Anders Logg • Kent-Andre Mardal • Garth N. Wells
Editors

Automated Solution of Differential Equations by the Finite Element Method

The FEniCS Book

fenicsproject.org

Editors
Anders Logg
Kent-Andre Mardal
Simula Research Laboratory
PO Box 134
1325 Lysaker
Norway
logg@simula.no
kent-and@simula.no

Garth N. Wells
University of Cambridge
Department of Engineering
Trumpington Street
CB2 1PZ Cambridge
United Kingdom
gnw20@cam.ac.uk

ISSN 1439-7358
ISBN 978-3-642-23098-1
e-ISBN 978-3-642-23099-8
DOI 10.1007/978-3-642-23099-8

Copyright © 2011 The FEniCS Project.

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 chapter entitled "GNU Free Documentation License".

Contents

1 A FEniCS tutorial **1**
1.1 Fundamentals . . . 1
1.2 Nonlinear problems . . . 38
1.3 Time-dependent problems . . . 46
1.4 Creating more complex domains . . . 57
1.5 Handling domains with different materials . . . 59
1.6 More examples . . . 65
1.7 Miscellaneous topics . . . 66

I Methodology **71**

2 The finite element method **73**
2.1 A simple model problem . . . 73
2.2 Finite element discretization . . . 74
2.3 Finite element abstract formalism . . . 76
2.4 Finite element function spaces . . . 78
2.5 Finite element solvers . . . 84
2.6 Finite element error estimation and adaptivity . . . 85
2.7 Automating the finite element method . . . 89
2.8 Historical notes . . . 90

3 Common and unusual finite elements **91**
3.1 The finite element definition . . . 91
3.2 Notation . . . 92
3.3 H^1 finite elements . . . 94
3.4 $H(\mathrm{div})$ finite elements . . . 97
3.5 $H(\mathrm{curl})$ finite elements . . . 102
3.6 L^2 finite elements . . . 104
3.7 H^2 finite elements . . . 105
3.8 Enriching finite elements . . . 109
3.9 Finite element exterior calculus . . . 110
3.10 Summary . . . 110

4 Constructing general reference finite elements **117**
4.1 Background . . . 117
4.2 Preliminaries . . . 118
4.3 Mathematical framework . . . 119

4.4 Examples of elements . . . 123
4.5 Operations on the polynomial spaces . . . 126

5 Finite element variational forms **129**
5.1 Background . . . 129
5.2 Multilinear forms . . . 130
5.3 Discretizing multilinear forms . . . 132
5.4 The action of a multilinear form . . . 133
5.5 The derivative of a multilinear form . . . 134
5.6 The adjoint of a bilinear form . . . 135
5.7 A note on the order of trial and test functions . . . 136

6 Finite element assembly **137**
6.1 Assembly algorithm . . . 137
6.2 Implementation . . . 139
6.3 Symmetric application of boundary conditions . . . 140
6.4 Parallel assembly . . . 140
6.5 Matrix-free methods . . . 141

7 Quadrature representation of finite element variational forms **143**
7.1 Standard quadrature representation . . . 143
7.2 Quadrature optimizations . . . 144
7.3 Performance comparisons . . . 150
7.4 Automatic selection of representation . . . 154

8 Tensor representation of finite element variational forms **157**
8.1 Tensor representation for Poisson's equation . . . 157
8.2 A representation theorem . . . 159
8.3 Extensions and limitations . . . 159

9 Discrete optimization of finite element matrix evaluation **161**
9.1 Optimization framework . . . 161
9.2 Topological optimization . . . 162
9.3 A graph problem . . . 164
9.4 Geometric optimization . . . 164
9.5 Optimization by dense linear algebra . . . 166
9.6 Notes on implementation . . . 167

II Implementation **169**

10 DOLFIN: A C++/Python finite element library **171**
10.1 Overview . . . 171
10.2 User interfaces . . . 171
10.3 Functionality . . . 174
10.4 Implementation notes . . . 214
10.5 Historical notes . . . 222

11 FFC: the FEniCS form compiler **223**
11.1 Compilation of variational forms . . . 223
11.2 Compiler interfaces . . . 224
11.3 Parameters affecting code generation . . . 228
11.4 Compiler design . . . 228
11.5 Form representation . . . 230
11.6 Optimization . . . 231
11.7 Just-in-time compilation . . . 232
11.8 Extending FFC . . . 233
11.9 Historical notes . . . 233

12 FErari: an optimizing compiler for variational forms **235**
12.1 Optimized form compilation . . . 235
12.2 Performance of optimizations . . . 237
12.3 Conclusions . . . 242
12.4 Historical notes . . . 242

13 FIAT: numerical construction of finite element basis functions **243**
13.1 Prime basis: collapsed-coordinate polynomials . . . 243
13.2 Representing polynomials and functionals . . . 244
13.3 Other polynomial spaces . . . 247
13.4 Conveying topological information to clients . . . 249
13.5 Functional evaluation . . . 250
13.6 Overview of fundamental class structure . . . 250

14 Instant: just-in-time compilation of C/C++ in Python **253**
14.1 Brief overview of Instant and its role in FEniCS . . . 253
14.2 Examples . . . 253
14.3 Errors encountered when using Instant . . . 258
14.4 Instant explained . . . 260
14.5 Instant API . . . 263
14.6 Related work . . . 268

15 SyFi and SFC: symbolic finite elements and form compilation **269**
15.1 GiNaC and Swiginac . . . 269
15.2 SyFi: symbolic finite elements . . . 270
15.3 SFC: SyFi form compiler . . . 273
15.4 Code generation design . . . 275

16 UFC: a finite element code generation interface **279**
16.1 Overview . . . 279
16.2 Finite element discretization and assembly . . . 280
16.3 The UFC interface . . . 281
16.4 Examples . . . 286
16.5 Numbering conventions . . . 290
16.6 Discussion . . . 297
16.7 Historical notes . . . 298

17 UFL: a finite element form language **299**
17.1 Overview . . . 300
17.2 Defining finite element spaces . . . 302
17.3 Defining forms . . . 303
17.4 Defining expressions . . . 305
17.5 Form operators . . . 311
17.6 Expression representation . . . 315
17.7 Computing derivatives . . . 321
17.8 Algorithms . . . 325
17.9 Implementation issues . . . 332
17.10 Conclusions and future directions . . . 334
17.11 Acknowledgements . . . 334

18 Unicorn: a unified continuum mechanics solver **335**
18.1 Background . . . 335
18.2 Unified continuum modeling . . . 337
18.3 Space-time general Galerkin discretization . . . 338
18.4 Implementation . . . 340
18.5 Solving continuum mechanics problems . . . 351
18.6 Acknowledgments . . . 355

19 Lessons learned in mixed language programming **361**
19.1 Background . . . 361
19.2 Using SWIG . . . 361
19.3 SWIG and the DOLFIN Python interface . . . 366
19.4 JIT compilation of UFL forms, `Expressions` and `SubDomains` . . . 376
19.5 Debugging mixed language applications . . . 377

III Applications **379**

20 Finite elements for incompressible fluids **381**
20.1 Stokes equations . . . 381
20.2 Finite element formulations for the mixed Stokes problem . . . 382
20.3 A penalty approach: the Scott–Vogelius method . . . 385
20.4 Numerical tests . . . 387
20.5 Conclusions . . . 394

21 A comparison of finite element schemes for the incompressible Navier–Stokes equations **395**
21.1 Preliminaries . . . 395
21.2 Implementation . . . 396
21.3 Solvers . . . 397
21.4 Test problems and results . . . 405
21.5 Summary of results . . . 413
21.6 Discussion . . . 415
21.7 Conclusions . . . 417

22 Simulation of transitional flows **419**
22.1 Background . . . 419
22.2 Numerical method and energy conservation . . . 421
22.3 Numerical investigations . . . 429
22.4 Conclusions . . . 437

23 Computational hemodynamics **439**
23.1 Medical background . . . 439
23.2 Preliminaries . . . 441
23.3 Gender differences in the intracranial vasculature . . . 444
23.4 CFD versus 4D PC MRA in an experimental canine aneurysm . . . 445
23.5 Patient-specific Circle of Willis . . . 450

24 Cerebrospinal fluid flow **455**
24.1 Medical background . . . 455
24.2 Mathematical description . . . 456
24.3 Numerical experiments . . . 457
24.4 Conclusions . . . 469

25 Improved Boussinesq equations for surface water waves **471**
25.1 Overview . . . 471
25.2 DOLFWAVE . . . 473
25.3 Model derivation . . . 474
25.4 Linear dispersion relation . . . 478
25.5 Numerical methods . . . 480
25.6 Wave generation . . . 482
25.7 Reflective walls and sponge layers . . . 483
25.8 Linear stability analysis . . . 484
25.9 Model validation and numerical applications . . . 486
25.10Conclusions . . . 503

26 Applications in solid mechanics **505**
26.1 Background . . . 505
26.2 Governing equations . . . 506
26.3 Constitutive models . . . 508
26.4 Time integration . . . 511
26.5 Linearization issues for complex constitutive models . . . 511
26.6 Implementation and examples . . . 515
26.7 Future developments . . . 522

27 A computational framework for nonlinear elasticity **527**
27.1 Brief overview of nonlinear elasticity theory . . . 527
27.2 Numerical methods and further implementation details . . . 532
27.3 Examples of CBC.Twist usage . . . 536
27.4 Conclusions . . . 543

28 Turbulent flow and fluid–structure interaction **545**
28.1 Background . . . 545
28.2 Simulation of high Re turbulent flow . . . 545
28.3 Turbulent boundary layers . . . 546
28.4 Adaptivity and a posteriori error estimation . . . 546
28.5 Robust fluid–structure coupling . . . 547
28.6 Applications . . . 548
28.7 Acknowledgments . . . 553

29 An adaptive finite element solver for fluid–structure interaction problems **555**
29.1 Fluid–structure interaction . . . 555
29.2 FSI and the ALE computational framework . . . 557
29.3 The FSI solver . . . 559
29.4 Duality–based error control . . . 562
29.5 Numerical examples . . . 566
29.6 Conclusions . . . 568

30 Modeling evolving discontinuities **573**
30.1 Background . . . 573
30.2 Partition-of-unity/extended finite element method . . . 574
30.3 Software components . . . 576
30.4 Design and implementation . . . 577
30.5 Examples . . . 579
30.6 Summary . . . 584

31 Dynamic simulations of convection in the Earth's mantle **587**
31.1 Convection in the Earth's mantle . . . 587
31.2 Mathematical statement of the problem . . . 588
31.3 Numerical method . . . 590
31.4 Implementation . . . 594
31.5 Results . . . 595
31.6 Discussion and concluding remarks . . . 599

32 Automatic calibration of depositional models **603**
32.1 Issues in sedimentary deposition . . . 603
32.2 A multidimensional sedimentation model . . . 603
32.3 An inverse approach . . . 604
32.4 The Landweber algorithm . . . 604
32.5 Evaluation of gradients by duality arguments . . . 605
32.6 Aspects of the implementation . . . 607
32.7 Numerical experiments . . . 608
32.8 Results and conclusions . . . 611

33 A coupled stochastic and deterministic model of Ca^{2+} dynamics in the dyadic cleft **613**
33.1 Biological background . . . 613
33.2 Mathematical models . . . 614
33.3 Numerical methods for the continuous system . . . 617
33.4 `diffsim`: an event-driven simulator . . . 625
33.5 Discussion . . . 627

34 Electromagnetic waveguide analysis **629**
34.1 Formulation . . . 630
34.2 Implementation . . . 634
34.3 Examples . . . 637
34.4 Conclusions . . . 642

35 Block preconditioning of systems of PDEs **643**
35.1 Abstract framework for constructing preconditioners . . . 643
35.2 Overview of CBC.Block . . . 644
35.3 Numerical examples . . . 645
35.4 Conclusions . . . 654

36 Automated testing of saddle point stability conditions **655**
36.1 Background . . . 655
36.2 Why does discrete stability matter? . . . 656
36.3 Discrete stability . . . 657
36.4 Eigenvalue problems associated with saddle point stability . . . 660
36.5 Automating the stability testing . . . 661
36.6 Examples . . . 665
36.7 Conclusions . . . 668

List of authors **671**

GNU Free Documentation License **679**
0. Preamble . . . 679
1. Applicability and definitions . . . 679
2. Verbatim copying . . . 680
3. Copying in quantity . . . 681
4. Modifications . . . 681
5. Combining documents . . . 683
6. Collections of documents . . . 683
7. Aggregation with independent works . . . 683
8. Translation . . . 683
9. Termination . . . 684
10. Future revisions of this license . . . 684
11. Relicensing . . . 684

References **685**

Index **713**

Preface

The FEniCS Project set out in 2003 with an idea to automate the solution of mathematical models based on differential equations. Initially, the FEniCS Project consisted of two libraries: DOLFIN and FIAT. Since then, the project has grown and now consists of the core components DOLFIN, FFC, FIAT, Instant, UFC and UFL. Other FEniCS components and applications described in this book are SyFi/SFC, FErari, ASCoT, Unicorn, CBC.Block, CBC.RANS, CBC.Solve and DOLFWAVE.

This book is written by researchers and developers behind the FEniCS Project. The presentation spans mathematical background, software design and the use of FEniCS in applications. The mathematical framework is outlined in Part I, the implementation of central components is described in Part II, while Part III concerns a wide range of applications. New users of FEniCS may find the tutorial included as the opening chapter particularly useful.

Feedback on this book is welcome, and can be given at `https://launchpad.net/fenics-book`. Use the Launchpad system to file bug reports if you find errors in the text. For more information about the FEniCS Project, access to the software presented in this book, documentation, articles and presentations, visit the FEniCS Project web site at `http://fenicsproject.org`. Some of the chapters in this book are accompanied by supplementary material in the form of code examples. These code examples can be downloaded from `http://fenicsproject.org/book/`.

Anders Logg, Kent-Andre Mardal and Garth N. Wells
Oslo and Cambridge, October 2011

1 A FEniCS tutorial

By Hans Petter Langtangen

This chapter presents a FEniCS tutorial to get new users quickly up and running with solving differential equations. FEniCS can be programmed both in C++ and Python, but this tutorial focuses exclusively on Python programming since this is the simplest approach to exploring FEniCS for beginners and it does not compromise on performance. After having digested the examples in this tutorial, the reader should be able to learn more from the FEniCS documentation and from the other chapters in this book.

1.1 Fundamentals

FEniCS is a user-friendly tool for solving partial differential equations (PDEs). The goal of this tutorial is to get you started with FEniCS through a series of simple examples that demonstrate

- how to define the PDE problem in terms of a variational problem,
- how to define simple domains,
- how to deal with Dirichlet, Neumann, and Robin conditions,
- how to deal with variable coefficients,
- how to deal with domains built of several materials (subdomains),
- how to compute derived quantities like the flux vector field or a functional of the solution,
- how to quickly visualize the mesh, the solution, the flux, etc.,
- how to solve nonlinear PDEs in various ways,
- how to deal with time-dependent PDEs,
- how to set parameters governing solution methods for linear systems,
- how to create domains of more complex shape.

The mathematics of the illustrations is kept simple to better focus on FEniCS functionality and syntax. This means that we mostly use the Poisson equation and the time-dependent diffusion equation as model problems, often with input data adjusted such that we get a very simple solution that can be exactly reproduced by any standard finite element method over a uniform, structured mesh. This latter property greatly simplifies the verification of the implementations. Occasionally we insert a physically more relevant example to remind the reader that changing the PDE and boundary conditions to something more real might often be a trivial task.

FEniCS may seem to require a thorough understanding of the abstract mathematical version of the finite element method as well as familiarity with the Python programming language. Nevertheless, it turns out that many are able to pick up the fundamentals of finite elements *and* Python programming as they go along with this tutorial. Simply keep on reading and try out the examples. You will be amazed of how easy it is to solve PDEs with FEniCS!

Reading this tutorial obviously requires access to a machine where the FEniCS software is installed. Section 1.7.5 explains briefly how to install the necessary tools. All the examples discussed in the following are available as executable Python source code files in a directory tree.

1.1.1 The Poisson equation

Our first example regards the Poisson problem,

$$\begin{aligned} -\Delta u &= f \quad \text{in } \Omega, \\ u &= u_0 \quad \text{on } \partial\Omega. \end{aligned} \tag{1.1}$$

Here, $u = u(x)$ is the unknown function, $f = f(x)$ is a prescribed function, Δ is the Laplace operator (also often written as ∇^2), Ω is the spatial domain, and $\partial\Omega$ is the boundary of Ω. A stationary PDE like this, together with a complete set of boundary conditions, constitute a *boundary-value problem*, which must be precisely stated before it makes sense to start solving it with FEniCS.

In two space dimensions with coordinates x and y, we can write out the Poisson equation (1.1) as

$$-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} = f(x, y). \tag{1.2}$$

The unknown u is now a function of two variables, $u(x, y)$, defined over a two-dimensional domain Ω.

The Poisson equation (1.1) arises in numerous physical contexts, including heat conduction, electrostatics, diffusion of substances, twisting of elastic rods, inviscid fluid flow, and water waves. Moreover, the equation appears in numerical splitting strategies of more complicated systems of PDEs, in particular the Navier–Stokes equations.

Solving a physical problem with FEniCS consists of the following steps:

1. Identify the PDE and its boundary conditions.
2. Reformulate the PDE problem as a variational problem.
3. Make a Python program where the formulas in the variational problem are coded, along with definitions of input data such as f, u_0, and a mesh for Ω in (1.1).
4. Add statements in the program for solving the variational problem, computing derived quantities such as ∇u, and visualizing the results.

We shall now go through steps 2–4 in detail. The key feature of FEniCS is that steps 3 and 4 result in fairly short code, while most other software frameworks for PDEs require much more code and more technically difficult programming.

1.1.2 Variational formulation

FEniCS makes it easy to solve PDEs if finite elements are used for discretization in space and the problem is expressed as a *variational problem*. Readers who are not familiar with variational problems will get a brief introduction to the topic in this tutorial, and in the forthcoming chapter, but getting

and reading a proper book on the finite element method in addition is encouraged. Section 1.7.6 contains a list of some suitable books.

