzi, and M. Fortin. A stable finite element for the Stokes equations. *Calcolo*, 21:337–344, 1984.

U. M. Ascher, S. J. Ruuth, and R. J. Spiteri. Implicit-explicit Runge-Kutta methods for time-dependent partial differential equations. *Appl. Numer. Math.*, 25(2):151–167, 1997.

C. Ashcraft and J. W. H. Liu. Robust ordering of sparse matrices using multisection. *SIAM J. Matrix Anal. Appl.*, 19(3):816–832, 1998.

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.

G. K. Batchelor. *An introduction to fluid dynamics*. Cambridge university press, UK, sixth edition, 1967.

R. Bird, R. C. Armstrong, and O. Hassager. *Dynamics of polymeric liquids. Volume 1. Fluid mechanics*. Wiley, New-York, second edition, 1987.

H. Borouchaki, P. L. George, F. Hecht, P. Laug, B. Mohammadi, and E. Saltel. Mailleur bidimensionnel de Delaunay gouverné par une carte de métriques. Partie II: applications. Technical Report RR-2760, INRIA, 1995.

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.

S. C. Brenner and L. R. Scott. *The mathematical theory of finite element methods*. Springer, second edition, 2002.

H. Brezis. *Analyse fonctionnelle. Théorie et application*. Masson, Paris, 1983.

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.

A. Caglar and A. Liakos. Weak imposition of boundary conditions for the Navier-Stokes equations by a penalty method. *Int. J. Numer. Meth. Fluids*, 61(4):411–431, 2009.

M. P. Calvo, J. de Frutos, and J. Novo. Linearly implicit Runge-Kutta methods for advection-reaction-diffusion equations. *Appl. Numer. Math.*, 37(4):535–549, 2001.

G. F. Carey and B. Jianng. Least-squares finite elements for first-order hyperbolic systems. *Int. J. Numer. Meth. Eng.*, 26(1):81–93, 1988.

P. Castillo. Performance of discontinuous Galerkin methods for elliptic PDEs. *SIAM J. Sci. Comput.*, 24(2):524–547, 2002.

M. J. Castro-Diaz, F. Hecht, B. Mohammadi, and O. Pironneau. Anisotropic unstructured mesh adaption for flow simulations. *Int. J. Numer. Meth. Fluids*, 25(4):475–491, 1997.

J. C. Chrispell, V. J. Ervin, and E. W. Jenkins. A fractional step θ-method approximation of time-dependent viscoelastic fluid flow. *J. Comput. Appl. Math.*, 232(2):159–175, 2009.

B. Cockburn. *An introduction to the discontinuous Galerkin method for convection-dominated problems*, chapter 2, pages 151–268. Springer, 1998.

B. Cockburn, S. Hou, and C.-W. Shu. The Runge-Kutta local projection discontinuous Galerkin finite element method for conservation laws. IV. the multidimensional case. *Math. Comput.*, 54 (190):545–581, 1990.

B. Cockburn, G. Kanschat, D. Schötzau, and C. Schwab. Local discontinuous Galerkin methods for the Stokes system. *SIAM J. Numer. Anal.*, 40(1):319–343, 2002.

B. Cockburn, G. Kanschat, and D. Schötzau. A locally conservative LDG method for the incompressible Navier-Stokes equations. *Math. Comput.*, 74(251):1067–1095, 2005.

B. Cockburn, B. Dong, and J. Guzmán. A superconvergent LDG-hybridizable Galerkin method for second-order elliptic problems. *Math. Comput.*, 77(264):1887–1916, 2008.

B. Cockburn, J. Guzmán, and H. Wang. Superconvergent discontinuous Galerkin methods for second-order elliptic problems. *Math. Comput.*, 78(265):1–24, 2009.

B. Cockburn, B. Dong, J. Guzmán, and J. Qian. Optimal convergence of the original DG method on special meshes for variable transport velocity. *SIAM J. Numer. Anal.*, 48(1):133–146, 2010a.

B. Cockburn, J. Gopalakrishnan, and F.-J. Sayas. A projection-based error analysis of HDG methods. *Math. Comput.*, 79(271):1351–1367, 2010b.

B. Cockburn, D. A. di Pietro, and A. Ern. Bridging the hybrid high-order and hybridizable discontinuous Galerkin methods. *ESAIM Math. Model. Numer. Anal.*, 50(3):635–650, 2016.

M. Crouzeix and J. Rappaz. *On numerical approximation in bifurcation theory*. Masson, Paris, 1990.

K. Deckelnick, G. Dziuk, C.M. Elliott, and C.-J. Heine. An h-narrow band finite element method for elliptic equations on implicit surfaces. *IMA Journal of Numerical Analysis*, to appear:0, 2009.

D. A. di Pietro and A. Ern. Discrete functional analysis tools for discontinuous Galerkin methods with application to the incompressible Navier-Stokes equations. *Math. Comp.*, 79:1303–1330, 2010.