The core of the recipe for turning a PDE into a variational problem is to multiply the PDE by a function v, integrate the resulting equation over Ω, and perform integration by parts of terms with second-order derivatives. The function v which multiplies the PDE is in the mathematical finite element literature called a *test function*. The unknown function u to be approximated is referred to as a *trial function*. The terms test and trial function are used in FEniCS programs too. Suitable function spaces must be specified for the test and trial functions. For standard PDEs arising in physics and mechanics such spaces are well known.

In the present case, we first multiply the Poisson equation by the test function v and integrate:

$$-\int_\Omega (\Delta u)v \,\mathrm{d}x = \int_\Omega fv \,\mathrm{d}x. \tag{1.3}$$

Then we apply integration by parts to the integrand with second-order derivatives:

$$-\int_\Omega (\Delta u)v \,\mathrm{d}x = \int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x - \int_{\partial\Omega} \frac{\partial u}{\partial n} v \,\mathrm{d}s, \tag{1.4}$$

where $\partial u/\partial n$ is the derivative of u in the outward normal direction on the boundary. The test function v is required to vanish on the parts of the boundary where u is known, which in the present problem implies that $v = 0$ on the whole boundary $\partial\Omega$. The second term on the right-hand side of (1.4) therefore vanishes. From (1.3) and (1.4) it follows that

$$\int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x = \int_\Omega fv \,\mathrm{d}x. \tag{1.5}$$

This equation is supposed to hold for all v in some function space $\hat{V}$. The trial function u lies in some (possibly different) function space V. We refer to (1.5) as the *weak form* of the original boundary-value problem (1.1).

The proper statement of our variational problem now goes as follows: find $u \in V$ such that

$$\int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x = \int_\Omega fv \,\mathrm{d}x \quad \forall\, v \in \hat{V}. \tag{1.6}$$

The trial and test spaces V and $\hat{V}$ are in the present problem defined as

$$\begin{aligned} V &= \{v \in H^1(\Omega) : v = u_0 \text{ on } \partial\Omega\}, \\ \hat{V} &= \{v \in H^1(\Omega) : v = 0 \text{ on } \partial\Omega\}. \end{aligned} \tag{1.7}$$

In short, $H^1(\Omega)$ is the mathematically well-known Sobolev space containing functions v such that v^2 and $|\nabla v|^2$ have finite integrals over Ω. The solution of the underlying PDE must lie in a function space where also the derivatives are continuous, but the Sobolev space $H^1(\Omega)$ allows functions with discontinuous derivatives. This weaker continuity requirement of u in the variational statement (1.6), caused by the integration by parts, has great practical consequences when it comes to constructing finite elements.

To solve the Poisson equation numerically, we need to transform the continuous variational problem (1.6) to a discrete variational problem. This is done by introducing *finite-dimensional* test and trial spaces, often denoted as $V_h \subset V$ and $\hat{V}_h \subset \hat{V}$. The discrete variational problem reads: find $u_h \in V_h \subset V$ such that

$$\int_\Omega \nabla u_h \cdot \nabla v \,\mathrm{d}x = \int_\Omega fv \,\mathrm{d}x \quad \forall\, v \in \hat{V}_h \subset \hat{V}. \tag{1.8}$$

The choice of V_h and $\hat{V}_h$ follows directly from the kind of finite elements we want to apply in our problem. For example, choosing the well-known linear triangular element with three nodes implies that V_h and $\hat{V}_h$ are the spaces of all piecewise linear functions over a mesh of triangles, where the functions in $\hat{V}_h$ are zero on the boundary and those in V_h equal u_0 on the boundary.

The mathematics literature on variational problems writes u_h for the solution of the discrete problem and u for the solution of the continuous problem. To obtain (almost) a one-to-one relationship between the mathematical formulation of a problem and the corresponding FEniCS program, we shall use u for the solution of the discrete problem and u_e for the exact solution of the continuous problem, *if* we need to explicitly distinguish between the two. In most cases, we will introduce the PDE problem with u as unknown, derive a variational equation $a(u,v) = L(v)$ with $u \in V$ and $v \in \hat{V}$, and then simply discretize the problem by saying that we choose finite-dimensional spaces for V and $\hat{V}$. This restriction of V implies that u becomes a discrete finite element function. In practice this means that we turn our PDE problem into a continuous variational problem, create a mesh and specify an element type, and then let V correspond to this mesh and element choice. Depending upon whether V is infinite- or finite-dimensional, u will be the exact or approximate solution.

It turns out to be convenient to introduce a unified notation for a linear weak form like (1.8):

$$a(u,v) = L(v). \tag{1.9}$$

In the present problem we have that

$$a(u,v) = \int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x, \tag{1.10}$$

$$L(v) = \int_\Omega fv \,\mathrm{d}x. \tag{1.11}$$

From the mathematics literature, $a(u,v)$ is known as a *bilinear form* and $L(v)$ as a *linear form*. We shall in every linear problem we solve identify the terms with the unknown u and collect them in $a(u,v)$, and similarly collect all terms with only known functions in $L(v)$. The formulas for a and L are then coded directly in the program.

To summarize, before making a FEniCS program for solving a PDE, we must first perform two steps:

1. Turn the PDE problem into a discrete variational problem: find $u \in V$ such that

$$a(u,v) = L(v) \quad \forall\, v \in \hat{V}. \tag{1.12}$$

2. Specify the choice of spaces (V and $\hat{V}$), which means specifying the mesh and type of finite elements.

1.1.3 Implementation

The test problem so far has a general domain Ω and general functions u_0 and f. For our first implementation we must decide on specific choices of Ω, u_0, and f. It will be wise to construct a specific problem where we can easily check that the computed solution is correct. Let us start with specifying an exact solution

$$u_{\mathrm{e}}(x,y) = 1 + x^2 + 2y^2 \tag{1.13}$$

on some 2D domain. By inserting (1.13) in our Poisson problem, we find that $u_{\mathrm{e}}(x,y)$ is a solution if

$$f(x,y) = -6, \quad u_0(x,y) = u_{\mathrm{e}}(x,y) = 1 + x^2 + 2y^2,$$

regardless of the shape of the domain. We choose here, for simplicity, the domain to be the unit square,

$$\Omega = [0,1] \times [0,1].$$

The reason for specifying the solution (1.13) is that the finite element method, with a rectangular domain uniformly partitioned into linear triangular elements, will exactly reproduce a second-order polynomial at the vertices of the cells, regardless of the size of the elements. This property allows us to verify the implementation by comparing the computed solution, called u in this document (except when setting up the PDE problem), with the exact solution, denoted by u_e: u should equal u to machine precision *at the nodes*. Test problems with this property will be frequently constructed throughout this tutorial.

A FEniCS program for solving the Poisson equation in 2D with the given choices of u_0, f, and Ω may look as follows:

Python code

```
from dolfin import *

# Create mesh and define function space
mesh = UnitSquare(6, 4)
V = FunctionSpace(mesh, "Lagrange", 1)

# Define boundary conditions
u0 = Expression("1 + x[0]*x[0] + 2*x[1]*x[1]")

def u0_boundary(x, on_boundary):
    return on_boundary

bc = DirichletBC(V, u0, u0_boundary)

# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)
f = Constant(-6.0)
a = inner(nabla_grad(u), nabla_grad(v))*dx
L = f*v*dx

# Compute solution
u = Function(V)
solve(a == L, u, bc)

# Plot solution and mesh
plot(u)
plot(mesh)

# Dump solution to file in VTK format
file = File("poisson.pvd")
file << u

# Hold plot
interactive()
```

The complete code can be found in the file `d1_p2D.py` in the directory `stationary/poisson`.

We shall now dissect this FEniCS program in detail. The program is written in the Python programming language. You may either take a quick look at a Python tutorial (The Python Tutorial) to pick up the basics of Python if you are unfamiliar with the language, or you may learn enough Python as you go along with the examples in the present tutorial. The latter strategy has proven to work for many newcomers to FEniCS. Section 1.7.7 lists some relevant Python books.

The listed FEniCS program defines a finite element mesh, the discrete function spaces V and $\hat{V}$ corresponding to this mesh and the element type, boundary conditions for u (the function u_0), $a(u,v)$, and $L(v)$. Thereafter, the unknown trial function u is computed. Then we can investigate u visually or analyze the computed values.

The first line in the program,

Python code

```
from dolfin import *
```

imports the key classes `UnitSquare`, `FunctionSpace`, `Function`, and so forth, from the DOLFIN library. All FEniCS programs for solving PDEs by the finite element method normally start with this line. DOLFIN is a software library with efficient and convenient C++ classes for finite element computing, and `dolfin` is a Python package providing access to this C++ library from Python programs. You can think of FEniCS as an umbrella, or project name, for a set of computational components, where DOLFIN is one important component for writing finite element programs. The `dolfin` package applies other components in the FEniCS suite under the hood, but newcomers to FEniCS programming do not need to care about this.

The statement

Python code

```
mesh = UnitSquare(6, 4)
```

defines a uniform finite element mesh over the unit square $[0,1] \times [0,1]$. The mesh consists of *cells*, which are triangles with straight sides. The parameters 6 and 4 tell that the square is first divided into 6×4 rectangles, and then each rectangle is divided into two triangles. The total number of triangles then becomes 48. The total number of vertices in this mesh is $7 \cdot 5 = 35$. DOLFIN offers some classes for creating meshes over very simple geometries. For domains of more complicated shape one needs to use a separate *preprocessor* program to create the mesh (see Section 1.4). The FEniCS program will then read the mesh from file.

Having a mesh, we can define a discrete function space `V` over this mesh:

Python code

```
V = FunctionSpace(mesh, "Lagrange", 1)
```

The second argument reflects the type of element, while the third argument is the degree of the basis functions on the element. The type of element is here `"Lagrange"`, implying the standard Lagrange family of elements (some FEniCS programs use `"CG"`, for Continuous Galerkin, as a synonym for `"Lagrange"`). With degree 1, we simply get the standard linear Lagrange element, which is a triangle with nodes at the three vertices. Some finite element practitioners refer to this element as the "linear triangle". The computed u will be continuous and linearly varying in x and y over each cell in the mesh. Higher-degree polynomial approximations over each cell are trivially obtained by increasing the third parameter in `FunctionSpace`. Changing the second parameter to `"DG"` creates a function space for discontinuous Galerkin methods.

In mathematics, we distinguish between the trial and test spaces V and $\hat{V}$. The only difference in the present problem is the boundary conditions. In FEniCS we do not specify the boundary conditions as part of the function space, so it is sufficient to work with one common space `V` for the test and trial functions in the program:

Python code

```
u = TrialFunction(V)
v = TestFunction(V)
```

The next step is to specify the boundary condition: $u = u_0$ on $\partial\Omega$. This is done by

Python code

```
bc = DirichletBC(V, u0, u0_boundary)
```

where `u0` is an instance holding the u_0 values, and `u0_boundary` is a function (or object) describing whether a point lies on the boundary where u is specified.

Boundary conditions of the type $u = u_0$ are known as *Dirichlet conditions*, and also as *essential boundary conditions* in a finite element context. Naturally, the name of the DOLFIN class holding the information about Dirichlet boundary conditions is `DirichletBC`.

The `u0` variable refers to an `Expression` object, which is used to represent a mathematical function. The typical construction is

Python code

```
u0 = Expression(formula)
```

where `formula` is a string containing the mathematical expression. This formula is written with C++ syntax (the expression is automatically turned into an efficient, compiled C++ function, see Section 1.7.3 and Chapter 10 for details on the syntax). The independent variables in the function expression are supposed to be available as a point vector `x`, where the first element `x[0]` corresponds to the x coordinate, the second element `x[1]` to the y coordinate, and (in a three-dimensional problem) `x[2]` to the z coordinate. With our choice of $u_0(x, y) = 1 + x^2 + 2y^2$, the formula string must be written as `1 + x[0]*x[0] + 2*x[1]*x[1]`:

Python code

```
u0 = Expression("1 + x[0]*x[0] + 2*x[1]*x[1]")
```

The information about where to apply the `u0` function as boundary condition is coded in a function `u0_boundary`:

Python code

```
def u0_boundary(x, on_boundary):
    return on_boundary
```

A function like `u0_boundary` for marking the boundary must return a boolean value: `True` if the given point `x` lies on the Dirichlet boundary and `False` otherwise. The argument `on_boundary` is supplied by DOLFIN and equals `True` if `x` is on the physical boundary of the mesh. In the present case, where we are supposed to return `True` for all points on the boundary, we can just return the supplied value of `on_boundary`. The `u0_boundary` function will be called for every discrete point in the mesh, which allows us to have boundaries where u are known also inside the domain, if desired.

One can also omit the `on_boundary` argument, but in that case we need to test on the value of the coordinates in `x`:

Python code

```
def u0_boundary(x):
    return x[0] == 0 or x[1] == 0 or x[0] == 1 or x[1] == 1
```

As for the formula in `Expression` objects, `x` in the `u0_boundary` function represents a point in space with coordinates `x[0]`, `x[1]`, etc. Comparing floating-point values using an exact match test with `==` is not good programming practice, because small round-off errors in the computations of the `x` values could make a test `x[0] == 1` become false even though `x` lies on the boundary. A better test is to check for equality with a tolerance:

Python code

```
def u0_boundary(x):
    tol = 1E-15
    return abs(x[0]) < tol or \
           abs(x[1]) < tol or \
           abs(x[0] - 1) < tol or \
           abs(x[1] - 1) < tol
```

Before defining $a(u,v)$ and $L(v)$ we have to specify the f function:

Python code

```
f = Expression("-6")
```

When f is constant over the domain, `f` can be more efficiently represented as a `Constant` object:

Python code

```
f = Constant(-6.0)
```

Now we have all the objects we need in order to specify this problem's $a(u,v)$ and $L(v)$:

Python code

```
a = inner(nabla_grad(u), nabla_grad(v))*dx
L = f*v*dx
```

In essence, these two lines specify the PDE to be solved. Note the very close correspondence between the Python syntax and the mathematical formulas $\nabla u \cdot \nabla v \,\mathrm{d}x$ and $fv\,\mathrm{d}x$. This is a key strength of FEniCS: the formulas in the variational formulation translate directly to very similar Python code, a feature that makes it easy to specify PDE problems with lots of PDEs and complicated terms in the equations. The language used to express weak forms is called UFL (Unified Form Language) and is an integral part of FEniCS.

Instead of `nabla_grad` we could also just have written `grad` in the examples in this tutorial. However, when taking gradients of vector fields, `grad` and `nabla_grad` differ. The latter is consistent with the tensor algebra commonly used to derive vector and tensor PDEs, where ∇ acts as a vector operator, and therefore this author prefers to always use `nabla_grad`.

Having `a` and `L` defined, and information about essential (Dirichlet) boundary conditions in `bc`, we can compute the solution, a finite element function `u`, by

Python code

```
u = Function(V)
solve(a == L, u, bc)
```

Some prefer to replace `a` and `L` by an `equation` variable, which is accomplished by this equivalent code:

Python code

```
equation = inner(nabla_grad(u), nabla_grad(v))*dx == f*v*dx
u = Function(V)
solve(equation, u, bc)
```

Note that we first defined the variable `u` as a `TrialFunction` and used it to represent the unknown in the form `a`. Thereafter, we redefined `u` to be a `Function` object representing the solution; that is, the computed finite element function u. This redefinition of the variable `u` is possible in Python and often done in FEniCS applications. The two types of objects that `u` refers to are equal from a mathematical point of view, and hence it is natural to use the same variable name for both objects. In a program,

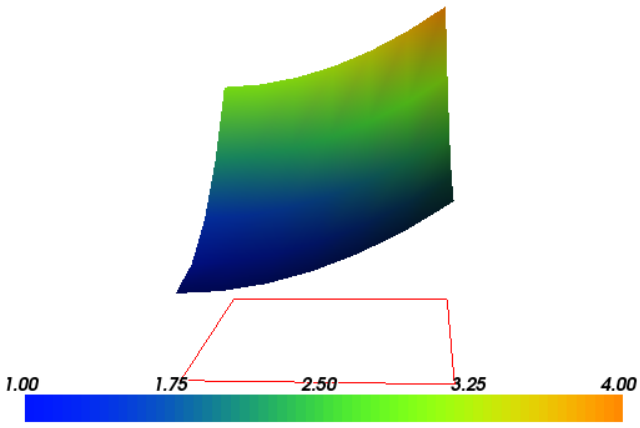


Figure 1.1: Plot of the solution in the first FEniCS example. (A bounding box around the mesh is added by pressing `o` in the plot window, and the mouse buttons are then used to rotate and move the plot, see Section 1.1.8.)

however, `TrialFunction` objects must always be used for the unknowns in the problem specification (the form `a`), while `Function` objects must be used for quantities that are computed (known).

The simplest way of quickly looking at `u` and the mesh is to say

Python code

```
plot(u)
plot(mesh)
interactive()
```

The `interactive()` call is necessary for the plot to remain on the screen. With the left, middle, and right mouse buttons you can rotate, translate, and zoom (respectively) the plotted surface to better examine what the solution looks like. Figures 1.1 and 1.2 display the resulting u function and the finite element mesh, respectively.

It is also possible to dump the computed solution to file, e.g., in the VTK format:

Python code

```
file = File("poisson.pvd")
file << u
```

The `poisson.pvd` file can now be loaded into any front-end to VTK, say ParaView or VisIt. The `plot` function is intended for quick examination of the solution during program development. More in-depth visual investigations of finite element solutions will normally benefit from using highly professional tools such as ParaView and VisIt.

The next three sections deal with some technicalities about specifying the solution method for linear systems (so that you can solve large problems) and examining array data from the computed solution (so that you can check that the program is correct). These technicalities are scattered around in forthcoming programs. However, the impatient reader who is more interested in seeing the previous program being adapted to a real physical problem, and play around with some interesting visualizations, can safely jump to Section 1.1.7. Information in the intermediate sections can be studied on demand.

1.1.4 Controlling the solution process

Sparse LU decomposition (Gaussian elimination) is used by default to solve linear systems of equations in FEniCS programs. This is a very robust and recommended method for a few thousand unknowns

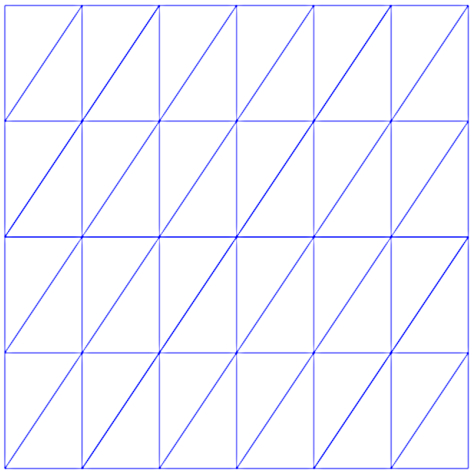


Figure 1.2: Plot of the mesh in the first FEniCS example.

in the equation system, and may hence be the method of choice in many 2D and smaller 3D problems. However, sparse LU decomposition becomes slow and memory demanding in large problems. This fact forces the use of iterative methods, which are faster and require much less memory.

Preconditioned Krylov solvers is a type of popular iterative methods that are easily accessible in FEniCS programs. The Poisson equation results in a symmetric, positive definite coefficient matrix, for which the optimal Krylov solver is the Conjugate Gradient (CG) method. Incomplete LU factorization (ILU) is a popular and robust all-round preconditioner, so let us try the CG–ILU pair:

Python code

```
solve(a == L, u, bc)
      solver_parameters={"linear_solver": "cg",
                         "preconditioner": "ilu"})
# Alternative syntax
solve(a == L, u, bc,
      solver_parameters=dict(linear_solver="cg",
                             preconditioner="ilu"))
```

Section 1.7.4 lists the most popular choices of Krylov solvers and preconditioners available in FEniCS.

The actual CG and ILU implementations that are brought into action depends on the choice of linear algebra package. FEniCS interfaces several linear algebra packages, called *linear algebra backends* in FEniCS terminology. PETSc is the default choice if DOLFIN is compiled with PETSc, otherwise uBLAS. Epetra (Trilinos) and MTL4 are two other supported backends. Which backend to apply can be controlled by setting

Python code

```
parameters["linear_algebra_backend"] = backendname
```

where `backendname` is a string, either `"PETSc"`, `"uBLAS"`, `"Epetra"`, or `"MTL4"`. All these backends offer high-quality implementations of both iterative and direct solvers for linear systems of equations.

A common platform for FEniCS users is Ubuntu Linux. The FEniCS distribution for Ubuntu contains PETSc, making this package the default linear algebra backend. The default solver is sparse LU decomposition (`"lu"`), and the actual software that is called is then the sparse LU solver from UMFPACK (which PETSc has an interface to).

We will normally like to control the tolerance in the stopping criterion and the maximum number of iterations when running an iterative method. Such parameters can be set by accessing the *global*

parameter database, which is called `parameters` and behaves as a nested dictionary. Write

Python code

```
info(parameters, True)
```

to list all parameters and their default values in the database. The nesting of parameter sets is indicated through indentation in the output from `info`. According to this output, the relevant parameter set is named `"krylov_solver"`, and the parameters are set like this:

Python code

```
prm = parameters["krylov_solver"] # short form
prm["absolute_tolerance"] = 1E-10
prm["relative_tolerance"] = 1E-6
prm["maximum_iterations"] = 1000
```

Stopping criteria for Krylov solvers usually involve the norm of the residual, which must be smaller than the absolute tolerance parameter and smaller than the relative tolerance parameter times the initial residual.

To see the number of actual iterations to reach the stopping criterion, we can insert

Python code

```
set_log_level(PROGRESS)
# or
set_log_level(DEBUG)
```

A message with the equation system size, solver type, and number of iterations arises from specifying the argument `PROGRESS`, while `DEBUG` results in more information, including CPU time spent in the various parts of the matrix assembly and solve process.

The complete solution process with control of the solver parameters now contains the statements

Python code

```
prm = parameters["krylov_solver"] # short form
prm["absolute_tolerance"] = 1E-10
prm["relative_tolerance"] = 1E-6
prm["maximum_iterations"] = 1000
set_log_level(PROGRESS)

solve(a == L, u, bc,
      solver_parameters={"linear_solver": "cg",
                         "preconditioner": "ilu"})
```

The demo program `d2_p2D.py` in the `stationary/poisson` directory incorporates the above shown control of the linear solver and precnditioner, but is otherwise similar to the previous `d1_p2D.py` program.

We remark that default values for the global parameter database can be defined in an XML file, see the example file `dolfin_parameters.xml` in the directory `stationary/poisson`. If such a file is found in the directory where a FEniCS program is run, this file is read and used to initialize the `parameters` object. Otherwise, the file `.config/fenics/dolfin_parameters.xml` in the user's home directory is read, if it exists. The XML file can also be in gzip'ed form with the extension `.xml.gz`.

1.1.5 Linear variational problem and solver objects

The `solve(a == L, u, bc)` call is just a compact syntax alternative to a slightly more comprehensive specification of the variational equation and the solution of the associated linear system. This

alternative syntax is used in a lot of FEniCS applications and will also be used later in this tutorial, so we show it already now:

Python code

```
u = Function(V)
problem = LinearVariationalProblem(a, L, u, bc)
solver  = LinearVariationalSolver(problem)
solver.solve()
```

Many objects have an attribute `parameters` corresponding to a parameter set in the global `parameters` database, but local to the object. Here, `solver.parameters` play that role. Setting the CG method with ILU preconditiong as solution method and specifying solver-specific parameters can be done like this:

Python code

```
solver.parameters["linear_solver"] = "cg"
solver.parameters["preconditioner"] = "ilu"
cg_prm = solver.parameters["krylov_solver"] # short form
cg_prm["absolute_tolerance"] = 1E-7
cg_prm["relative_tolerance"] = 1E-4
cg_prm["maximum_iterations"] = 1000
```

Calling `info(solver.parameters, True)` lists all the available parameter sets with default values for each parameter. Settings in the global `parameters` database are propagated to parameter sets in individual objects, with the possibility of being overwritten as done above.

The `d3_p2D.py` program modifies the `d2_p2D.py` file to incorporate objects for the variational problem and solver.

1.1.6 Examining the discrete solution

We know that, in the particular boundary-value problem of Section 1.1.3, the computed solution u should equal the exact solution at the vertices of the cells. An important extension of our first program is therefore to examine the computed values of the solution, which is the focus of the present section.

A finite element function like u is expressed as a linear combination of basis functions ϕ_j, spanning the space V:

$$\sum_{j=1}^{N} U_j \phi_j. \tag{1.14}$$

By writing `solve(a == L, u, bc)` in the program, a linear system will be formed from a and L, and this system is solved for the $U_1, \ldots, U_N$ values. The $U_1, \ldots, U_N$ values are known as *degrees of freedom* of u. For Lagrange elements (and many other element types) U_k is simply the value of u at the node with global number k. (The nodes and cell vertices coincide for linear Lagrange elements, while for higher-order elements there may be additional nodes at the facets and in the interior of cells.)

Having `u` represented as a `Function` object, we can either evaluate `u(x)` at any vertex `x` in the mesh, or we can grab all the values U_j directly by

Python code

```
u_nodal_values = u.vector()
```

The result is a DOLFIN `Vector` object, which is basically an encapsulation of the vector object used in the linear algebra package that is used to solve the linear system arising from the variational problem. Since we program in Python it is convenient to convert the `Vector` object to a standard `numpy` array for further processing:

Python code

```
u_array = u_nodal_values.array()
```

With `numpy` arrays we can write "MATLAB-like" code to analyze the data. Indexing is done with square brackets: `u_array[i]`, where the index `i` always starts at `0`.

Mesh information can be gathered from the `mesh` object, e.g.,

- `mesh.coordinates()` returns the coordinates of the vertices as an $M \times d$ `numpy` array, M being the number of vertices in the mesh and d being the number of space dimensions,
- `mesh.num_cells()` returns the number of cells (triangles) in the mesh,
- `mesh.num_vertices()` returns the number of vertices in the mesh (with our choice of linear Lagrange elements this equals the number of nodes),
- `str(mesh)` returns a short "pretty print" description of the mesh, e.g.,

  Output

  ```
  <Mesh of topological dimension 2 (triangles) with
  16 vertices and 18 cells, ordered>
  ```

  and `print mesh` is actually the same as `print str(mesh)`.

All mesh objects are of type `Mesh` so typing the command `pydoc dolfin.Mesh` in a terminal window will give a list of methods[1] that can be called through any `Mesh` object. In fact, `pydoc dolfin.X` shows the documentation of any DOLFIN name `X`.

Writing out the solution on the screen can now be done by a simple loop:

Python code

```
coor = mesh.coordinates()
if mesh.num_vertices() == len(u_array):
    for i in range(mesh.num_vertices()):
        print 'u(%8g,%8g) = %g' % (coor[i][0], coor[i][1], u_array[i])
```

The beginning of the output looks like this:

Output

```
u(       0,       0) = 1
u(0.166667,       0) = 1.02778
u(0.333333,       0) = 1.11111
u(     0.5,       0) = 1.25
u(0.666667,       0) = 1.44444
u(0.833333,       0) = 1.69444
u(       1,       0) = 2
```

For Lagrange elements of degree higher than one, the vertices do not correspond to all the nodal points and the `if`-test fails.

For verification purposes we want to compare the values of the computed `u` at the nodes (given by `u_array`) with the exact solution `u0` evaluated at the nodes. The difference between the computed and exact solution should be less than a small tolerance at all the nodes. The `Expression` object `u0` can be evaluated at any point `x` by calling `u0(x)`. Specifically, `u0(coor[i])` returns the value of `u0` at the vertex or node with global number `i`. Alternatively, we can make a finite element field `u_e`, representing the exact solution, whose values at the nodes are given by the `u0` function. With

[1] A method in Python (and other languages supporting the class construct) is simply a function in a class.

mathematics, $u_e = \sum_{j=1}^{N} E_j \phi_j$, where $E_j = u_0(x_j, y_j)$, (x_j, y_j) being the coordinates of node number j. This process is known as interpolation. FEniCS has a function for performing the operation:

Python code

```
u_e = interpolate(u0, V)
```

The maximum error can now be computed as

Python code

```
u_e_array = u_e.vector().array()
print "Max error:", numpy.abs(u_e_array - u_array).max()
```

The value of the error should be at the level of the machine precision (10^{-16}).

To demonstrate the use of point evaluations of `Function` objects, we write out the computed `u` at the center point of the domain and compare it with the exact solution:

Python code

```
center = (0.5, 0.5)
print "numerical u at the center point:",  u(center)
print "exact     u at the center point:", u0(center)
```

Trying a 3×3 mesh, the output from the previous snippet becomes

Output

```
numerical u at the center point: [ 1.83333333]
exact     u at the center point: [ 1.75]
```

The discrepancy is due to the fact that the center point is not a node in this particular mesh, but a point in the interior of a cell, and `u` varies linearly over the cell while `u0` is a quadratic function.

We have seen how to extract the nodal values in a `numpy` array. If desired, we can adjust the nodal values too. Say we want to normalize the solution such that the maximum value is 1. Then we must divide all U_j values by $\max\{U_1, \ldots, U_N\}$. The following snippet performs the task:

Python code

```
max_u = u_array.max()
u_array /= max_u
u.vector()[:] = u_array
u.vector().set_local(u_array)  # alternative
print u.vector().array()
```

That is, we manipulate `u_array` as desired, and then we insert this array into `u`'s `Vector` object. The `/=` operator implies an in-place modification of the object on the left-hand side: all elements of the `u_array` are divided by the value `max_u`. Alternatively, one could write `u_array = u_array/max_u`, which implies creating a new array on the right-hand side and assigning this array to the name `u_array`.

A call like `u.vector().array()` returns a copy of the data in `u.vector()`. One must therefore never perform assignments like `u.vector.array()[:] = ...`, but instead extract the `numpy` array (that is, a copy), manipulate it, and insert it back with `u.vector()[:] =` or `u.set_local(...)`.

All the code in this subsection can be found in the file `d4_p2D.py` in the `stationary/poisson` directory.

1.1.7 Solving a real physical problem

Perhaps you are not particularly amazed by viewing the simple surface of u in the test problem from Section 1.1.3. However, solving a real physical problem with a more interesting and amazing solution

on the screen is only a matter of specifying a more exciting domain, boundary condition, and/or right-hand side f.

One possible physical problem regards the deflection $D(x,y)$ of an elastic circular membrane with radius R, subject to a localized perpendicular pressure force, modeled as a Gaussian function. The appropriate PDE model is

$$-T\Delta D = p(x,y) \quad \text{in } \Omega = \{(x,y) \,|\, x^2 + y^2 \leqslant R\}, \tag{1.15}$$

with

$$p(x,y) = \frac{A}{2\pi\sigma}\exp\left(-\frac{1}{2}\left(\frac{x-x_0}{\sigma}\right)^2 - \frac{1}{2}\left(\frac{y-y_0}{\sigma}\right)^2\right). \tag{1.16}$$

Here, T is the tension in the membrane (constant), p is the external pressure load, A the amplitude of the pressure, (x_0, y_0) the localization of the Gaussian pressure function, and σ the "width" of this function. The boundary of the membrane has no deflection, implying $D = 0$ as boundary condition.

For scaling and verification it is convenient to simplify the problem to find an analytical solution. In the limit $\sigma \to \infty$, $p \to A/(2\pi\sigma)$, which allows us to integrate an axi–symmetric version of the equation in the radial coordinate $r \in [0, R]$ and obtain $D(r) = (r^2 - R^2)A/(8\pi\sigma T)$. This result gives a rough estimate of the characteristic size of the deflection: $|D(0)| = AR^2/(8\pi\sigma T)$, which can be used to scale the deflection. With R as characteristic length scale, we can derive the equivalent dimensionless problem on the unit circle,

$$-\Delta w = f, \tag{1.17}$$

with $w = 0$ on the boundary and with

$$f(x,y) = 4\exp\left(-\frac{1}{2}\left(\frac{Rx-x_0}{\sigma}\right)^2 - \frac{1}{2}\left(\frac{Ry-y_0}{\sigma}\right)^2\right). \tag{1.18}$$

For notational convenience we have dropped introducing new symbols for the scaled coordinates in (1.18). Now D is related to w through $D = AR^2w/(8\pi\sigma T)$.

Let us list the modifications of the `d1_p2D.py` program that are needed to solve this membrane problem:

1. Initialize T, A, R, x_0, y_0, and σ,
2. create a mesh over the unit circle,
3. make an expression object for the scaled pressure function f,
4. define the `a` and `L` formulas in the variational problem for w and compute the solution,
5. plot the mesh, w, and f,
6. write out the maximum real deflection D,

Some suitable values of T, A, R, x_0, y_0, and σ are

Python code

```
T = 10.0  # tension
A = 1.0   # pressure amplitude
R = 0.3   # radius of domain
theta = 0.2
x0 = 0.6*R*cos(theta)
y0 = 0.6*R*sin(theta)
sigma = 0.025
```

A mesh over the unit circle can be created by

Python code

```
mesh = UnitCircle(n)
```

where `n` is the typical number of elements in the radial direction.

The function f is represented by an `Expression` object. There are many physical parameters in the formula for f that enter the expression string and these parameters must have their values set by keyword arguments:

Python code

```
f = Expression("4*exp(-0.5*(pow((R*x[0] - x0)/sigma, 2)) "\
               "     - 0.5*(pow((R*x[1] - y0)/sigma, 2)))",
               R=R, x0=x0, y0=y0, sigma=sigma)
```

The coordinates in `Expression` objects *must* be a vector with indices 0, 1, and 2, and with the name `x`. Otherwise we are free to introduce names of parameters as long as these are given default values by keyword arguments. All the parameters initialized by keyword arguments can at any time have their values modified. For example, we may set

Python code

```
f.sigma = 50
f.x0 = 0.3
```

It would be of interest to visualize f along with w so that we can examine the pressure force and its response. We must then transform the formula (`Expression`) to a finite element function (`Function`). The most natural approach is to construct a finite element function whose degrees of freedom (values at the nodes in this case) are calculated from f. That is, we interpolate f (see Section 1.1.6):

Python code

```
f = interpolate(f, V)
```

Calling `plot(f)` will produce a plot of f. Note that the assignment to `f` destroys the previous `Expression` object `f`, so if it is of interest to still have access to this object another name must be used for the `Function` object returned by `interpolate`.

We need some evidence that the program works, and to this end we may use the analytical solution listed above for the case $\sigma \to \infty$. In scaled coordinates the solution reads

$$w(x, y) = 1 - x^2 - y^2.$$

Practical values for an infinite σ may be 50 or larger, and in such cases the program will report the maximum deviation between the computed w and the (approximate) exact w_e.

Note that the variational formulation remains the same as in the program from Section 1.1.3, except that u is replaced by w and $u_0 = 0$. The final program is found in the file `membrane1.py`, located in the `stationary/poisson` directory, and also listed below. We have inserted capabilities for iterative solution methods and hence large meshes (Section 1.1.4), used objects for the variational problem and solver (Section 1.1.5), and made numerical comparison of the numerical and (approximate) analytical solution (Section 1.1.6).