D. A. di Pietro and A. Ern. *Mathematical aspects of discontinuous Galerkin methods*. Springer, 2012.

D. A. di Pietro and A. Ern. Hybrid high-order methods for variable-diffusion problems on general meshes. *C. R. Math.*, 353(1):31–34, 2015.

D. A. di Pietro, S. Lo Forte, and N. Parolini. Mass preserving finite element implementations of the level set method. *Appl. Numer. Math.*, 56(9):1179–1195, 2006.

M. Dicko. Méthodes mathématiques pour les écoulements sur des surfaces. Master's thesis, M2P Université J. Fourier, Grenoble, 2011. URL http://www-ljk.imag.fr/membres/Pierre.Saramito/mahamar-dicko-m2r.pdf.

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

M. Dryja. On discontinuous Galerkin methods for elliptic problems with discontinuous coefficients. *Comput. Meth. Appl. Math.*, 3(1):76–85, 2003.

D. Enright, R. Fedkiw, J. Ferziger, and I. Mitchell. A hybrid particle level set method for improved interface capturing. *J. Comput. Phys.*, 183(1):83–116, 2002.

Y. Epshteyn and B. Rivière. Estimation of penalty parameters for symmetric interior penalty Galerkin methods. *J. Comput. Appl. Math.*, 206(2):843–872, 2007.

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.

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.

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.

A. George. Nested dissection of a regular finite element mesh. *SIAM J. Numer. Anal.*, 10:345–363, 1973.

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.

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.

V. Girault and P. A. Raviart. *Finite element methods for the Navier-Stokes equations. Theory and algorithms*. Springer, 1986.

S. Gottlieb and C.-W. Shu. Total variation diminishing Runge-Kutta schemes. *Math. Comput.*, 67(221):73–85, 1998.

S. Gottlieb, Chi-W. Shu, and E. Tadmor. Strong stability-preserving high-order time discretization methods. *SIAM review*, 43(1):89–112, 2001.

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.

A. Harten, B. Engquist, S. Osher, and S. R. Chakravarthy. Uniformly high order accurate essentially non-oscillatory schemes, III. *J. Comput. Phys.*, 71(2):231–303, 1987.

F. Hecht. *BAMG: bidimensional anisotropic mesh generator*, 2006. https://www.ljll.math.upmc.fr/hecht/ftp/bamg.

J. S. Hesthaven and T. Warburton. *Nodal discontinuous Galerkin methods. Algorithms, analysis and applications*. Springer, 2008.

A. J. Hoffman, M. S. Martin, and D. J. Rose. Complexity bounds for regular finite difference and finite element grids. *SIAM J. Numer. Anal.*, 10(2):364–369, 1973.

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.

J. S. Howell. Computation of viscoelastic fluid flows using continuation methods. *J. Comput. Appl. Math.*, 225(1):187–201, 2009.

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

C. Johnson and J. Pitkäranta. An analysis of the discontinuous Galerkin method for a scalar hyperbolic equation. *Math. Comp.*, 46(173):1–26, 1986.

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.

A. Laadhari, C. Misbah, and P. Saramito. On the equilibrium equation for a generalized biological membrane energy by using a shape optimization approach. *Phys. D*, 239:1568–1572, 2010.

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.

R. J. LeVeque. High-resolution conservative algorithms for advection in incompressible flow. *SIAM J. Numer. Anal.*, 33(2):627–665, 1996.

A. Liakos. Discretization of the Navier-Stokes equations with slip boundary condition. *Numer. Meth. Part. Diff. Eqn.*, 17(1):26–42, 2001.

M. Lukáčová-Medvidová, H. Mizerová, Š. Nečasová, and M. Renardy. Global existence result for the generalized Peterlin viscoelastic model. *SIAM J. Math. Anal.*, 49(4):2950–2964, 2017.

J. Málek, V. Pruša, T. Skřivan, and E. Süli. Thermodynamics of viscoelastic rate-type fluids with stress diffusion. *Phys. Fluids*, 30(2):023101, 2018.

E. Marchandise, J.-F. Remacle, and N. Chevaugeon. A quadrature-free discontinuous Galerkin method for the level set equations. *J. Comput. Phys.*, 212:338–357, 2006.

S. Melchior, V. Legat, P. Van Dooren, and A. J. Wathen. Analysis of preconditioned iterative solvers for incompressible flow problems. *Int. J. Numer. Meth. Fluids*, 68(3):269–286, 2012.

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.

P. P. Mosolov and V. P. Miasnikov. Variational methods in the theory of the fluidity of a viscous-plastic medium. *J. Appl. Math. Mech.*, 29(3):545–577, 1965.

P. P. Mosolov and V. P. Miasnikov. On stagnant flow regions of a viscous-plastic medium in pipes. *J. Appl. Math. Mech.*, 30(4):841–853, 1966.

P. P. Mosolov and V. P. Miasnikov. On qualitative singularities of the flow of a viscoplastic medium in pipes. *J. Appl. Math. Mech.*, 31(3):609–613, 1967.