Python code

```
from dolfin import *

# Set pressure function:
```

```
T = 10.0  # tension
A = 1.0   # pressure amplitude
R = 0.3   # radius of domain
theta = 0.2
x0 = 0.6*R*cos(theta)
y0 = 0.6*R*sin(theta)
sigma = 0.025
#sigma = 50  # large value for verification
n = 40   # approx no of elements in radial direction
mesh = UnitCircle(n)
V = FunctionSpace(mesh, "Lagrange", 1)

# Define boundary condition w=0
def boundary(x, on_boundary):
    return on_boundary

bc = DirichletBC(V, Constant(0.0), boundary)

# Define variational problem
w = TrialFunction(V)
v = TestFunction(V)
a = inner(nabla_grad(w), nabla_grad(v))*dx
f = Expression("4*exp(-0.5*(pow((R*x[0] - x0)/sigma, 2)) "\
               "     - 0.5*(pow((R*x[1] - y0)/sigma, 2)))",
               R=R, x0=x0, y0=y0, sigma=sigma)
L = f*v*dx

# Compute solution
w = Function(V)
problem = LinearVariationalProblem(a, L, w, bc)
solver  = LinearVariationalSolver(problem)
solver.parameters["linear_solver"] = "cg"
solver.parameters["preconditioner"] = "ilu"
solver.solve()

# Plot scaled solution, mesh and pressure
plot(mesh, title="Mesh over scaled domain")
plot(w, title="Scaled deflection")
f = interpolate(f, V)
plot(f, title="Scaled pressure")

# Find maximum real deflection
max_w = w.vector().array().max()
max_D = A*max_w/(8*pi*sigma*T)
print "Maximum real deflection is", max_D

# Verification for "flat" pressure (large sigma)
if sigma >= 50:
    w_exact = Expression("1 - x[0]*x[0] - x[1]*x[1]")
    w_e = interpolate(w_exact, V)
    dev = numpy.abs(w_e.vector().array() - w.vector().array()).max()
    print 'sigma=%g: max deviation=%e' % dev

# Should be at the end
interactive()
```

Choosing a small width σ (say 0.01) and a location (x_0, y_0) toward the circular boundary (say $(0.6R\cos\theta, 0.6R\sin\theta)$ for any $\theta \in [0, 2\pi]$), may produce an exciting visual comparison of w and f that demonstrates the very smoothed elastic response to a peak force (or mathematically, the smoothing properties of the inverse of the Laplace operator). One needs to experiment with the mesh resolution

to get a smooth visual representation of f. You are strongly encouraged to play around with the plots and different mesh resolutions.

1.1.8 Quick visualization with VTK

As we go along with examples it is fun to play around with `plot` commands and visualize what is computed. This section explains some useful visualization features.

The `plot(u)` command launches a FEniCS component called Viper, which applies the VTK package to visualize finite element functions. Viper is not a full-fledged, easy-to-use front-end to VTK (like Mayavi2, ParaView, or VisIt), but rather a thin layer on top of VTK's Python interface, allowing us to quickly visualize a DOLFIN function or mesh, or data in plain Numerical Python arrays, within a Python program. Viper is ideal for debugging, teaching, and initial scientific investigations. The visualization can be interactive, or you can steer and automate it through program statements. More advanced and professional visualizations are usually better done with advanced tools like MayaVi2, ParaView, or VisIt.

We have made a program `membrane1v.py` for the membrane deflection problem in Section 1.1.7 and added various demonstrations of Viper capabilities. You are encouraged to play around with `membrane1v.py` and modify the code as you read about various features.

The `plot` function can take additional arguments, such as a title of the plot, or a specification of a wireframe plot (elevated mesh) instead of a colored surface plot:

Python code

```
plot(mesh, title="Finite element mesh")
plot(w, wireframe=True, title="solution")
```

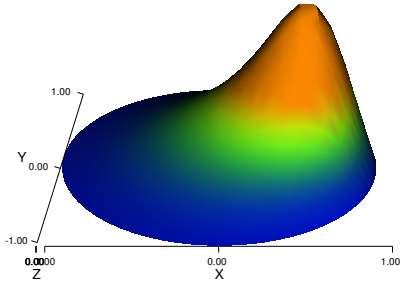
The three mouse buttons can be used to rotate, translate, and zoom the surface. Pressing `h` in the plot window makes a printout of several key bindings that are available in such windows. For example, pressing `m` in the mesh plot window dumps the plot of the mesh to an Encapsulated PostScript (`.eps`) file, while pressing `i` saves the plot in PNG format. All file names are automatically generated as `simulationX.eps`, where `X` is a counter `0000`, `0001`, `0002`, etc., being increased every time a new plot file in that format is generated (the extension of PNG files is `.png` instead of `.eps`). Pressing `o` adds a red outline of a bounding box around the domain.

One can alternatively control the visualization from the program code directly. This is done through a `Viper` object returned from the `plot` command. Let us grab this object and use it to 1) tilt the camera -65 degrees in the latitude direction, 2) add x and y axes, 3) change the default name of the plot files, 4) change the color scale, and 5) write the plot to a PNG and an EPS file. Here is the code:

Python code

```
viz_w = plot(w,
             wireframe=False,
             title="Scaled membrane deflection",
             rescale=False,
             axes=True,              # include axes
             basename="deflection",  # default plotfile name
             )

viz_w.elevate(-65) # tilt camera -65 degrees (latitude dir)
viz_w.set_min_max(0, 0.5*max_w)  # color scale
viz_w.update(w)    # bring settings above into action
viz_w.write_png("deflection.png")
viz_w.write_ps("deflection", format="eps")
```

Figure 1.3: Plot of the deflection of a membrane.



The `format` argument in the latter line can also take the values `"ps"` for a standard PostScript file and `"pdf"` for a PDF file. Note the necessity of the `viz_w.update(w)` call – without it we will not see the effects of tilting the camera and changing the color scale. Figure 1.3 shows the resulting scalar surface.

1.1.9 Computing derivatives

In Poisson and many other problems the gradient of the solution is of interest. The computation is in principle simple: since $u = \sum_{j=1}^{N} U_j \phi_j$, we have that

$$\nabla u = \sum_{j=1}^{N} U_j \nabla \phi_j. \tag{1.19}$$

Given the solution variable `u` in the program, its gradient is obtained by `grad(u)` or `nabla_grad(u)`. However, the gradient of a piecewise continuous finite element scalar field is a discontinuous vector field since the ϕ_j has discontinuous derivatives at the boundaries of the cells. For example, using Lagrange elements of degree 1, u is linear over each cell, and the numerical ∇u becomes a piecewise constant vector field. On the contrary, the exact gradient is continuous. For visualization and data analysis purposes we often want the computed gradient to be a continuous vector field. Typically, we want each component of ∇u to be represented in the same way as u itself. To this end, we can project the components of ∇u onto the same function space as we used for u. This means that we solve $w = \nabla u$ approximately by a finite element method. This process is known as *projection*. Looking at the component $\partial u/\partial x$ of the gradient, we project the (discrete) derivative $\sum_j U_j \partial \phi_j/\partial x$ onto a function space with basis $\phi_1, \phi_2, \ldots$ such that the derivative in this space is expressed by the standard sum $\sum_j \bar{U}_j \phi_j$, for suitable (new) coefficients $\bar{U}_j$.

The variational problem for w reads: find $w \in V^{(\mathrm{g})}$ such that

$$a(w, v) = L(v) \quad \forall\, v \in \hat{V}^{(\mathrm{g})}, \tag{1.20}$$

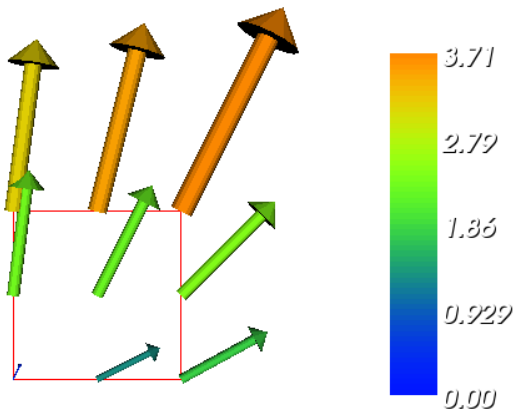


Figure 1.4: Example of visualizing the vector field ∇u by arrows at the nodes.

where

$$a(w, v) = \int_\Omega w \cdot v \, \mathrm{d}x, \tag{1.21}$$

$$L(v) = \int_\Omega \nabla u \cdot v \, \mathrm{d}x. \tag{1.22}$$

The function spaces $V^{(\mathrm{g})}$ and $\hat{V}^{(\mathrm{g})}$ (with the superscript g denoting "gradient") are vector versions of the function space for u, with boundary conditions removed (if V is the space we used for u, with no restrictions on boundary values, $V^{(\mathrm{g})} = \hat{V}^{(\mathrm{g})} = [V]^d$, where d is the number of space dimensions). For example, if we used piecewise linear functions on the mesh to approximate u, the variational problem for w corresponds to approximating each component field of w by piecewise linear functions.

The variational problem for the vector field w, called `grad_u` in the code, is easy to solve in FEniCS:

Python code

```
V_g = VectorFunctionSpace(mesh, "Lagrange", 1)
w = TrialFunction(V_g)
v = TestFunction(V_g)

a = inner(w, v)*dx
L = inner(grad(u), v)*dx
grad_u = Function(V_g)
solve(a == L, grad_u)

plot(grad_u, title="grad(u)")
```

The boundary condition argument to `solve` is dropped since there are no essential boundary conditions in this problem. The new thing is basically that we work with a `VectorFunctionSpace`, since the unknown is now a vector field, instead of the `FunctionSpace` object for scalar fields. Figure 1.4 shows an example of how Viper can visualize such a vector field.

The scalar component fields of the gradient can be extracted as separate fields and, e.g., visualized:

Python code

```
grad_u_x, grad_u_y = grad_u.split(deepcopy=True)  # extract components
plot(grad_u_x, title="x-component of grad(u)")
plot(grad_u_y, title="y-component of grad(u)")
```

The `deepcopy=True` argument signifies a *deep copy*, which is a general term in computer science implying that a copy of the data is returned. (The opposite, `deepcopy=False`, means a *shallow copy*, where the returned objects are just pointers to the original data.)

The `grad_u_x` and `grad_u_y` variables behave as `Function` objects. In particular, we can extract the underlying arrays of nodal values by

Python code

```
grad_u_x_array = grad_u_x.vector().array()
grad_u_y_array = grad_u_y.vector().array()
```

The degrees of freedom of the `grad_u` vector field can also be reached by

Python code

```
grad_u_array = grad_u.vector().array()
```

but this is a flat `numpy` array where the degrees of freedom for the x component of the gradient is stored in the first part, then the degrees of freedom of the y component, and so on.

The program `d5_p2D.py` extends the code `d4_p2D.py` from Section 1.1.6 with computations and visualizations of the gradient. Examining the arrays `grad_u_x_array` and `grad_u_y_array`, or looking at the plots of `grad_u_x` and `grad_u_y`, quickly reveals that the computed `grad_u` field does not equal the exact gradient $(2x, 4y)$ in this particular test problem where $u = 1 + x^2 + 2y^2$. There are inaccuracies at the boundaries, arising from the approximation problem for w. Increasing the mesh resolution shows, however, that the components of the gradient vary linearly as $2x$ and $4y$ in the interior of the mesh (as soon as we are one element away from the boundary). See Section 1.1.8 for illustrations of this phenomenon.

Projecting some function onto some space is a very common operation in finite element programs. The manual steps in this process have therefore been collected in a utility function `project(q, W)`, which returns the projection of some `Function` or `Expression` object named `q` onto the `FunctionSpace` or `VectorFunctionSpace` named `W`. Specifically, the previous code for projecting each component of `grad(u)` onto the same space that we use for `u`, can now be done by a one–line call:

Python code

```
grad_u = project(grad(u), VectorFunctionSpace(mesh, "Lagrange", 1))
```

The applications of projection are many, including turning discontinuous gradient fields into continuous ones, comparing higher- and lower-order function approximations, and transforming a higher-order finite element solution down to a piecewise linear field, which is required by many visualization packages.

1.1.10 A variable-coefficient Poisson problem

Suppose we have a variable coefficient $p(x, y)$ in the Laplace operator, as in the boundary-value problem

$$\begin{aligned} -\nabla \cdot [p(x,y)\nabla u(x,y)] &= f(x,y) \quad \text{in } \Omega, \\ u(x,y) &= u_0(x,y) \quad \text{on } \partial\Omega. \end{aligned} \tag{1.23}$$

We shall quickly demonstrate that this simple extension of our model problem only requires an equally simple extension of the FEniCS program.

Let us continue to use our favorite solution $u(x, y) = 1 + x^2 + 2y^2$ and then prescribe $p(x, y) = x + y$. It follows that $u_0(x, y) = 1 + x^2 + 2y^2$ and $f(x, y) = -8x - 10y$.

What are the modifications we need to do in the `d4_p2D.py` program from Section 1.1.6?

1. `f` must be an `Expression` since it is no longer a constant,
2. a new `Expression` `p` must be defined for the variable coefficient,
3. the variational problem is slightly changed.

First we address the modified variational problem. Multiplying the PDE in (1.23) and integrating by parts now results in

$$\int_\Omega p\nabla u\cdot\nabla v\,\mathrm{d}x - \int_{\partial\Omega} p\frac{\partial u}{\partial n}v\,\mathrm{d}s = \int_\Omega fv\,\mathrm{d}x. \tag{1.24}$$

The function spaces for u and v are the same as in Section 1.1.2, implying that the boundary integral vanishes since $v = 0$ on $\partial\Omega$ where we have Dirichlet conditions. The weak form $a(u,v) = L(v)$ then has

$$a(u,v) = \int_\Omega p\nabla u\cdot\nabla v\,\mathrm{d}x, \tag{1.25}$$

$$L(v) = \int_\Omega fv\,\mathrm{d}x. \tag{1.26}$$

In the code from Section 1.1.3 we must replace

Python code

```
a = inner(nabla_grad(u), nabla_grad(v))*dx
```

by

Python code

```
a = p*inner(nabla_grad(u), nabla_grad(v))*dx
```

The definitions of `p` and `f` read

Python code

```
p = Expression("x[0] + x[1]")
f = Expression("-8*x[0] - 10*x[1]")
```

No additional modifications are necessary. The complete code can be found in in the file `vcp2D.py` (variable-coefficient Poisson problem in 2D). You can run it and confirm that it recovers the exact u at the nodes.

The flux $-p\nabla u$ may be of particular interest in variable-coefficient Poisson problems as it often has an interesting physical significance. As explained in Section 1.1.9, we normally want the piecewise discontinuous flux or gradient to be approximated by a continuous vector field, using the same elements as used for the numerical solution u. The approximation now consists of solving $w = -p\nabla u$ by a finite element method: find $w \in V^{(\mathrm{g})}$ such that

$$a(w,v) = L(v) \quad \forall\, v \in \hat{V}^{(\mathrm{g})}, \tag{1.27}$$

where

$$a(w,v) = \int_\Omega w\cdot v\,\mathrm{d}x, \tag{1.28}$$

$$L(v) = \int_\Omega (-p\nabla u)\cdot v\,\mathrm{d}x. \tag{1.29}$$

This problem is identical to the one in Section 1.1.9, except that p enters the integral in L.

The relevant Python statements for computing the flux field take the form

Python code

```
V_g = VectorFunctionSpace(mesh, "Lagrange", 1)
w = TrialFunction(V_g)
v = TestFunction(V_g)

a = inner(w, v)*dx
L = inner(-p*grad(u), v)*dx
flux = Function(V_g)
solve(a == L, flux)
```

The following call to `project` is equivalent to the above statements:

Python code

```
flux = project(-p*nabla_grad(u),
               VectorFunctionSpace(mesh, "Lagrange", 1))
```

Plotting the flux vector field is naturally as easy as plotting the gradient in Section 1.1.9:

Python code

```
plot(flux, title="flux field")

flux_x, flux_y = flux.split(deepcopy=True)  # extract components
plot(flux_x, title="x-component of flux (-p*grad(u))")
plot(flux_y, title="y-component of flux (-p*grad(u))")
```

For data analysis of the nodal values of the flux field we can grab the underlying `numpy` arrays:

Python code

```
flux_x_array = flux_x.vector().array()
flux_y_array = flux_y.vector().array()
```

The program `vcp2D.py` contains in addition some plots, including a curve plot comparing `flux_x` and the exact counterpart along the line $y = 1/2$. The associated programming details related to this visualization are explained in Section 1.1.12.

1.1.11 Computing functionals

After the solution u of a PDE is computed, we occasionally want to compute functionals of u, for example,

$$\frac{1}{2}||\nabla u||^2 \equiv \frac{1}{2}\int_\Omega \nabla u \cdot \nabla u \,\mathrm{d}x, \tag{1.30}$$

which often reflects some energy quantity. Another frequently occurring functional is the error

$$||u_\mathrm{e} - u|| = \left(\int_\Omega (u_\mathrm{e} - u)^2 \,\mathrm{d}x\right)^{1/2}, \tag{1.31}$$

where u_e is the exact solution. The error is of particular interest when studying convergence properties. Sometimes the interest concerns the flux out of a part Γ of the boundary $\partial\Omega$,

$$F = -\int_\Gamma p\nabla u \cdot \,\mathrm{d}s, \tag{1.32}$$

where n is an outward unit normal at Γ and p is a coefficient (see the problem in Section 1.1.10 for a specific example). All these functionals are easy to compute with FEniCS, and this section describes how it can be done.

Energy functional. The integrand of the energy functional (1.30) is described in the UFL language in the same manner as we describe weak forms:

Python code

```
energy = 0.5*inner(grad(u), grad(u))*dx
E = assemble(energy)
```

The `assemble` call performs the integration. It is possible to restrict the integration to subdomains, or parts of the boundary, by using a mesh function to mark the subdomains (this technique will be explained in Section 1.5.3). The program `membrane2.py` carries out the computation of the elastic energy

$$\frac{1}{2}||T\nabla D||^2 = \frac{1}{2}\left(\frac{AR}{8\pi\sigma}\right)^2 ||\nabla w||^2 \tag{1.33}$$

in the membrane problem from Section 1.1.7.

Convergence estimation. To illustrate error computations and convergence of finite element solutions, we modify the `d5_p2D.py` program from Section 1.1.9 and specify a more complicated solution,

$$u(x,y) = \sin(\omega\pi x)\sin(\omega\pi y) \tag{1.34}$$

on the unit square. This choice implies $f(x,y) = 2\omega^2\pi^2 u(x,y)$. With ω restricted to an integer it follows that $u_0 = 0$. We must define the appropriate boundary conditions, the exact solution, and the f function in the code:

Python code

```
def boundary(x, on_boundary):
    return on_boundary

bc = DirichletBC(V, Constant(0.0), boundary)

omega = 1.0
u_e = Expression("sin(omega*pi*x[0])*sin(omega*pi*x[1])",
                 omega=omega)

f = 2*pi**2*omega**2*u_e
```

The computation of (1.31) can be done by

Python code

```
error = (u - u_e)**2*dx
E = sqrt(assemble(error))
```

Here, `u_e` will be interpolated onto the function space `V`. This implies that the exact solution used in the integral will vary linearly over the cells, and not as a sine function, if `V` corresponds to linear Lagrange elements. This situation may yield a smaller error `u - u_e` than what is actually true.

More accurate representation of the exact solution is easily achieved by interpolating the formula onto a space defined by higher-order elements, say of third degree:

Python code

```
Ve = FunctionSpace(mesh, "Lagrange", degree=3)
u_e_Ve = interpolate(u_e, Ve)
error = (u - u_e_Ve)**2*dx
E = sqrt(assemble(error))
```

To achieve complete mathematical control of which function space the computations are carried out in, we can explicitly interpolate u too:

Python code

```
u_Ve = interpolate(u, Ve)
error = (u_Ve - u_e_Ve)**2*dx
```

The square in the expression for `error` will be expanded and lead to a lot of terms that almost cancel when the error is small, with the potential of introducing significant round-off errors. The function `errornorm` is available for avoiding this effect by first interpolating `u` and `u_e` to a space with higher-order elements, then subtracting the degrees of freedom, and then performing the integration of the error field. The usage is simple:

Python code

```
E = errornorm(u_e, u, normtype="L2", degree=3)
```

It is illustrative to look at the short implementation of `errornorm`:

Python code

```
def errornorm(u_e, u, Ve):
    u_Ve = interpolate(u, Ve)
    u_e_Ve = interpolate(u_e, Ve)
    e_Ve = Function(Ve)
    # Subtract degrees of freedom for the error field
    e_Ve.vector()[:] = u_e_Ve.vector().array() - \
                       u_Ve.vector().array()
    error = e_Ve**2*dx
    return sqrt(assemble(error))
```

The `errornorm` procedure turns out to be identical to computing the expression `(u_e - u)**2*dx` directly in the present test case.

Sometimes it is of interest to compute the error of the gradient field: $||\nabla(u - u_e)||$ (often referred to as the H^1 seminorm of the error). Given the error field `e_Ve` above, we simply write

Python code

```
H1seminorm = sqrt(assemble(inner(grad(e_Ve), grad(e_Ve))*dx))
```

Finally, we remove all `plot` calls and printouts of u values in the original program, and collect the computations in a function:

Python code

```
def compute(nx, ny, degree):
    mesh = UnitSquare(nx, ny)
    V = FunctionSpace(mesh, "Lagrange", degree=degree)
    ...
    Ve = FunctionSpace(mesh, "Lagrange", degree=5)
    E = errornorm(u_e, u, Ve)
    return E
```

Calling `compute` for finer and finer meshes enables us to study the convergence rate. Define the element size $h = 1/n$, where n is the number of divisions in x and y direction (`nx=ny` in the code). We perform experiments with $h_0 > h_1 > h_2 \cdots$ and compute the corresponding errors E_0, E_1, E_3 and so forth. Assuming $E_i = Ch_i^r$ for unknown constants C and r, we can compare two consecutive

experiments, $E_i = Ch_i^r$ and $E_{i-1} = Ch_{i-1}^r$, and solve for r:

$$r = \frac{\ln(E_i/E_{i-1})}{\ln(h_i/h_{i-1})}. \tag{1.35}$$

The r values should approach the expected convergence rate `degree+1` as i increases.

The procedure above can easily be turned into Python code:

Python code

```
import sys
degree = int(sys.argv[1])  # read degree as 1st command-line arg
h = []  # element sizes
E = []  # errors
for nx in [4, 8, 16, 32, 64, 128, 264]:
    h.append(1.0/nx)
    E.append(compute(nx, nx, degree))

# Convergence rates
from math import log as ln  # (log is a dolfin name too)
for i in range(1, len(E)):
    r = ln(E[i]/E[i-1])/ln(h[i]/h[i-1])
    print "h=%10.2E r=%.2f" % (h[i], r)
```

The resulting program has the name `d6_p2D.py` and computes error norms in various ways. Running this program for elements of first degree and $\omega = 1$ yields the output

Output

```
h=1.25E-01 E=3.25E-02 r=1.83
h=6.25E-02 E=8.37E-03 r=1.96
h=3.12E-02 E=2.11E-03 r=1.99
h=1.56E-02 E=5.29E-04 r=2.00
h=7.81E-03 E=1.32E-04 r=2.00
h=3.79E-03 E=3.11E-05 r=2.00
```

That is, we approach the expected second-order convergence of linear Lagrange elements as the meshes become sufficiently fine.

Running the program for second-degree elements results in the expected value $r = 3$,

Output

```
h=1.25E-01 E=5.66E-04 r=3.09
h=6.25E-02 E=6.93E-05 r=3.03
h=3.12E-02 E=8.62E-06 r=3.01
h=1.56E-02 E=1.08E-06 r=3.00
h=7.81E-03 E=1.34E-07 r=3.00
h=3.79E-03 E=1.53E-08 r=3.00
```

However, using `(u - u_e)**2` for the error computation, which implies interpolating `u_e` onto the same space as `u`, results in $r = 4$ (!). This is an example where it is important to interpolate `u_e` to a higher-order space (polynomials of degree 3 are sufficient here) to avoid computing a too optimistic convergence rate.

Running the program for third-degree elements results in the expected value $r = 4$:

Output

```
h=1.25E-01 r=4.09
h=6.25E-02 r=4.03
h=3.12E-02 r=4.01
h=1.56E-02 r=4.00
h=7.81E-03 r=4.00
```

Checking convergence rates is the next best method for verifying PDE codes (the best being exact recovery of a solution as in Section 1.1.6 and many other places in this tutorial).

Flux functionals. To compute flux integrals like (1.32) we need to define the n vector, referred to as *facet normal* in FEniCS. If Γ is the complete boundary we can perform the flux computation by

Python code

```
n = FacetNormal(mesh)
flux = -p*dot(nabla_grad(u), n)*ds
total_flux = assemble(flux)
```

Although `nabla_grad(u)` and `grad(u)` are interchangeable in the above expression when `u` is a scalar function, we have chosen to write `nabla_grad(u)` because this is the right expression if we generalize the underlying equation to a vector Laplace/Poisson PDE. With `grad(u)` we must in that case write `dot(n, grad(u))`.

It is possible to restrict the integration to a part of the boundary using a mesh function to mark the relevant part, as explained in Section 1.5.3. Assuming that the part corresponds to subdomain number `i`, the relevant form for the flux is `-p*dot(nabla_grad(u), n)*ds(i)`.

1.1.12 *Visualization of structured mesh data*

When finite element computations are done on a structured rectangular mesh, maybe with uniform partitioning, VTK-based tools for completely unstructured 2D/3D meshes are not required. Instead we can use visualization and data analysis tools for *structured data*. Such data typically appear in finite difference simulations and image analysis. Analysis and visualization of structured data are faster and easier than doing the same with data on unstructured meshes, and the collection of tools to choose among is much larger. We shall demonstrate the potential of such tools and how they allow for tailored and flexible visualization and data analysis.

A necessary first step is to transform our `mesh` object to an object representing a rectangle with equally-shaped *rectangular* cells. The Python package `scitools` has this type of structure, called a `UniformBoxGrid`. The second step is to transform the one-dimensional array of nodal values to a two-dimensional array holding the values at the corners of the cells in the structured grid. In such grids, we want to access a value by its i and j indices, i counting cells in the x direction, and j counting cells in the y direction. This transformation is in principle straightforward, yet it frequently leads to obscure indexing errors. The `BoxField` object in `scitools` takes conveniently care of the details of the transformation. With a `BoxField` defined on a `UniformBoxGrid` it is very easy to call up more standard plotting packages to visualize the solution along lines in the domain or as 2D contours or lifted surfaces.

Let us go back to the `vcp2D.py` code from Section 1.1.10 and map `u` onto a `BoxField` object:

Python code

```
import scitools.BoxField
u2 = u if u.ufl_element().degree() == 1 else \
     interpolate(u, FunctionSpace(mesh, "Lagrange", 1))
u_box = scitools.BoxField.dolfin_function2BoxField(
        u2, mesh, (nx,ny), uniform_mesh=True)
```

The function `dolfin_function2BoxField` can only work with finite element fields with *linear* (degree 1) elements, so for higher-degree elements we here simply interpolate the solution onto a mesh with linear elements. We could also interpolate/project onto a finer mesh in the higher-degree case. Such transformations to linear finite element fields are very often needed when calling up plotting packages

or data analysis tools. The `u.ufl_element()` method returns an object holding the element type, and this object has a method `degree()` for returning the element degree as an integer. The parameters `nx` and `ny` are the number of divisions in each space direction that were used when calling `UnitSquare` to make the `mesh` object. The result `u_box` is a `BoxField` object that supports "finite difference" indexing and an underlying grid suitable for `numpy` operations on 2D data. Also 1D and 3D meshes (with linear elements) can be turned into `BoxField` objects.

The ability to access a finite element field in the way one can access a finite difference-type of field is handy in many occasions, including visualization and data analysis. Here is an example of writing out the coordinates and the field value at a grid point with indices `i` and `j` (going from 0 to `nx` and `ny`, respectively, from lower left to upper right corner):

Python code

```
X = 0; Y = 1; Z = 0  # convenient indices

i = nx; j = ny   # upper right corner
print "u(%g,%g)=%g" % (u_box.grid.coor[X][i],
                      u_box.grid.coor[Y][j],
                      u_box.values[i,j])
```

For instance, the x coordinates are reached by `u_box.grid.coor[X]`. The `grid` attribute is an instance of class `UniformBoxGrid`.

Many plotting programs can be used to visualize the data in `u_box`. Matplotlib is now a very popular plotting program in the Python world and could be used to make contour plots of `u_box`. However, other programs like Gnuplot, VTK, and MATLAB have better support for surface plots at the time of this writing. Our choice in this tutorial is to use the Python package `scitools.easyviz`, which offers a uniform MATLAB-like syntax as interface to various plotting packages such as Gnuplot, matplotlib, VTK, OpenDX, MATLAB, and others. With `scitools.easyviz` we write one set of statements, close to what one would do in MATLAB or Octave, and then it is easy to switch between different plotting programs, at a later stage, through a command-line option, a line in a configuration file, or an import statement in the program.

A contour plot is made by the following `scitools.easyviz` command:

Python code

```
import scitools.easyviz as ev
ev.contour(u_box.grid.coorv[X], u_box.grid.coorv[Y], u_box.values,
           5, clabels="on")
ev.title("Contour plot of u")
ev.savefig("u_contours.eps")

# or more compact syntax:
ev.contour(u_box.grid.coorv[X], u_box.grid.coorv[Y], u_box.values,
           5, clabels="on",
           savefig="u_contours.eps", title="Contour plot of u")
```

The resulting plot can be viewed in Figure 1.5a. The `contour` function needs arrays with the x and y coordinates expanded to 2D arrays (in the same way as demanded when making vectorized `numpy` calculations of arithmetic expressions over all grid points). The correctly expanded arrays are stored in `grid.coorv`. The above call to `contour` creates 5 equally spaced contour lines, and with `clabels="on"` the contour values can be seen in the plot.

Other functions for visualizing 2D scalar fields are `surf` and `mesh` as known from MATLAB:

Python code

```
import scitools.easyviz as ev
ev.figure()
ev.surf(u_box.grid.coorv[X], u_box.grid.coorv[Y], u_box.values,
        shading="interp", colorbar="on",
        title="surf plot of u", savefig="u_surf.eps")

ev.figure()
ev.mesh(u_box.grid.coorv[X], u_box.grid.coorv[Y], u_box.values,
        title="mesh plot of u", savefig="u_mesh.eps")
```

Figure 1.6 exemplifies the surfaces arising from the two plotting commands above. You can type `pydoc scitools.easyviz` in a terminal window to get a full tutorial. Note that `scitools.easyviz` offers function names like `plot` and `mesh`, which clash with `plot` from `dolfin` and the `mesh` variable in our programs. Therefore, we recommend the `ev` prefix.

A handy feature of `BoxField` is the ability to give a start point in the grid and a direction, and then extract the field and corresponding coordinates along the nearest grid line. In 3D fields one can also extract data in a plane. Say we want to plot u along the line $y = 1/2$ in the grid. The grid points, `x`, and the u values along this line, `uval`, are extracted by

Python code

```
start = (0, 0.5)
x, uval, y_fixed, snapped = u_box.gridline(start, direction=X)
```

The variable `snapped` is true if the line had to be snapped onto a grid line and in that case `y_fixed` holds the snapped (altered) y value. Plotting u versus the x coordinate along this line, using `scitools.easyviz`, is now a matter of

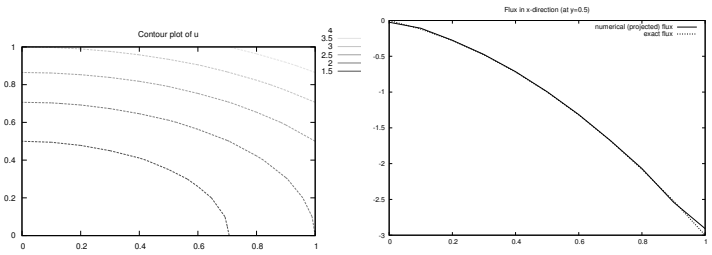
Python code

```
ev.figure()  # new plot window
ev.plot(x, uval, "r-")  # "r--: red solid line
ev.title("Solution")
ev.legend("finite element solution")

# or more compactly:
ev.plot(x, uval, "r-", title="Solution",
        legend="finite element solution")
```

A more exciting plot compares the projected numerical flux in x direction along the line $y = 1/2$ with the exact flux:

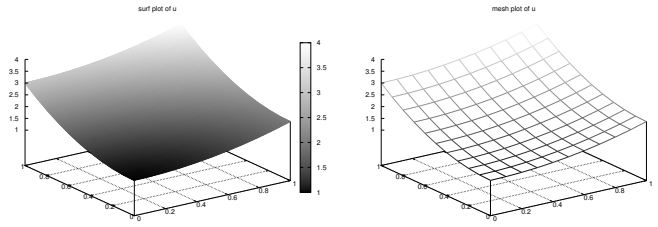
Python code

```
ev.figure()
flux2_x = flux_x if flux_x.ufl_element().degree() == 1 else \
          interpolate(flux_x, FunctionSpace(mesh, "Lagrange", 1))
flux_x_box = scitools.BoxField.dolfin_function2BoxField(
          flux2_x, mesh, (nx,ny), uniform_mesh=True)
x, fluxval, y_fixed, snapped = \
          flux_x_box.gridline(start, direction=X)
y = y_fixed
flux_x_exact = -(x + y)*2*x
ev.plot(x, fluxval, "r-",
        x, flux_x_exact, "b-",
        legend=("numerical (projected) flux", "exact flux"),
        title="Flux in x-direction (at y=%g)" % y_fixed,
        savefig="flux.eps")
```



(a) (b)

Figure 1.5: Examples of plots created by transforming the finite element field to a field on a uniform, structured 2D grid: (a) contour plot of the solution; (b) curve plot of the exact flux $-p\partial u/\partial x$ against the corresponding projected numerical flux.



(a) (b)

Figure 1.6: Examples of plots created by transforming the finite element field to a field on a uniform, structured 2D grid: (a) a surface plot of the solution; (b) lifted mesh plot of the solution.

As seen from Figure 1.5b, the numerical flux is accurate except in the elements closest to the boundaries.

The visualization constructions shown above and used to generate the figures are found in the program `vcp2D.py` in the `stationary/poisson` directory.

It should be easy with the information above to transform a finite element field over a uniform rectangular or box-shaped mesh to the corresponding `BoxField` object and perform MATLAB-style visualizations of the whole field or the field over planes or along lines through the domain. By the transformation to a regular grid we have some more flexibility than what Viper offers. However, we must remark that comprehensive tools like VisIt, MayaVi2, or ParaView also have the possibility for plotting fields along lines and extracting planes in 3D geometries, though usually with less degree of control compared to Gnuplot, MATLAB, and matplotlib.

1.1.13 Combining Dirichlet and Neumann conditions

Let us make a slight extension of our two-dimensional Poisson problem from Section 1.1.1 and add a Neumann boundary condition. The domain is still the unit square, but now we set the Dirichlet condition $u = u_0$ at the left and right sides, $x = 0$ and $x = 1$, while the Neumann condition

$$-\frac{\partial u}{\partial n} = g \tag{1.36}$$

is applied to the remaining sides $y = 0$ and $y = 1$. The Neumann condition is also known as a *natural boundary condition* (in contrast to an essential boundary condition).

Let Γ_{D} and Γ_{N} denote the parts of $\partial\Omega$ where the Dirichlet and Neumann conditions apply,

respectively. The complete boundary-value problem can be written as

$$-\Delta u = f \text{ in } \Omega, \tag{1.37}$$

$$u = u_0 \text{ on } \Gamma_{\mathrm{D}}, \tag{1.38}$$

$$-\frac{\partial u}{\partial n} = g \text{ on } \Gamma_{\mathrm{N}}. \tag{1.39}$$

Again we choose $u = 1 + x^2 + 2y^2$ as the exact solution and adjust f, g, and u_0 accordingly:

$$f = -6, \tag{1.40}$$

$$g = \begin{cases} -4, & y = 1 \\ 0, & y = 0 \end{cases} \tag{1.41}$$

$$u_0 = 1 + x^2 + 2y^2. \tag{1.42}$$

For ease of programming we may introduce a g function defined over the whole of Ω such that g takes on the right values at $y = 0$ and $y = 1$. One possible extension is

$$g(x, y) = -4y. \tag{1.43}$$

The first task is to derive the variational problem. This time we cannot omit the boundary term arising from the integration by parts, because v is only zero on Γ_{D}. We have

$$-\int_\Omega (\Delta u)v \,\mathrm{d}x = \int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x - \int_{\partial\Omega} \frac{\partial u}{\partial n} v \,\mathrm{d}s, \tag{1.44}$$

and since $v = 0$ on Γ_{D},

$$-\int_{\partial\Omega} \frac{\partial u}{\partial n} v \,\mathrm{d}s = -\int_{\Gamma_{\mathrm{N}}} \frac{\partial u}{\partial n} v \,\mathrm{d}s = \int_{\Gamma_{\mathrm{N}}} gv \,\mathrm{d}s, \tag{1.45}$$

by applying the boundary condition on Γ_{N}. The resulting weak form reads

$$\int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x + \int_{\Gamma_{\mathrm{N}}} gv \,\mathrm{d}s = \int_\Omega fv \,\mathrm{d}x. \tag{1.46}$$

Expressing (1.46) in the standard notation $a(u, v) = L(v)$ is straightforward with

$$a(u, v) = \int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x, \tag{1.47}$$

$$L(v) = \int_\Omega fv \,\mathrm{d}x - \int_{\Gamma_{\mathrm{N}}} gv \,\mathrm{d}s. \tag{1.48}$$

How does the Neumann condition impact the implementation? The code in the file `d4_p2D.py` in the directory `stationary/poisson` remains almost the same. Only two adjustments are necessary:

1. The function describing the boundary where Dirichlet conditions apply must be modified.
2. The new boundary term must be added to the expression in `L`.