D. R. Musser and A. Saini. *C++ STL tutorial and reference guide*. Addison Wesley, Reading, 1996a.

D. R. Musser and A. Saini. *STL tutorial and reference guide*. Addison-Wesley, 1996b.

N. C. Nguyen, J. Peraire, and B. Cockburn. Hybridizable discontinuous Galerkin methods. In *Spectral and high order methods for partial differential equations*, pages 63–84. Springer, 2011.

J. G. Oldroyd. On the formulation of rheological equations of states. *Proc. R. Soc. Lond. A*, 200: 523–541, 1950.

M. A. Olshanskii and A. Reusken. A finite element method for surface PDEs: matrix properties. *Numer. Math.*, 114:491–520, 2010. URL http://www.mathcs.emory.edu/~molshan/ftp/pub/interfaceLA.pdf.

M. A. Olshanskii, A. Reusken, and J. Grande. A finite element method for elliptic equations on surfaces. *SIAM J. Num. Anal.*, 47(5):3339–3358, 2009. URL http://www.mathcs.emory.edu/~molshan/ftp/pub/PaperInterface.pdf.

M. A. Olshanskii, A. Reusken, and X. Xu. On surface meshes induced by level set functions. *Computing and visualization in science*, 15(2):53–60, 2012.

S. Osher and J. A. Sethian. Front propaging with curvature-dependent speed: agorithms based on Hamilton-Jacobi formulations. *J. Comput. Phys.*, 79(12), 1988. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.46.1266&rep=rep1&type=pdf.

M. L. Ould Salihi. *Couplage de méthodes numériques en simulation directe d'écoulements incompressibles*. PhD thesis, Université J. Fourier, Grenoble, 1998.

O. Ozenda, P. Saramito, and G. Chambon. A new rate-independent tensorial model for suspensions of non-colloidal rigid particles in newtonian fluids. *J. Rheol.*, 62:889–903, 2018.

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

T.-W. Pan, J. Hao, and R. Glowinski. On the simulation of a time-dependent cavity flow of an Oldroyd-B fluid. *Int. J. Numer. Meth. Fluids*, 60(7):791–808, 2009.

J.-C. Paumier. *Bifurcation et méthodes numériques. Applications aux problèmes elliptiques semi-linéaires*. Masson, Paris, 1997.

F. Pellegrini. *PT-Scotch and libscotch 5.1 user's guide*. Université de Bordeaux and INRIA, France, 2010. URL http://www.labri.fr/~pelegrin/scotch.

T. E. Peterson. A note on the convergence of the discontinuous Galerkin method for a scalar hyperbolic equation. *SIAM J. Numer. Anal.*, 28(1):133–140, 1991.

D. A. Di Pietro and J. Droniou. *The hybrid high-order method for polytopal meshes: design, analysis, and applications*. Springer, 2020.

O. Pironneau. *Méthode des éléments finis pour les fluides*. Masson, Paris, 1988.

O. Pironneau and M. Tabata. Stability and convergence of a galerkin-characteristics finite element scheme of lumped mass type. *Int. J. Numer. Meth. Fluids*, 64:1240–1253, 2010.

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.

P.-A. Raviart and J.-M. Thomas. A mixed finite element method for 2-nd order elliptic problems. In *Mathematical aspects of finite element methods*, pages 292–315. Springer, 1977.

G. R. Richter. An optimal-order error estimate for the discontinuous galerkin method. *Math. Comput.*, 50(181):75–88, 1988.

J. E. Roberts and J.-M. Thomas. Mixed and hybrid methods. In P. G. Ciarlet and J.-L. Lions, editors, *Handbook of numerical analysis. Volume 2. Finite element methods (part 1)*, chapter 4, pages 524–639. Elsevier, 1991.

N. Roquet and P. Saramito. An adaptive finite element method for Bingham fluid flows around a cylinder. *Comput. Meth. Appl. Mech. Engrg.*, 192(31-32):3317–3341, 2003. URL http://www-ljk.imag.fr/membres/Pierre.Saramito/cylindre.pdf.

N. Roquet and P. Saramito. Stick-slip transition capturing by using an adaptive finite element method. *M2AN*, 38(2):249–260, 2004. URL http://archive.numdam.org/article/M2AN_2004__38_2_249_0.djvu.

N. Roquet and P. Saramito. An adaptive finite element method for viscoplastic flows in a square pipe with stick-slip at the wall. *J. Non-Newt. Fluid Mech.*, 155:101–115, 2008.

N. Roquet, R. Michel, and P. Saramito. Errors estimate for a viscoplastic fluid by using Pk finite elements and adaptive meshes. *C. R. Acad. Sci. Paris, ser. I*, 331(7):563–568, 2000. URL http://www-ljk.imag.fr/membres/Pierre.Saramito/cr2000.pdf.

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

P. Saramito. *Simulation numérique d'écoulements de fluides viscoélastiques par éléments finis incompressibles et une méthode de directions alternées; applications*. PhD thesis, Institut National Polytechnique de Grenoble, 1990. URL http://www-ljk.imag.fr/membres/Pierre.Saramito/these.pdf.

P. Saramito. Numerical simulation of viscoelastic fluid flows using incompressible finite element method and a θ-method. *Math. Model. Numer. Anal.*, 28(1):1–35, 1994. URL http://archive.numdam.org/article/M2AN_1994__28_1_1_0.pdf.

P. Saramito. Efficient simulation of nonlinear viscoelastic fluid flows. *J. Non Newt. Fluid Mech.*, 60:199–223, 1995. URL http://www-ljk.imag.fr/membres/Pierre.Saramito/jnnfm2.pdf.

P. Saramito. Operator splitting in viscoelasticity. *Élasticité, Viscoélasticité et Contrôle Optimal, Lyon, décembre 1995, ESAIM: Proceedings*, 2:275–281, 1997. URL http://www.esaim-proc.org/articles/proc/pdf/1997/01/saramito.pdf.

P. Saramito. **Rheolef** home page. https://www-ljk.imag.fr/membres/Pierre.Saramito/rheolef, 2012a.

P. Saramito. *Language C++ et calcul scientifique*. College Publications, London, 2013a.

P. Saramito. *Méthodes numériques en fluides complexes : théorie et algorithmes*. CNRS-CCSD, 2013b. http://cel.archives-ouvertes.fr/cel-00673816.

P. Saramito. On a modified non-singular log-conformation formulation for Johnson-Segalman viscoelastic fluids. *J. Non-Newt. Fluid Mech.*, 211:16–30, 2014.

P. Saramito. A damped Newton algorithm for computing viscoplastic fluid flows. *J. Non-Newt. Fluid Mech.*, 238:6–15, 2016a.

P. Saramito. *Complex fluids: modelling and algorithms*. Springer, 2016b.

P. Saramito and N. Roquet. An adaptive finite element method for viscoplastic fluid flows in pipes. *Comput. Meth. Appl. Mech. Engrg.*, 190(40-41):5391–5412, 2001. URL http://www-ljk.imag.fr/membres/Pierre.Saramito/publi-poiseuille.pdf.

P. Saramito and A. Wachs. Progress in numerical simulation of yield stress fluid flows. *J. Rheol.*, 56(3):211–230, 2017.

Pierre Saramito. Are curved and high order gmsh meshes really high order ?, 2012b. http://www.geuz.org/pipermail/gmsh/2012/006967.html.

L. R. Scott and M. Vogelius. Norm estimates for a maximal right inverse of the divergence operator in spaces of piecewise polynomials. *M2AN*, 19(1):111–143, 1985.

N. D. Scurtu. *Stability analysis and numerical simulation of non-Newtonian fluids of Oldroyd kind*. PhD thesis, U. Nürnberg, Deutschland, 2005.

J. Sethian. *Level set methods and fast marching methods*. Cambridge University Press, UK, 1999.

K. Shahbazi. An explicit expression for the penalty parameter of the interior penalty method. *J. Comput. Phys.*, 205(2):401–407, 2005.

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.

C.-W. Shu. TVB boundary treatment for numerical solutions of conservation laws. *Math. Comput.*, 49(179):123–134, 1987.

C.-W. Shu and S. Osher. Efficient implementation of essentially non-oscillatory shock-capturing schemes. *J. Comput. Phys.*, 77(2):439–471, 1988.

P. Singh and L. G. Leal. Finite-element simulation of the start-up problem for a viscoelastic fluid in an eccentric rotating cylinder geometry using a third-order upwind scheme. *Theor. Comput. Fluid Dyn.*, 5(2-3):107–137, 1993.

B. Stroustrup. C++ programming styles and libraries. *InformIt.com*, 0:0, 2002.

E. Süli and D. F. Mayers. *An introduction to numerical analysis*. Cambridge University Press, UK, 2003.

M. Ta, F. Pigeonneau, and P. Saramito. An implicit high order discontinuous Galerkin level set method for two-phase flow problems. In *9th international conference on multiphase flow (ICMF)*, 2016.

G. I. Taylor. On the decay of vortices in a viscous fluid. *Philos. Mag.*, 46:671–674, 1923.

M.-G. Vallet. Génération de maillages anisotropes adaptés, application à la capture de couches limites. Technical Report RR-1360, INRIA, 1990.

R. Verfürth. Finite element approximation of steady Navier-Stokes equations with mixed boundary conditions. *ESAIM Math. Model. and Numer. Anal.*, 19(3):461–475, 1985.

R. Verfürth. Finite element approximation of incompressible Navier-Stokes equations with slip boundary condition. *Numer. Math.*, 50:697–721, 1987.

R. Verfürth. Finite element approximation of incompressible Navier-Stokes equations with slip boundary conditions II. *Numer. Math.*, 59:615–636, 1991.

H. Wang, C.-W. Shu, and Q. Zhang. Stability and error estimates of local discontinuous Galerkin methods with implicit-explicit time-marching for advection-diffusion problems. *SIAM J. Numer. Anal.*, 53(1):206–227, 2015a.

H. Wang, C.-W. Shu, and Q. Zhang. Stability analysis and error estimates of local discontinuous Galerkin methods with implicit–explicit time-marching for nonlinear convection–diffusion problems. *Appl. Math. Comput.*, 2015b.