Step 1 can be coded as

Python code

```
def Dirichlet_boundary(x, on_boundary):
    if on_boundary:
```

```
        if x[0] == 0 or x[0] == 1:
            return True
        else:
            return False
    else:
        return False
```

A more compact implementation reads

Python code

```
def Dirichlet_boundary(x, on_boundary):
    return on_boundary and (x[0] == 0 or x[0] == 1)
```

As pointed out already in Section 1.1.3, testing for an exact match of real numbers is not good programming practice so we introduce a tolerance in the test:

Python code

```
def Dirichlet_boundary(x, on_boundary):
    tol = 1E-14   # tolerance for coordinate comparisons
    return on_boundary and \
           (abs(x[0]) < tol or abs(x[0] - 1) < tol)
```

The second adjustment of our program concerns the definition of `L`, where we have to add a boundary integral and a definition of the g function to be integrated:

Python code

```
g = Expression("-4*x[1]")
L = f*v*dx - g*v*ds
```

The `ds` variable implies a boundary integral, while `dx` implies an integral over the domain Ω. No more modifications are necessary.

The file `dn1_p2D.py` in the `stationary/poisson` directory implements this problem. Running the program verifies the implementation: u equals the exact solution at all the nodes, regardless of how many elements we use.

1.1.14 Multiple Dirichlet conditions

The PDE problem from the previous section applies a function $u_0(x, y)$ for setting Dirichlet conditions at two parts of the boundary. Having a single function to set multiple Dirichlet conditions is seldom possible. The more general case is to have m functions for setting Dirichlet conditions on m parts of the boundary. The purpose of this section is to explain how such multiple conditions are treated in FEniCS programs.

Let us return to the case from Section 1.1.13 and define two separate functions for the two Dirichlet conditions:

$$-\Delta u = -6 \text{ in } \Omega, \tag{1.49}$$

$$u = u_L \text{ on } \Gamma_0, \tag{1.50}$$

$$u = u_R \text{ on } \Gamma_1, \tag{1.51}$$

$$-\frac{\partial u}{\partial n} = g \text{ on } \Gamma_{\mathrm{N}}. \tag{1.52}$$

Here, Γ_0 is the boundary $x = 0$, while Γ_1 corresponds to the boundary $x = 1$. We have that $u_L = 1 + 2y^2$, $u_R = 2 + 2y^2$, and $g = -4y$. For the left boundary Γ_0 we define the usual triple of a

function for the boundary value, a function for defining the boundary of interest, and a `DirichletBC` object:

Python code

```
u_L = Expression("1 + 2*x[1]*x[1]")

def left_boundary(x, on_boundary):
    tol = 1E-14   # tolerance for coordinate comparisons
    return on_boundary and abs(x[0]) < tol

Gamma_0 = DirichletBC(V, u_L, left_boundary)
```

For the boundary $x = 1$ we define a similar code:

Python code

```
u_R = Expression("2 + 2*x[1]*x[1]")

def right_boundary(x, on_boundary):
    tol = 1E-14   # tolerance for coordinate comparisons
    return on_boundary and abs(x[0] - 1) < tol

Gamma_1 = DirichletBC(V, u_R, right_boundary)
```

The various essential conditions are then collected in a list and used in the solution process:

Python code

```
bcs = [Gamma_0, Gamma_1]
...
solve(a == L, u, bcs)
# or
problem = LinearVariationalProblem(a, L, u, bcs)
solver  = LinearVariationalSolver(problem)
solver.solve()
```

If the u values are constant at a part of the boundary, we may use a simple `Constant` object instead of an `Expression` object.

The file `dn2_p2D.py` contains a complete program which demonstrates the constructions above. An extended example with multiple Neumann conditions would have been quite natural now, but this requires marking various parts of the boundary using the mesh function concept and is therefore left to Section 1.5.3.

1.1.15 A linear algebra formulation

Given $a(u,v) = L(v)$, the discrete solution u is computed by inserting $u = \sum_{j=1}^{N} U_j\phi_j$ into $a(u,v)$ and demanding $a(u,v) = L(v)$ to be fulfilled for N test functions $\hat{\phi}_1, \ldots, \hat{\phi}_N$. This implies

$$\sum_{j=1}^{N} a(\phi_j, \hat{\phi}_i)U_j = L(\hat{\phi}_i), \quad i = 1, \ldots, N, \tag{1.53}$$

which is nothing but a linear system,

$$AU = b, \tag{1.54}$$

where the entries in A and b are given by

$$A_{ij} = a(\phi_j, \hat{\phi}_i), \\ b_i = L(\hat{\phi}_i). \tag{1.55}$$

The examples so far have specified the left- and right-hand side of the variational formulation and then asked FEniCS to assemble the linear system and solve it. An alternative to is explicitly call functions for assembling the coefficient matrix A and the right-side vector b, and then solve the linear system $AU = b$ with respect to the U vector. Instead of `solve(a == L, u, b)` we now write

Python code

```
A = assemble(a)
b = assemble(L)
bc.apply(A, b)
u = Function(V)
U = u.vector()
solve(A, U, b)
```

The variables `a` and `L` are as before; that is, `a` refers to the bilinear form involving a `TrialFunction` object (say `u`) and a `TestFunction` object (`v`), and `L` involves a `TestFunction` object (`v`). From `a` and `L`, the `assemble` function can compute the matrix elements $A_{i,j}$ and the vector elements b_i.

The matrix A and vector b are first assembled without incorporating essential (Dirichlet) boundary conditions. Thereafter, the `bc.apply(A, b)` call performs the necessary modifications to the linear system. When we have multiple Dirichlet conditions stored in a list `bcs`, as explained in Section 1.1.14, we must apply each condition in `bcs` to the system:

Python code

```
# bcs is a list of DirichletBC objects
for bc in bcs:
    bc.apply(A, b)
```

There is an alternative function `assemble_system` that can assemble the system and take boundary conditions into account in one call:

Python code

```
A, b = assemble_system(a, L, bcs)
```

The `assemble_system` function incorporates the boundary conditions in the element matrices and vectors, prior to assembly. The conditions are also incorporated in a symmetric way to preserve eventual symmetry of the coefficient matrix. With `bc.apply(A,b)` the matrix `A` is modified in an unsymmetric way.

Note that the solution `u` is, as before, a `Function` object. The degrees of freedom, $U = A^{-1}b$, are filled into `u`'s `Vector` object (`u.vector()`) by the `solve` function.

The object `A` is of type `Matrix`, while `b` and `u.vector()` are of type `Vector`. We may convert the matrix and vector data to `numpy` arrays by calling the `array()` method as shown before. If you wonder how essential boundary conditions are incorporated in the linear system, you can print out `A` and `b` before and after the `bc.apply(A, b)` call:

Python code

```
if mesh.num_cells() < 16:  # print for small meshes only
    print A.array()
    print b.array()
bc.apply(A, b)
```

```
if mesh.num_cells() < 16:
    print A.array()
    print b.array()
```

With access to the elements in `A` as a `numpy` array we can easily do computations on this matrix, such as computing the eigenvalues (using the `numpy.linalg.eig` function). We can alternatively dump `A` and `b` to file in MATLAB format and invoke MATLAB or Octave to analyze the linear system. Dumping the arrays `A` and `b` to MATLAB format is done by

Python code

```
import scipy.io
scipy.io.savemat("Ab.mat", {"A": A, "b": b})
```

Writing `load Ab.mat` in MATLAB or Octave will then make the variables `A` and `b` available for computations.

Matrix processing in Python or MATLAB/Octave is only feasible for small PDE problems since the `numpy` arrays or matrices in MATLAB file format are dense matrices. DOLFIN also has an interface to the eigensolver package SLEPc, which is a preferred tool for computing the eigenvalues of large, sparse matrices of the type encountered in PDE problems (see `demo/la/eigenvalue` in the DOLFIN source code tree for a demo).

A complete code where the linear system $AU = b$ is explicitly assembled and solved is found in the file `dn3_p2D.py` in the directory `stationary/poisson`. This code solves the same problem as in `dn2_p2D.py` (Section 1.1.14). For small linear systems, the program writes out `A` and `b` before and after incorporation of essential boundary conditions and illustrates the difference between `assemble` and `assemble_system`. The reader is encouraged to run the code for a 2×1 mesh (`UnitSquare(2, 1)` and study the output of `A`.

By default, `solve(A, U, b)` applies sparse LU decomposition as solver. Specification of an iterative solver and preconditioner is done through two optional arguments:

Python code

```
solve(A, U, b, "cg", "ilu")
```

Appropriate names of solvers and preconditioners are found in Section 1.7.4.

To control tolerances in the stopping criterion and the maximum number of iterations, one can explicitly form a `KrylovSolver` object and set items in its `parameters` attribute (see Section 1.1.5):

Python code

```
solver = KrylovSolver("cg", "ilu")
solver.parameters["absolute_tolerance"] = 1E-7
solver.parameters["relative_tolerance"] = 1E-4
solver.parameters["maximum_iterations"] = 1000
u = Function(V)
U = u.vector()
set_log_level(DEBUG)
solver.solve(A, U, b)
```

The program `dn4_p2D.py` is a modification of `dn3_p2D.py` illustrating this latter approach.

The choice of start vector for the iterations in a linear solver is often important. With the `solve(A, U, b)` function the start vector is the vector `U` we feed in for the solution. A start vector with random numbers in the interval $[-100, 100]$ can be computed as

Python code

```
n = u.vector().array().size
U = u.vector()
```

```
U[:] = numpy.random.uniform(-100, 100, n)
solver.parameters['nonzero_initial_guess'] = True
solver.solve(A, U, b)
```

Note that we must turn off the default behavior of setting the start vector ("initial guess") to zero. A random start vector is included in the `dn4_p2D.py` code.

Creating the linear system explicitly in a program can have some advantages in more advanced problem settings. For example, A may be constant throughout a time-dependent simulation, so we can avoid recalculating A at every time level and save a significant amount of simulation time. Sections 1.3.2 and 1.3.3 deal with this topic in detail.

1.1.16 Parameterizing the number of space dimensions

FEniCS makes it is easy to write a unified simulation code that can operate in 1D, 2D, and 3D. We will conveniently make use of this feature in forthcoming examples. As an appetizer, go back to the introductory program `d1_p2D.py` in the `stationary/poisson` directory and change the mesh construction from `UnitSquare(6, 4)` to `UnitCube(6, 4, 5)`. Now the domain is the unit cube with $6 \times 4 \times 5$ cells. Run the program and observe that we can solve a 3D problem without any other modifications (!). The visualization allows you rotate to the cube and observe the function values as colors on the boundary.

The forthcoming material introduces some convenient technicalities such that the same program can run in 1D, 2D, or 3D without any modifications. Consider the simple problem

$$u''(x) = 2 \text{ in } [0,1], \quad u(0) = 0,\ u(1) = 1, \tag{1.56}$$

with exact solution $u(x) = x^2$. Our aim is to formulate and solve this problem in a 2D and a 3D domain as well. We may generalize the domain $[0,1]$ to a box of any size in the y and z directions and pose homogeneous Neumann conditions $\partial u/\partial n = 0$ at all additional boundaries $y = \text{const}$ and $z = \text{const}$ to ensure that u only varies with x. For example, let us choose a unit hypercube as domain: $\Omega = [0,1]^d$, where d is the number of space dimensions. The generalized d-dimensional Poisson problem then reads

$$\begin{aligned} \Delta u &= 2 \quad \text{in } \Omega, \\ u &= 0 \quad \text{on } \Gamma_0, \\ u &= 1 \quad \text{on } \Gamma_1, \\ \frac{\partial u}{\partial n} &= 0 \quad \text{on } \partial\Omega \setminus (\Gamma_0 \cup \Gamma_1), \end{aligned} \tag{1.57}$$

where Γ_0 is the side of the hypercube where $x = 0$, and where Γ_1 is the side where $x = 1$.

Implementing (1.57) for any d is no more complicated than solving a problem with a specific number of dimensions. The only non-trivial part of the code is actually to define the mesh. We use the command-line to provide user-input to the program. The first argument can be the degree of the polynomial in the finite element basis functions. Thereafter, we supply the cell divisions in the various spatial directions. The number of command-line arguments will then imply the number of space dimensions. For example, writing `3 10 3 4` on the command-line means that we want to approximate u by piecewise polynomials of degree 3, and that the domain is a three-dimensional cube with $10 \times 3 \times 4$ divisions in the x, y, and z directions, respectively. Each of the $10 \times 3 \times 4 = 120$ boxes will be divided into six tetrahedra. The Python code can be quite compact:

Python code

```
degree = int(sys.argv[1])
divisions = [int(arg) for arg in sys.argv[2:]]
d = len(divisions)
domain_type = [UnitInterval, UnitSquare, UnitCube]
```

```
mesh = domain_type[d-1](*divisions)
V = FunctionSpace(mesh, "Lagrange", degree)
```

First note that although `sys.argv[2:]` holds the divisions of the mesh, all elements of the list `sys.argv[2:]` are string objects, so we need to explicitly convert each element to an integer. The construction `domain_type[d-1]` will pick the right name of the object used to define the domain and generate the mesh. Moreover, the argument `*divisions` sends each component of the list `divisions` as a separate argument. For example, in a 2D problem where `divisions` has two elements, the statement

Python code

```
mesh = domain_type[d-1](*divisions)
```

is equivalent to

Python code

```
mesh = UnitSquare(divisions[0], divisions[1])
```

The next part of the program is to set up the boundary conditions. Since the Neumann conditions have $\partial u/\partial n = 0$ we can omit the boundary integral from the weak form. We then only need to take care of Dirichlet conditions at two sides:

Python code

```
tol = 1E-14   # tolerance for coordinate comparisons
def Dirichlet_boundary0(x, on_boundary):
    return on_boundary and abs(x[0]) < tol

def Dirichlet_boundary1(x, on_boundary):
    return on_boundary and abs(x[0] - 1) < tol

bc0 = DirichletBC(V, Constant(0), Dirichlet_boundary0)
bc1 = DirichletBC(V, Constant(1), Dirichlet_boundary1)
bcs = [bc0, bc1]
```

Note that this code is independent of the number of space dimensions. So are the statements defining and solving the variational problem:

Python code

```
u = TrialFunction(V)
v = TestFunction(V)
f = Constant(-2)
a = inner(nabla_grad(u), nabla_grad(v))*dx
L = f*v*dx

u = Function(V)
solve(a == L, u, bcs)
```

The complete code is found in `paD.py` (Poisson problem in any–D).

If we want to parameterize the direction in which u varies, say by the space direction number `e`, we only need to replace `x[0]` in the code by `x[e]`. The parameter `e` could be given as a second command-line argument. The reader is encouraged to perform this modification.

1.2 Nonlinear problems

Now we shall address how to solve nonlinear PDEs in FEniCS. Our sample PDE for implementation is taken as a nonlinear Poisson equation:

$$-\nabla \cdot (q(u)\nabla u) = f. \tag{1.58}$$

The coefficient $q(u)$ makes the equation nonlinear (unless $q(u)$ is constant in u).

To be able to easily verify our implementation, we choose the domain, $q(u)$, f, and the boundary conditions such that we have a simple, exact solution u. Let Ω be the unit hypercube $[0,1]^d$ in d dimensions, $q(u) = (1+u)^m$, $f = 0$, $u = 0$ for $x_0 = 0$, $u = 1$ for $x_0 = 1$, and $\partial u/\partial n = 0$ at all other boundaries $x_i = 0$ and $x_i = 1$, $i = 1, \ldots, d-1$. The coordinates are now represented by the symbols $x_0, \ldots, x_{d-1}$. The exact solution is then

$$u(x_0, \ldots, x_d) = \left((2^{m+1}-1)x_0 + 1\right)^{1/(m+1)} - 1. \tag{1.59}$$

The variational formulation of our model problem reads: find $u \in V$ such that

$$F(u; v) = 0 \quad \forall\, v \in \hat{V}, \tag{1.60}$$

where

$$F(u; v) = \int_\Omega q(u)\nabla u \cdot \nabla v \,\mathrm{d}x, \tag{1.61}$$

and

$$\begin{aligned} \hat{V} &= \{v \in H^1(\Omega) : v = 0 \text{ on } x_0 = 0 \text{ and } x_0 = 1\}, \\ V &= \{v \in H^1(\Omega) : v = 0 \text{ on } x_0 = 0 \text{ and } v = 1 \text{ on } x_0 = 1\}. \end{aligned} \tag{1.62}$$

The discrete problem arises as usual by restricting V and $\hat{V}$ to a pair of discrete spaces. As usual, we omit any subscript on discrete spaces and simply say V and $\hat{V}$ are chosen finite dimensional according to some mesh and element type. The nonlinear problem then reads: find $u \in V$ such that

$$F(u; v) = 0 \quad \forall\, v \in \hat{V}, \tag{1.63}$$

with $u = \sum_{j=1}^N U_j \phi_j$. Since F is a nonlinear function of u, (1.63) gives rise to a system of nonlinear algebraic equations. From now on the interest is only in the discrete problem, and as mentioned in Section 1.1.2, we simply write u instead of u_h to get a closer resemblance in notation between the mathematics and the Python code. When the exact solution needs to be distinguished, we denote it by u_e.