Wikipedia. The Stokes stream function. *Wikipedia*, 2012. http://en.wikipedia.org/wiki/Stokes_stream_function.

N. Wirth. *Algorithm + data structure = programs*. Prentice Hall, NJ, USA, 1985.

S. T. Zalesak. Fully multidimensional flux-corrected transport algorithms for fluids. *J. Comput. Phys.*, 31(3):335–362, 1979.

S. Zhang. A new family of stable mixed finite elements for the 3D Stokes equations. *Math. Comput*, 74(250):543–554, 2005.

X. Zhong and C.-W. Shu. A simple weighted essentially nonoscillatory limiter for Runge-Kutta discontinuous galerkin methods. *J. Comput. Phys.*, 232(1):397–415, 2013.

List of example files

Makefile, 12
burgers.icc, 157
burgers_dg.cc, 159
burgers_diffusion_dg.cc, 173
burgers_diffusion_exact.h, 172
burgers_diffusion_operators.icc, 174
burgers_flux_godunov.icc, 157
cavity.h, 50
cavity_dg.h, 191
combustion.h, 127
combustion1.icc, 127
combustion2.icc, 128
combustion_continuation.cc, 133
combustion_error.cc, 131
combustion_exact.icc, 131
combustion_keller.cc, 139
combustion_keller_post.cc, 140
combustion_newton.cc, 129
commute_rtd.cc, 247
contraction.h, 60
convect.cc, 75
convect_error.cc, 77
cosinusprod.h, 23
cosinusprod_error.cc, 23
cosinusprod_laplace.h, 21
cosinusrad.h, 24
cosinusrad_laplace.h, 23
d_projection_dx.h, 207
diffusion_transport_tensor_dg.cc, 242
dirichlet-nh.cc, 20
dirichlet.cc, 10
dirichlet.icc, 110
dirichlet_dg.cc, 164
dirichlet_hdg.cc, 251
dirichlet_hdg_average.cc, 254
dirichlet_hdg_average.icc, 254
dirichlet_hdg_post.cc, 256
dirichlet_hdg_post_rt.cc, 259
dirichlet_hho.cc, 270
dirichlet_homogeneous.h, 271
elasticity_criterion.icc, 47
elasticity_solve.icc, 47
elasticity_taylor_dg.cc, 177
embankment.cc, 40
embankment.icc, 41
embankment_adapt.cc, 46
eta.h, 110
harten.h, 157
harten_show.cc, 158
heat.cc, 73
helmholtz_band.cc, 102
helmholtz_band_iterative.cc, 100
helmholtz_s.cc, 90
incompressible-elasticity.cc, 57
inertia.h, 184
inertia_cks.icc, 187
inertia_upw.icc, 192
lambda2alpha.h, 131
lambda_c.cc, 130
lambda_c.h, 130
laplace_band.cc, 103
laplace_s.cc, 94
level_set_sphere.cc, 97
leveque_dg.cc, 151
mosolov_augmented_lagrangian.cc, 216
mosolov_augmented_lagrangian.h, 215
mosolov_augmented_lagrangian1.icc, 214
mosolov_augmented_lagrangian2.icc, 215
mosolov_error_yield_surface.cc, 222
mosolov_exact_circle.h, 220
mosolov_yield_surface.cc, 219
navier_stokes_cavity.cc, 82
navier_stokes_criterion.icc, 83
navier_stokes_dg.h, 189
navier_stokes_dg1.icc, 189
navier_stokes_dg2.icc, 191
navier_stokes_solve.icc, 81
navier_stokes_taylor_dg.cc, 184
navier_stokes_taylor_newton_dg.cc, 189
navier_stokes_upw_dg.h, 193
navier_stokes_upw_dg.icc, 193
neumann-laplace.cc, 32
neumann-nh.cc, 28
neumann_dg.cc, 166
nu.h, 118
oldroyd_contraction.cc, 235