FEniCS can be used in alternative ways for solving a nonlinear PDE problem. We shall in the following subsections go through four solution strategies: 1) a simple Picard-type iteration, 2) a Newton method at the algebraic level, 3) a Newton method at the PDE level, and 4) an automatic approach where FEniCS attacks the nonlinear variational problem directly. The "black box" strategy 4) is definitely the simplest one from a programmer's point of view, but the others give more control of the solution process for nonlinear equations (which also has some pedagogical advantages).

1.2.1 Picard iteration

Picard iteration is an easy way of handling nonlinear PDEs: we simply use a known, previous solution in the nonlinear terms so that these terms become linear in the unknown u. The strategy is also known

as the method of successive substitutions. For our particular problem, we use a known, previous solution in the coefficient $q(u)$. More precisely, given a solution u^k from iteration k, we seek a new (hopefully improved) solution u^{k+1} in iteration $k+1$ such that u^{k+1} solves the *linear problem*

$$\nabla \cdot \left(q(u^k) \nabla u^{k+1} \right) = 0, \quad k = 0, 1, \ldots \tag{1.64}$$

The iterations require an initial guess u^0. The hope is that $u^k \to u$ as $k \to \infty$, and that u^{k+1} is sufficiently close to the exact solution u of the discrete problem after just a few iterations.

We can easily formulate a variational problem for u^{k+1} from Equation (1.64). Equivalently, we can approximate $q(u)$ by $q(u^k)$ in (1.61) to obtain the same linear variational problem. In both cases, the problem consists of seeking $u^{k+1} \in V$ such that

$$\tilde{F}(u^{k+1}; v) = 0 \quad \forall\, v \in \hat{V}, \quad k = 0, 1, \ldots, \tag{1.65}$$

with

$$\tilde{F}(u^{k+1}; v) = \int_\Omega q(u^k) \nabla u^{k+1} \cdot \nabla v \, \mathrm{d}x. \tag{1.66}$$

Since this is a linear problem in the unknown u^{k+1}, we can equivalently use the formulation

$$a(u^{k+1}, v) = L(v), \tag{1.67}$$

with

$$a(u, v) = \int_\Omega q(u^k) \nabla u \cdot \nabla v \, \mathrm{d}x, \tag{1.68}$$

$$L(v) = 0. \tag{1.69}$$

The iterations can be stopped when $\epsilon \equiv ||u^{k+1} - u^k|| < \text{tol}$, where tol is small, say 10^{-5}, or when the number of iterations exceed some critical limit. The latter case will pick up divergence of the method or unacceptable slow convergence.

In the solution algorithm we only need to store u^k and u^{k+1}, called `u_k` and `u` in the code below. The algorithm can then be expressed as follows:

Python code

```
def q(u):
    return (1+u)**m

# Define variational problem for Picard iteration
u = TrialFunction(V)
v = TestFunction(V)
u_k = interpolate(Constant(0.0), V)  # previous (known) u
a = inner(q(u_k)*nabla_grad(u), nabla_grad(v))*dx
f = Constant(0.0)
L = f*v*dx

# Picard iterations
u = Function(V)     # new unknown function
eps = 1.0           # error measure ||u-u_k||
tol = 1.0E-5        # tolerance
iter = 0            # iteration counter
maxiter = 25        # max no of iterations allowed
while eps > tol and iter < maxiter:
    iter += 1
    solve(a == L, u, bcs)
```

```
diff = u.vector().array() - u_k.vector().array()
eps = numpy.linalg.norm(diff, ord=numpy.Inf)
print "iter=%d: norm=%g" % (iter, eps)
u_k.assign(u)   # update for next iteration
```

We need to define the previous solution in the iterations, `u_k`, as a finite element function so that `u_k` can be updated with `u` at the end of the loop. We may create the initial `Function` `u_k` by interpolating an `Expression` or a `Constant` to the same vector space as `u` lives in (`V`).

In the code above we demonstrate how to use `numpy` functionality to compute the norm of the difference between the two most recent solutions. Here we apply the maximum norm (ℓ_∞ norm) on the difference of the solution vectors (`ord=1` and `ord=2` give the ℓ_1 and ℓ_2 vector norms – other norms are possible for `numpy` arrays, see `pydoc numpy.linalg.norm`).

The file `picard_np.py` (Picard iteration for a nonlinear Poisson problem) contains the complete code for this problem. The implementation is d dimensional, with mesh construction and setting of Dirichlet conditions as explained in Section 1.1.16. For a 33×33 grid with $m = 2$ we need 9 iterations for convergence when the tolerance is 10^{-5}.

1.2.2 A Newton method at the algebraic level

After having discretized our nonlinear PDE problem, we may use Newton's method to solve the system of nonlinear algebraic equations. From the continuous variational problem (1.60), the discrete version (1.63) results in a system of equations for the unknown parameters $U_1, \ldots, U_N$ (by inserting $u = \sum_{j=1}^N U_j\phi_j$ and $v = \hat{\phi}_i$ in (1.63)):

$$F_i(U_1, \ldots, U_N) \equiv \sum_{j=1}^N \int_\Omega \left(q\left(\sum_{\ell=1}^N U_\ell \phi_\ell\right) \nabla\phi_j U_j \right) \cdot \nabla\hat{\phi}_i \, \mathrm{d}x = 0, \quad i = 1, \ldots, N. \tag{1.70}$$

Newton's method for the system $F_i(U_1, \ldots, U_j) = 0$, $i = 1, \ldots, N$ can be formulated as

$$\sum_{j=1}^N \frac{\partial}{\partial U_j} F_i(U_1^k, \ldots, U_N^k)\delta U_j = -F_i(U_1^k, \ldots, U_N^k), \quad i = 1, \ldots, N, \tag{1.71}$$

$$U_j^{k+1} = U_j^k + \omega\delta U_j, \quad j = 1, \ldots, N, \tag{1.72}$$

where $\omega \in [0, 1]$ is a relaxation parameter, and k is an iteration index. An initial guess u^0 must be provided to start the algorithm. The original Newton method has $\omega = 1$, but in problems where it is difficult to obtain convergence, so-called *under-relaxation* with $\omega < 1$ may help.

We need, in a program, to compute the Jacobian matrix $\partial F_i/\partial U_j$ and the right-hand side vector $-F_i$. Our present problem has F_i given by (1.70). The derivative $\partial F_i/\partial U_j$ becomes

$$\int_\Omega \left[q'(\sum_{\ell=1}^N U_\ell^k \phi_\ell)\phi_j \nabla(\sum_{j=1}^N U_j^k \phi_j) \cdot \nabla\hat{\phi}_i + q\left(\sum_{\ell=1}^N U_\ell^k \phi_\ell\right) \nabla\phi_j \cdot \nabla\hat{\phi}_i \right] \mathrm{d}x. \tag{1.73}$$

The following results were used to obtain (1.73):

$$\frac{\partial u}{\partial U_j} = \frac{\partial}{\partial U_j} \sum_{j=1}^N U_j\phi_j = \phi_j, \quad \frac{\partial}{\partial U_j}\nabla u = \nabla\phi_j, \quad \frac{\partial}{\partial U_j} q(u) = q'(u)\phi_j. \tag{1.74}$$

We can reformulate the Jacobian matrix in (1.73) by introducing the short notation $u^k = \sum_{j=1}^N U_j^k \phi_j$:

$$\frac{\partial F_i}{\partial U_j} = \int_\Omega \left[q'(u^k)\phi_j \nabla u^k \cdot \nabla \hat{\phi}_i + q(u^k)\nabla \phi_j \cdot \nabla \hat{\phi}_i \right] \mathrm{d}x. \tag{1.75}$$

In order to make FEniCS compute this matrix, we need to formulate a corresponding variational problem. Looking at the linear system of equations in Newton's method,

$$\sum_{j=1}^N \frac{\partial F_i}{\partial U_j} \delta U_j = -F_i, \quad i = 1, \ldots, N,$$

we can introduce v as a general test function replacing $\hat{\phi}_i$, and we can identify the unknown $\delta u = \sum_{j=1}^N \delta U_j \phi_j$. From the linear system we can now go "backwards" to construct the corresponding discrete weak form

$$\int_\Omega \left[q'(u^k)\delta u \nabla u^k \cdot \nabla v + q(u^k)\nabla \delta u \cdot \nabla v \right] \mathrm{d}x = -\int_\Omega q(u^k)\nabla u^k \cdot \nabla v \, \mathrm{d}x. \tag{1.76}$$

Equation (1.76) fits the standard form $a(\delta u, v) = L(v)$ with

$$a(\delta u, v) = \int_\Omega \left[q'(u^k)\delta u \nabla u^k \cdot \nabla v + q(u^k)\nabla \delta u \cdot \nabla v \right] \mathrm{d}x \tag{1.77}$$

$$L(v) = -\int_\Omega q(u^k)\nabla u^k \cdot \nabla v \, \mathrm{d}x. \tag{1.78}$$

Note the important feature in Newton's method that the previous solution u^k replaces u in the formulas when computing the matrix $\partial F_i/\partial U_j$ and vector F_i for the linear system in each Newton iteration.

We now turn to the implementation. To obtain a good initial guess u^0, we can solve a simplified, linear problem, typically with $q(u) = 1$, which yields the standard Laplace equation $\Delta u^0 = 0$. The recipe for solving this problem appears in Sections 1.1.2, 1.1.3, and 1.1.13. The code for computing u^0 becomes as follows:

Python code

```
tol = 1E-14
def left_boundary(x, on_boundary):
    return on_boundary and abs(x[0]) < tol

def right_boundary(x, on_boundary):
    return on_boundary and abs(x[0]-1) < tol

Gamma_0 = DirichletBC(V, Constant(0.0), left_boundary)
Gamma_1 = DirichletBC(V, Constant(1.0), right_boundary)
bcs = [Gamma_0, Gamma_1]

# Define variational problem for initial guess (q(u)=1, m=0)
u = TrialFunction(V)
v = TestFunction(V)
a = inner(nabla_grad(u), nabla_grad(v))*dx
f = Constant(0.0)
L = f*v*dx
A, b = assemble_system(a, L, bcs)
u_k = Function(V)
U_k = u_k.vector()
solve(A, U_k, b)
```

Here, `u_k` denotes the solution function for the previous iteration, so that the solution after each Newton iteration is `u = u_k + omega*du`. Initially, `u_k` is the initial guess we call u^0 in the mathematics.

The Dirichlet boundary conditions for the problem to be solved in each Newton iteration are somewhat different than the conditions for u. Assuming that u^k fulfills the Dirichlet conditions for u, δu must be zero at the boundaries where the Dirichlet conditions apply, in order for $u^{k+1} = u^k + \omega\delta u$ to fulfill the right Dirichlet values. We therefore define an additional list of Dirichlet boundary conditions objects for δu:

Python code

```
Gamma_0_du = DirichletBC(V, Constant(0), left_boundary)
Gamma_1_du = DirichletBC(V, Constant(0), right_boundary)
bcs_du = [Gamma_0_du, Gamma_1_du]
```

The nonlinear coefficient and its derivative must be defined before coding the weak form of the Newton system:

Python code

```
def q(u):
    return (1+u)**m

def Dq(u):
    return m*(1+u)**(m-1)

du = TrialFunction(V) # u = u_k + omega*du
a = inner(q(u_k)*nabla_grad(du), nabla_grad(v))*dx + \
    inner(Dq(u_k)*du*nabla_grad(u_k), nabla_grad(v))*dx
L = -inner(q(u_k)*nabla_grad(u_k), nabla_grad(v))*dx
```

The Newton iteration loop is very similar to the Picard iteration loop in Section 1.2.1:

Python code

```
du = Function(V)
u  = Function(V)  # u = u_k + omega*du
omega = 1.0       # relaxation parameter
eps = 1.0
tol = 1.0E-5
iter = 0
maxiter = 25
while eps > tol and iter < maxiter:
    iter += 1
    A, b = assemble_system(a, L, bcs_du)
    solve(A, du.vector(), b)
    eps = numpy.linalg.norm(du.vector().array(), ord=numpy.Inf)
    print "Norm:", eps
    u.vector()[:] = u_k.vector() + omega*du.vector()
    u_k.assign(u)
```

There are other ways of implementing the update of the solution as well:

Python code

```
u.assign(u_k)  # u = u_k
u.vector().axpy(omega, du.vector())

# or
u.vector()[:] += omega*du.vector()
```

The `axpy(a, y)` operation adds a scalar `a` times a `Vector` `y` to a `Vector` object. It is usually a fast operation calling up an optimized BLAS routine for the calculation.

Mesh construction for a d-dimensional problem with arbitrary degree of the Lagrange elements can be done as explained in Section 1.1.16. The complete program appears in the file `alg_newton_np.py`.

1.2.3 A Newton method at the PDE level

Although Newton's method in PDE problems is normally formulated at the linear algebra level; that is, as a solution method for systems of nonlinear algebraic equations, we can also formulate the method at the PDE level. This approach yields a linearization of the PDEs before they are discretized. FEniCS users will probably find this technique simpler to apply than the more standard method of Section 1.2.2.

Given an approximation to the solution field, u^k, we seek a perturbation δu so that

$$u^{k+1} = u^k + \delta u \tag{1.79}$$

fulfills the nonlinear PDE. However, the problem for δu is still nonlinear and nothing is gained. The idea is therefore to assume that δu is sufficiently small so that we can linearize the problem with respect to δu. Inserting u^{k+1} in the PDE, linearizing the q term as

$$q(u^{k+1}) = q(u^k) + q'(u^k)\delta u + \mathcal{O}((\delta u)^2) \approx q(u^k) + q'(u^k)\delta u, \tag{1.80}$$

and dropping other nonlinear terms in δu, we get

$$\nabla\cdot\left(q(u^k)\nabla u^k\right) + \nabla\cdot\left(q(u^k)\nabla\delta u\right) + \nabla\cdot\left(q'(u^k)\delta u\nabla u^k\right) = 0.$$

We may collect the terms with the unknown δu on the left-hand side,

$$\nabla\cdot\left(q(u^k)\nabla\delta u\right) + \nabla\cdot\left(q'(u^k)\delta u\nabla u^k\right) = -\nabla\cdot\left(q(u^k)\nabla u^k\right), \tag{1.81}$$

The weak form of this PDE is derived by multiplying by a test function v and integrating over Ω, integrating the second-order derivatives by parts:

$$\int_\Omega \left(q(u^k)\nabla\delta u\cdot\nabla v + q'(u^k)\delta u\nabla u^k\cdot\nabla v\right)\,\mathrm{d}x = -\int_\Omega q(u^k)\nabla u^k\cdot\nabla v\,\mathrm{d}x. \tag{1.82}$$

The variational problem reads: find $\delta u \in V$ such that $a(\delta u, v) = L(v)$ for all $v \in \hat{V}$, where

$$a(\delta u, v) = \int_\Omega \left(q(u^k)\nabla\delta u\cdot\nabla v + q'(u^k)\delta u\nabla u^k\cdot\nabla v\right)\,\mathrm{d}x, \tag{1.83}$$

$$L(v) = -\int_\Omega q(u^k)\nabla u^k\cdot\nabla v\,\mathrm{d}x. \tag{1.84}$$

The function spaces V and $\hat{V}$, being continuous or discrete, are as in the linear Poisson problem from Section 1.1.2.

We must provide some initial guess, e.g., the solution of the PDE with $q(u) = 1$. The corresponding weak form $a_0(u^0, v) = L_0(v)$ has

$$a_0(u, v) = \int_\Omega \nabla u\cdot\nabla v\,\mathrm{d}x, \quad L(v) = 0. \tag{1.85}$$

Thereafter, we enter a loop and solve $a(\delta u, v) = L(v)$ for δu and compute a new approximation $u^{k+1} = u^k + \delta u$. Note that δu is a correction, so if u^0 satisfies the prescribed Dirichlet conditions on some part Γ_D of the boundary, we must demand $\delta u = 0$ on Γ_D.

Looking at (1.83) and (1.84), we see that the variational form is the same as for the Newton method at the algebraic level in Section 1.2.2. Since Newton's method at the algebraic level required some "backward" construction of the underlying weak forms, FEniCS users may prefer Newton's method at the PDE level, which is more straightforward. There is seemingly no need for differentiations to derive a Jacobian matrix, but a mathematically equivalent derivation is done when nonlinear terms are linearized using the first two Taylor series terms and when products in the perturbation δu are neglected.

The implementation is identical to the one in Section 1.2.2 and is found in the file `pde_newton_np.py`. The reader is encouraged to go through this code to be convinced that the present method actually ends up with the same program as needed for the Newton method at the linear algebra level in Section 1.2.2.

1.2.4 Solving the nonlinear variational problem directly

The previous hand-calculations and manual implementation of Picard or Newton methods can be automated by tools in FEniCS. In a nutshell, one can just write

Python code

```
problem = NonlinearVariationalProblem(F, u, bcs, J)
solver  = NonlinearVariationalSolver(problem)
solver.solve()
```

where `F` corresponds to the nonlinear form $F(u;v)$, `u` is the unknown `Function` object, `bcs` represents the essential boundary conditions (list of `DirichletBC` objects), and `J` is a variational form for the Jacobian of `F`.

Let us explain in detail how to use the built-in tools for nonlinear variational problems and their solution. The `F` form corresponding to (1.61) is straightforwardly defined as follows, assuming `q(u)` is coded as a Python function:

Python code

```
v  = TestFunction(V)
u_ = Function(V)  # the unknown
F  = inner(q(u_)*nabla_grad(u_), nabla_grad(v))*dx
```

Note here that `u_` is a `Function` (not a `TrialFunction`). An alternative and perhaps more intuitive formula for F is to define $F(u;v)$ directly in terms of a trial function for u and a test function for v, and then create the proper `F` by

Python code

```
u  = TrialFunction(V)
v  = TestFunction(V)
F  = inner(q(u)*nabla_grad(u), nabla_grad(v))*dx
u_ = Function(V)  # most recently computed solution
F  = action(F, u_)
```

The latter statement is equivalent to $F(u = u_-;v)$, where u_- is an existing finite element function representing the most recently computed approximation to the solution.