oldroyd_contraction.h, 235
oldroyd_theta_scheme.h, 231
oldroyd_theta_scheme1.h, 232
oldroyd_theta_scheme2.h, 232
oldroyd_theta_scheme3.h, 233
p_laplacian.h, 116
p_laplacian1.icc, 117
p_laplacian2.icc, 117
p_laplacian_circle.h, 125
p_laplacian_damped_newton.cc, 122
p_laplacian_error.cc, 125
p_laplacian_fixed_point.cc, 108
p_laplacian_newton.cc, 116
phi.h, 201
poisson_robin.icc, 202
proj_band.cc, 101
projection.h, 200
reconstruction_hho.cc, 263
robin.cc, 30
rotating-hill.h, 77
sgn.icc, 193
sinusprod_helmholtz.h, 29
sphere.icc, 91
stokes_cavity.cc, 50
stokes_contraction_bubble.cc, 60
stokes_dirichlet_dg.icc, 180
stokes_obstacle_slip_regul.cc, 70
stokes_taylor_dg.cc, 180
streamf_cavity.cc, 55
streamf_contraction.cc, 64
streamf_obstacle_slip_move.cc, 70
stress.cc, 43
taylor.h, 178
torus.icc, 94
transmission.cc, 35
transmission_dg.cc, 168
transport_dg.cc, 145
transport_tensor_dg.cc, 227
vector_projection.h, 214
vortex_position.cc, 86
vorticity.cc, 53
yield_slip.h, 208
yield_slip1.icc, 208
yield_slip2.icc, 209
yield_slip_augmented_lagrangian.cc, 202
yield_slip_augmented_lagrangian.icc, 202
yield_slip_damped_newton.cc, 207
zalesak_dg.cc, 148
bdf.icc, 147
cavity.h, 83
commute_rtd_error.cc, 247
contraction.h, 235
contraction.mshcad, 62, 66
convect.cc, 82
cosinus_vector.h, 247
cosinusprod_error_dg.cc, 166
cosinusrad_error.cc, 25
cube.mshcad, 279
diffusion_isotropic.h, 264
diffusion_transport_tensor_error_dg.cc, 243
diffusion_transport_tensor_exact.icc, 242
dirichlet_nh_ball.cc, 25
elasticity_taylor_error_dg.cc, 178
harten0.h, 157
helmholtz_s_error.cc, 93
level_set_torus.cc, 98
leveque.h, 151
line.mshcad, 49, 278
mkview_mosolov, 219
mosolov_error.cc, 221
mosolov_residue.cc, 217
navier_stokes_cavity_newton_dg.cc, 191
navier_stokes_cavity_newton_upw_dg.cc, 193
navier_stokes_taylor_error_dg.cc, 185
neumann-nh.cc, 90
runge_kutta_semiimplicit.icc, 173
runge_kutta_ssp.icc, 154
sinusprod.h, 264
sinusprod_dirichlet.h, 251
sinusprod_error_hdg.cc, 251
sinusprod_error_hdg_average.cc, 254
sinusprod_error_hdg_post_rt.cc, 259
sinusprod_error_hho_reconstruction.cc, 264
sinusprod_grad.h, 251
square.bamgcad, 48, 84, 277
square.dmn, 48, 84, 277
square.mshcad, 278
stokes_contraction.cc, 64
stokes_taylor_error_dg.cc, 181
streamf_cavity.cc, 85
streamf_contraction.cc, 66, 237
stress.cc, 68
taylor.h, 181, 185
taylor_exact.h, 178
torus.mshcad, 95
transport_tensor_error_dg.cc, 228
transport_tensor_exact.icc, 228
yield_slip_circle.h, 211
yield_slip_error.cc, 211
yield_slip_residue.cc, 204
zalesak.h, 148

List of commands

bamg2geo, 277
bamg, 48, 84, 277
    -splitpbedge, 111
branch, 77
    -branch, 134, 140, 237
    -bw, 150, 152
    -extract, 134, 140, 237
    -gnuplot, 175
    -iso, 150, 152
    -toc, 134, 140, 237
    -umax, 175
    -umin, 175
    -volume, 75, 80
    -vtk, 150
convect, 77
field, 13
    -, 15
    -bw, 13, 64, 67, 85, 101
    -comp, 43, 46, 67, 68, 85
    -cut, 67, 68, 85, 110, 204
    -domain, 204
    -elevation, 13, 45, 74, 101, 110, 204
    -fill, 43
    -gnuplot, 13, 14, 67, 68, 85, 110, 204
    -gray, 13
    -mark, 52
    -max, 65, 86, 134
    -min, 86
    -n-iso, 64, 219
    -n-iso-negative, 64, 67, 85, 219, 237
    -noclean, 56
    -noexecute, 56
    -nofill, 13, 42, 48
    -normal, 67, 68, 85, 110, 204
    -origin, 67, 68, 85, 110, 204
    -proj, 45, 219
    -scale, 58
    -stereo, 13, 15, 43, 45, 46, 91, 101
    -velocity, 52, 85
    -volume, 16, 22
geo, 13
    -cut, 15
    -fill, 15
    -full, 15
    -gnuplot, 96
    -paraview, 277
    -shrink, 15
    -stereo, 15, 91
    -subdivide, 92
gmsh, 49, 63, 66, 91, 95, 111, 278
gnuplot, 13-15, 36, 43, 74, 77, 277
gzip, 48
library
    boost, 11
    CGAL, computational geometry, 75
    MPI, message passing interface, 11
    mumps, linear system direct solver, 17
    scotch, mesh partition library, 17
    STL, standard template library, 77
make, 12
man, 14
mkgeo_ball, 91
    -e, 91
    -s, 91
    -t, 25, 91
mkgeo_contraction, 236
    -split, 236
mkgeo_grid, 13, 77, 111
    -H, 16
    -T, 15, 80
    -a, 77
    -b, 77
    -c, 79
    -d, 79
    -e, 15
    -f, 80
    -g, 80
    -q, 16
    -region, 36
    -t, 13
    -zr, 65
mkgeo_obstacle, 71
mkgeo_sector, 218
mkgeo_ugrid, 23, 111

mpirun, 18, 37, 85, 150, 152, 218,
219, 237
msh2geo, 63, 278, 279
-zr, 66
paraview, 13, 15, 45, 46, 56, 74, 79,
150, 219, 277
rheolef-config, 9
-check, 9
-docdir, 9
-exampledir, 9
sed, 25
time, 85
visualization
mesh, 13
deformed, 42
vlc, 74
zcat, 85

Index

approximation, 10
  P0, 45, 144
  P1b-P1, 58
  P1d, 45
  P1, 11, 41, 45, 57, 58, 80, 181
  P2-P1, Taylor-Hood, 50, 58, 59, 64, 70
  P2-P1d, Scott-Vogelius, 230
  P2, 11, 15, 41, 45, 57, 80, 181
  Pk, 11, 15
  Raviart-Thomas, 245, 258
  bubble, 58
  discontinuous
    trace, 258, 265
  discontinuous, 35, 44, 45, 53, 143, 148, 150, 226, 241, 245
  high-order, 11, 15, 37
  isoparametric, 92
  mixed, 50
  geometry
    curved, 23
  high-order, 23, 221
  isoparametric, 23, 221
  Raviart-Thomas, 258
argc, argv, command line arguments, 11

Babuška paradox, 70
benchmark
  driven cavity flow, 49, 80, 179, 181
  Dziuk-Elliott-Heine on a sphere, 91
  embankment, 39, 177
  flow in an abrupt contraction, 59, 66
  Leveque vortex-in-box, 150
  Olshanskii-Reusken-Grande on a torus, 95
  pipe flow, 107
  rotating hill, 75
  Zalesak slotted disk, 148
boundary condition
  Dirichlet, 10, 20, 35, 39, 49, 56, 80, 107, 115, 163, 166, 167, 179, 181, 199, 261
  mixed, 56, 59, 66
  Neumann, 28, 31, 35, 39, 56
  Poiseuille flow, 62
  Robin, 30, 69, 199
  slip, 68
  weakly imposed, 143, 163, 227, 235
broken Sobolev space $H^1(\mathcal{T}_h)$, 163, 172

class
  `Float`, 21
  `adapt_option`, 48
  `band`, 100
  `branch`, 73, 79
  `characteristic`, 76
  `eye`, 101
  `field`, 11
  `form`, 11
  `geo`, 11
  `integrate_option`, 60, 66, 76, 126
  `level_set_option`, 97
  `man`, 12, 43, 51, 101, 116
  `oditstream`, 47
  `pbl em`, 12
  `point`, 21
  `problem_mixed`, 51, 57
  `problem`, 11
  `reference manual`, 12, 43, 51, 101, 116
  `solver_abtb`, 82
  `solver_option`, 16, 169
  `solver`, 12, 16
  `space`, 11
compilation, 12
convergence
  error
    superconvergence, 253, 254
    versus mesh, 22, 93, 124, 135, 145, 165
    versus polynomial degree, 22, 93, 124
  postprocessing, 255
  residue
    rate, 111, 113
    super-linear, 118
coordinate system
  axisymmetric, 62, 64, 65
  Cartesian, 21, 36
  spherical, 91
  torus, 95

directory of example files, 9, 62, 64, 211
distributed computation, 11, 18, 218, 237

element shape, 37
error analysis, 22, 79, 93, 124, 135

file format
‘`.avi`’ avi file (video), 74
‘`.bamgcad`’ bamg geometry, 48, 277
‘`.bamg`’ bamg mesh, 48, 277
‘`.branch`’ family of fields, 74
‘`.dmn`’ domain names, 277
‘`.field`’ field, 13
‘`.field`’ multi-component field, 42
‘`.geo`’ mesh, 13, 63, 65, 277–279
‘`.gz`’ gzip compressed file, 48
‘`.mshcad`’ gmsh geometry, 49, 63, 66, 95, 278
‘`.msh`’ gmsh mesh, 49, 63, 66, 278
‘`.vtk`’ vtk file, 56
form
$(\eta\nabla u).\nabla v$, 117
$2D(\mathbf{u}) : D(\mathbf{v})$, 50, 56
$2D(\mathbf{u}) : D(\mathbf{v}) + \lambda \operatorname{div}\mathbf{u} \operatorname{div}\mathbf{v}$, 41
$2D(\mathbf{u}) : D(\mathbf{v}) + \mathbf{u}.\mathbf{v}$, 81
$\eta\nabla u.\nabla v$, 35, 109
$[\![u]\!]\{\!\{\nabla_h v.\mathbf{n}\}\!\}$, 163, 166, 173
$[\![u]\!]\{\!\{\nabla_h v.\mathbf{n}\}\!\}_w$, 168
$[\![u]\!]\{\!\{v\}\!\}$, 144
$[\![u]\!][\![v]\!]$, 144, 163, 166, 168, 173
$\nabla_s u.\nabla_s v + uv$, 90
$\nabla u.\nabla v$, 10
$\nabla u.\nabla v + uv$, 29
$\operatorname{bcurl}(\mathbf{u}).\boldsymbol{\xi}$, 64
$\operatorname{curl}(\mathbf{u}).\boldsymbol{\xi}$, 53
$\operatorname{div}(\mathbf{u})\, q$, 50, 56
energy, 10, 35, 107
product, 60
weighted, 35, 109
quadrature formula, 117
tensorial weight, 117
formal adjoint, 206
Fréchet derivative, 115
function
`adapt`, 46, 48
`catchmark`, 41, 52, 73
`compose`, 76, 83, 109, 117
`damped_newton`, 122
`diag`, 101
`dis_wall_time`, 16
`grad`, 109, 117
`integrate`, 11, 29, 60, 64, 76, 90, 109, 117, 126
on a band, 101
on the boundary, 29
`interpolate`, 22, 47, 78
`ldlt`, 33
`level_set`, 97
`newton`, 116
`norm2`, 47, 83, 109
`sqr`, 47
class-function object, 21, 77, 118
functor, 22, 126

geometry
axisymmetric, 62, 64
circle, 25, 91
contraction, 59, 66, 234
cube, 15, 279
curved, 92
line, 15, 278
obstacle, 68
pipe, 199, 213
sphere, 91
square, 13, 278
surface, 89
curvature, 276
torus, 95
Green formula, 90, 275

interface
moving, 146
yield surface, 219
internal sides of a mesh, 144

Lagrange
interpolation, 20, 22, 28, 32
multiplier, 32, 246
node, 12
Lamé coefficients, 39

Makefile, 12
matrix
bloc-diagonal
inverse, 60
block structure, 12
concatenation, 33
diagonal, 101
factorization
Choleski, 12
identity, 101
indefinite, 33
Schur complement, 251, 263
singular, 33, 101
mesh, 11, 276
adaptation, 218, 221
anisotropic, 46, 83
connected components, 102

generation, 63, 66, 276
method
augmented Lagrangian, 199, 213
BDF(p), backward differentiation formula, 147
BDF2 scheme, 81
characteristic, 75, 80
conjugate gradient algorithm, 31, 51, 57
continuation, 133
Euler explicit scheme, 154
Euler implicit scheme, 72, 75
fixed-point, 107, 118
relaxation, 108, 113
level set, 89, 97, 146
banded, 100
minres algorithm, 31, 101
Newton, 115
damped, 121
newton, 127, 130, 133, 201, 204, 206, 218
regularization, 69
Runge-Kutta scheme, 147, 154, 173
upwind scheme, 192

namespace
rheolef, 11
std, 11
norm
in $H(\mathrm{div})$, 260
in $W^{-1,p}$, 111
discrete version, 112
in $W^{1,p}$, 107
in $W_0^{1,p}$, 107
mesh-dependent on $\mathscr{S}_h$, 253
numerical flux, 249

operator
`average`, across sides, 144, 163, 250
`jump`, across sides, 144, 163, 250
adjoint, 121
curl, 53
divergence, 39
gradient, 39
symmetric part, 39
Helmholtz, 28
Helmholtz-Beltrami, 89
Laplace, 10
Laplace-Beltrami, 89

parallel computation, 11, 18, 237
penalty parameter, 163
polar coordinate system, 65
preconditioner, 51
for nearly incompressible elasticity, 57
for Stokes problem, 51
problem
Bingham, 213
Helmholtz, 28
Herschel-Bulkley, 213
Mosolov, 213
Navier-Stokes, 80, 181
Poisson, 10, 20, 30, 31, 73, 107, 109, 163, 166, 167, 199
Stokes, 58, 80, 179, 181
combustion, 127
convection-diffusion, 75
diffusion, 261
elasticity, 39, 177
heat, 72
linear tangent, 115
nonlinear, 80, 181
p-Laplacian, 107
stabilized Stokes, 60
transmission, 34, 167
yield slip, 199
Burgers equation, 157
elasticity
incompressible, 56
hyperbolic nonlinear equation, 153
oldroyd, 229
Poisson
non-constant tensorial coefficients, 115
Stokes, 49, 68
transport equation
evolution, 148, 150
steady, 143
tensor, 226
unsteady, 75, 79
projection
commuting, 245
in $H(\mathrm{div})$, 245
in L^2 norm, 45, 54, 265

quadrature formulae
Gauss, 76
Gauss-Lobatto, 76

region, 34, 36
residual term, 111, 115

singular solution, 54
space
$W^{-1,p}$, 107
$W^{-1,p}$, dual of $W_0^{1,p}$, 111
$W^{1,p}$, 107
$W_0^{1,p}$, 107
dual, 111
duality bracket $\langle .,. \rangle$, 112

weighted (axisymmetric), 65
speedup, 18, 85
stabilization, 56
stream function, 54, 64, 85
axisymmetric, 65

tensor
Cauchy stress, 39, 68
field, 45
rate of deformation, 67
visualization as ellipsoid, 44

unknown and blocked components, 12
upstream boundary, 143
upwinding, 144

visualization
animation, 74
elevation view, 13, 110
stereoscopic anaglyph, 13, 43
vortex, 64, 67
vorticity, 53