The Jacobian or derivative J (`J`) of F (`F`) is formally the Gateaux derivative $DF(u^k;\delta u,v)$ of $F(u;v)$ at $u = u^k$ in the direction of δu. Technically, this Gateaux derivative is derived by computing

$$\lim_{\epsilon\to 0}\frac{d}{d\epsilon}F_i(u^k + \epsilon\delta u;v) \tag{1.86}$$

The δu is now the trial function and u^k is as usual the previous approximation to the solution u. We start with

$$\frac{d}{d\epsilon}\int_\Omega \nabla v\cdot\left(q(u^k+\epsilon\delta u)\nabla(u^k+\epsilon\delta u)\right)\,\mathrm{d}x \tag{1.87}$$

and obtain

$$\int_\Omega \nabla v\cdot\left[q'(u^k+\epsilon\delta u)\delta u\nabla(u^k+\epsilon\delta u)+q(u^k+\epsilon\delta u)\nabla\delta u\right]\,\mathrm{d}x, \tag{1.88}$$

which leads to

$$\int_\Omega \nabla v\cdot\left[q'(u^k)\delta u\nabla(u^k)+q(u^k)\nabla\delta u\right]\,\mathrm{d}x, \tag{1.89}$$

as $\epsilon\to 0$. This last expression is the Gateaux derivative of F. We may use J or $a(\delta u,v)$ for this derivative, the latter having the advantage that we easily recognize the expression as a bilinear form. However, in the forthcoming code examples `J` is used as variable name for the Jacobian.

The specification of `J` goes as follows if `du` is the `TrialFunction`:

Python code

```
du = TrialFunction(V)
v  = TestFunction(V)
u_ = Function(V)      # the most recently computed solution
F  = inner(q(u_)*nabla_grad(u_), nabla_grad(v))*dx

J = inner(q(u_)*nabla_grad(du), nabla_grad(v))*dx + \
    inner(Dq(u_)*du*nabla_grad(u_), nabla_grad(v))*dx
```

The alternative specification of `F`, with `u` as `TrialFunction`, leads to

Python code

```
u  = TrialFunction(V)
v  = TestFunction(V)
u_ = Function(V)      # the most recently computed solution
F  = inner(q(u)*nabla_grad(u), nabla_grad(v))*dx
F  = action(F, u_)

J = inner(q(u_)*nabla_grad(u), nabla_grad(v))*dx + \
    inner(Dq(u_)*u*nabla_grad(u_), nabla_grad(v))*dx
```

The UFL language, used to specify weak forms, supports differentiation of forms. This feature facilitates automatic *symbolic* computation of the Jacobian `J` by calling the function `derivative` with `F`, the most recently computed solution (`Function`), and the unknown (`TrialFunction`) as parameters:

Python code

```
du = TrialFunction(V)
v  = TestFunction(V)
u_ = Function(V)      # the most recently computed solution
F  = inner(q(u_)*nabla_grad(u_), nabla_grad(v))*dx

J  = derivative(F, u_, du)  # Gateaux derivative in dir. of du
```

or

Python code

```
u  = TrialFunction(V)
v  = TestFunction(V)
u_ = Function(V)      # the most recently computed solution
F  = inner(q(u)*nabla_grad(u), nabla_grad(v))*dx
F  = action(F, u_)
```

```
J  = derivative(F, u_, u)   # Gateaux derivative in dir. of u
```

The `derivative` function is obviously very convenient in problems where differentiating `F` by hand implies lengthy calculations.

The preferred implementation of `F` and `J`, depending on whether `du` or `u` is the `TrialFunction` object, is a matter of personal taste. Derivation of the Gateaux derivative by hand, as shown above, is most naturally matched by an implementation where `du` is the `TrialFunction`, while use of automatic symbolic differentiation through the `derivative` function is most naturally matched with an implementation where `u` is the `TrialFunction`. We have implemented both approaches in two files: `vp1_np.py` with `u` as `TrialFunction`, and `vp2_np.py` with `du` as `TrialFunction`. Both files are located in the `stationary/nonlinear_poisson` directory. The first command-line argument determines if the Jacobian is to be automatically derived or computed from the hand-derived formula.

The following code defines the nonlinear variational problem and an associated solver based on Newton's method. We also demonstrate how key parameters in Newton's method can be set, as well as the choice of solver and preconditioner, and associated parameters, for the linear system occurring in the Newton iteration.

Python code

```
problem = NonlinearVariationalProblem(F, u_, bcs, J)
solver  = NonlinearVariationalSolver(problem)

prm = solver.parameters
prm["newton_solver"]["absolute_tolerance"] = 1E-8
prm["newton_solver"]["relative_tolerance"] = 1E-7
prm["newton_solver"]["maximum_iterations"] = 25
prm["newton_solver"]["relaxation_parameter"] = 1.0
if iterative_solver:
    prm["linear_solver"] = "gmres"
    prm["preconditioner"] = "ilu"
    prm["krylov_solver"]["absolute_tolerance"] = 1E-9
    prm["krylov_solver"]["relative_tolerance"] = 1E-7
    prm["krylov_solver"]["maximum_iterations"] = 1000
    prm["krylov_solver"]["gmres"]["restart"] = 40
    prm["krylov_solver"]["preconditioner"]["ilu"]["fill_level"] = 0
set_log_level(PROGRESS)

solver.solve()
```

A list of available parameters and their default values can as usual be printed by calling `info(prm, True)`. The `u_` we feed to the nonlinear variational problem object is filled with the solution by the call `solver.solve()`.

1.3 Time-dependent problems

The examples in Section 1.1 illustrate that solving linear, stationary PDE problems with the aid of FEniCS is easy and requires little programming. That is, FEniCS automates the spatial discretization by the finite element method. The solution of nonlinear problems, as we showed in Section 1.2, can also be automated (see Section 1.2.4), but many scientists will prefer to code the solution strategy of the nonlinear problem themselves and experiment with various combinations of strategies in difficult

problems. Time-dependent problems are somewhat similar in this respect: we have to add a time discretization scheme, which is often quite simple, making it natural to explicitly code the details of the scheme so that the programmer has full control. We shall explain how easily this is accomplished through examples.

1.3.1 A diffusion problem and its discretization

Our time-dependent model problem for teaching purposes is naturally the simplest extension of the Poisson problem into the time domain; that is, the diffusion problem

$$\frac{\partial u}{\partial t} = \Delta u + f \text{ in } \Omega, \text{ for } t > 0, \tag{1.90}$$

$$u = u_0 \text{ on } \partial\Omega, \text{ for } t > 0, \tag{1.91}$$

$$u = I \text{ at } t = 0. \tag{1.92}$$

Here, u varies with space and time, e.g., $u = u(x, y, t)$ if the spatial domain Ω is two-dimensional. The source function f and the boundary values u_0 may also vary with space and time. The initial condition I is a function of space only.

A straightforward approach to solving time-dependent PDEs by the finite element method is to first discretize the time derivative by a finite difference approximation, which yields a recursive set of stationary problems, and then turn each stationary problem into a variational formulation.

Let superscript k denote a quantity at time t_k, where k is an integer counting time levels. For example, u^k means u at time level k. A finite difference discretization in time first consists in sampling the PDE at some time level, say k:

$$\frac{\partial}{\partial t}u^k = \Delta u^k + f^k. \tag{1.93}$$

The time-derivative can be approximated by a finite difference. For simplicity and stability reasons we choose a simple backward difference:

$$\frac{\partial}{\partial t}u^k \approx \frac{u^k - u^{k-1}}{\mathrm{d}t}, \tag{1.94}$$

where $\mathrm{d}t$ is the time discretization parameter. Inserting (1.94) in (1.93) yields

$$\frac{u^k - u^{k-1}}{\mathrm{d}t} = \Delta u^k + f^k. \tag{1.95}$$

This is our time-discrete version of the diffusion PDE (1.90). Reordering (1.95) so that u^k appears on the left-hand side only, shows that (1.95) is a recursive set of spatial (stationary) problems for u^k (assuming u^{k-1} is known from computations at the previous time level):

$$u^0 = I, \tag{1.96}$$

$$u^k - \mathrm{d}t\Delta u^k = u^{k-1} + \mathrm{d}t f^k, \quad k = 1, 2, \ldots \tag{1.97}$$

Given I, we can solve for u^0, u^1, u^2, and so on.

We use a finite element method to solve the equations (1.96) and (1.97). This requires turning the equations into weak forms. As usual, we multiply by a test function $v \in \hat{V}$ and integrate second-derivatives by parts. Introducing the symbol u for u^k (which is natural in the program too), the resulting weak form can be conveniently written in the standard notation: $a_0(u, v) = L_0(v)$ for (1.96)

and $a(u,v) = L(v)$ for (1.97), where

$$a_0(u,v) = \int_\Omega uv \,\mathrm{d}x, \tag{1.98}$$

$$L_0(v) = \int_\Omega Iv \,\mathrm{d}x, \tag{1.99}$$

$$a(u,v) = \int_\Omega (uv + \,\mathrm{d}t\nabla u \cdot \nabla v) \,\mathrm{d}x, \tag{1.100}$$

$$L(v) = \int_\Omega \left(u^{k-1} + \,\mathrm{d}t f^k\right) v \,\mathrm{d}x. \tag{1.101}$$

The continuous variational problem is to find $u^0 \in V$ such that $a_0(u^0, v) = L_0(v)$ holds for all $v \in \hat{V}$, and then find $u^k \in V$ such that $a(u^k, v) = L(v)$ for all $v \in \hat{V}$, $k = 1, 2, \ldots$.

Approximate solutions in space are found by restricting the functional spaces V and $\hat{V}$ to finite-dimensional spaces, exactly as we have done in the Poisson problems. We shall use the symbol u for the finite element approximation at time t_k. In case we need to distinguish this space-time discrete approximation from the exact solution of the continuous diffusion problem, we use u_e for the latter. By u^{k-1} we mean, from now on, the finite element approximation of the solution at time t_{k-1}.

Note that the forms a_0 and L_0 are identical to the forms met in Section 1.1.9, except that the test and trial functions are now scalar fields and not vector fields. Instead of solving (1.96) by a finite element method; that is, projecting I onto V via the problem $a_0(u,v) = L_0(v)$, we could simply interpolate u^0 from I. That is, if $u^0 = \sum_{j=1}^N U_j^0 \phi_j$, we simply set $U_j = I(x_j, y_j)$, where (x_j, y_j) are the coordinates of node number j. We refer to these two strategies as computing the initial condition by either projecting I or interpolating I. Both operations are easy to compute through one statement, using either the `project` or `interpolate` function.

1.3.2 Implementation

Our program needs to perform the time stepping explicitly, but can rely on FEniCS to easily compute a_0, L_0, a, and L, and solve the linear systems for the unknowns. We realize that a does not depend on time, which means that its associated matrix also will be time independent. Therefore, it is wise to explicitly create matrices and vectors as in Section 1.1.15. The matrix A arising from a can be computed prior to the time stepping, so that we only need to compute the right-hand side b, corresponding to L, in each pass in the time loop. Let us express the solution procedure in algorithmic form, writing u for the unknown spatial function at the new time level (u^k) and u_1 for the spatial solution at one earlier time level (u^{k-1}):

define Dirichlet boundary condition (u_0, Dirichlet boundary, etc.)
if u_1 is to be computed by projecting I:
    define a_0 and L_0
    assemble matrix M from a_0 and vector b from L_0
    solve $MU = b$ and store U in u_1
else: (interpolation)
    let u_1 interpolate I
define a and L
assemble matrix A from a
set some stopping time T
$t = \,\mathrm{d}t$
while $t \leqslant T$
    assemble vector b from L

apply essential boundary conditions
solve $AU = b$ for U and store in u
$t \leftarrow t + \mathrm{d}t$
$u_1 \leftarrow u$ (be ready for next step)

Before starting the coding, we shall construct a problem where it is easy to determine if the calculations are correct. The simple backward time difference is exact for linear functions, so we decide to have a linear variation in time. Combining a second-degree polynomial in space with a linear term in time,

$$u = 1 + x^2 + \alpha y^2 + \beta t, \tag{1.102}$$

yields a function whose computed values at the nodes may be exact, regardless of the size of the elements and $\mathrm{d}t$, as long as the mesh is uniformly partitioned. Inserting (1.102) in the PDE problem (1.90), it follows that u_0 must be given as (1.102) and that $f(x,y,t) = \beta - 2 - 2\alpha$ and $I(x,y) = 1 + x^2 + \alpha y^2$.

A new programming issue is how to deal with functions that vary in space *and time*, such as the boundary condition u_0 given by (1.102). A natural solution is to apply an `Expression` object with time t as a parameter, in addition to the parameters α and β (see Section 1.1.7 for `Expression` objects with parameters):

Python code

```
alpha = 3; beta = 1.2
u0 = Expression("1 + x[0]*x[0] + alpha*x[1]*x[1] + beta*t",
                alpha=alpha, beta=beta, t=0)
```

The time parameter can later be updated by assigning values to `u0.t`.

The essential boundary conditions, along the whole boundary in this case, are set in the usual way,

Python code

```
def boundary(x, on_boundary):  # define the Dirichlet boundary
    return on_boundary

bc = DirichletBC(V, u0, boundary)
```

We shall use `u` for the unknown u at the new time level and `u_1` for u at the previous time level. The initial value of `u_1`, implied by the initial condition on u, can be computed by either projecting or interpolating I. The $I(x,y)$ function is available in the program through `u0`, as long as `u0.t` is zero. We can then do

Python code

```
u_1 = interpolate(u0, V)
# or
u_1 = project(u0, V)
```

Note that we could, as an equivalent alternative to using `project`, define a_0 and L_0 as we did in Section 1.1.9 and solve the associated variational problem. To actually recover (1.102) to machine precision, it is important not to compute the discrete initial condition by projecting I, but by interpolating I so that the nodal values are exact at $t = 0$ (projection results in approximative values at the nodes).

The definition of a and L goes as follows:

Python code

```
dt = 0.3      # time step

u = TrialFunction(V)
```

```
v = TestFunction(V)
f = Constant(beta - 2 - 2*alpha)

a = u*v*dx + dt*inner(nabla_grad(u), nabla_grad(v))*dx
L = (u_1 + dt*f)*v*dx

A = assemble(a)   # assemble only once, before the time stepping
```

Finally, we perform the time stepping in a loop:

Python code

```
u = Function(V)   # the unknown at a new time level
T = 2             # total simulation time
t = dt

while t <= T:
    b = assemble(L)
    u0.t = t
    bc.apply(A, b)
    solve(A, u.vector(), b)

    t += dt
    u_1.assign(u)
```

Observe that `u0.t` must be updated before the `bc.apply` statement, to enforce computation of Dirichlet conditions at the current time level.

The time loop above does not contain any comparison of the numerical and the exact solution, which we must include in order to verify the implementation. As in many previous examples, we compute the difference between the array of nodal values of `u` and the array of the interpolated exact solution. The following code is to be included inside the loop, after `u` is found:

Python code

```
u_e = interpolate(u0, V)
maxdiff = numpy.abs(u_e.vector().array()-u.vector().array()).max()
print "Max error, t=%.2f: %-10.3f" % (t, maxdiff)
```

The right-hand side vector `b` must obviously be recomputed at each time level. With the construction `b = assemble(L)`, a new vector for `b` is allocated in memory in every pass of the time loop. It would be much more memory friendly to reuse the storage of the `b` we already have. This is easily accomplished by

Python code

```
b = assemble(L, tensor=b)
```

That is, we send in our previous `b`, which is then filled with new values and returned from `assemble`. Now there will be only a single memory allocation of the right-hand side vector. Before the time loop we set `b = None` such that `b` is defined in the first call to `assemble`.

The complete program code for this time-dependent case is stored in the file `d1_d2D.py` in the directory `transient/diffusion`.

1.3.3 *Avoiding assembly*

The purpose of this section is to present a technique for speeding up FEniCS simulators for time-dependent problems where it is possible to perform all assembly operations prior to the time loop. There are two costly operations in the time loop: assembly of the right-hand side b and solution of

the linear system via the `solve` call. The assembly process involves work proportional to the number of degrees of freedom N, while the solve operation has a work estimate of $\mathcal{O}(N^\alpha)$, for some $\alpha \geqslant 1$. As $N \to \infty$, the solve operation will dominate for $\alpha > 1$, but for the values of N typically used on smaller computers, the assembly step may still represent a considerable part of the total work at each time level. Avoiding repeated assembly can therefore contribute to a significant speed-up of a finite element code in time-dependent problems.

To see how repeated assembly can be avoided, we look at the $L(v)$ form in (1.101), which in general varies with time through u^{k-1}, f^k, and possibly also with $\mathrm{d}t$ if the time step is adjusted during the simulation. The technique for avoiding repeated assembly consists in expanding the finite element functions in sums over the basis functions ϕ_i, as explained in Section 1.1.15, to identify matrix-vector products that build up the complete system. We have $u^{k-1} = \sum_{j=1}^N U_j^{k-1}\phi_j$, and we can expand f^k as $f^k = \sum_{j=1}^N F_j^k\phi_j$. Inserting these expressions in $L(v)$ and using $v = \hat{\phi}_i$ result in

$$\begin{aligned}\int_\Omega \left(u^{k-1} + \mathrm{d}t f^k\right) v \,\mathrm{d}x &= \int_\Omega \left(\sum_{j=1}^N U_j^{k-1}\phi_j + \mathrm{d}t \sum_{j=1}^N F_j^k\phi_j\right)\hat{\phi}_i \,\mathrm{d}x, \\ &= \sum_{j=1}^N \left(\int_\Omega \hat{\phi}_i\phi_j \,\mathrm{d}x\right) U_j^{k-1} + \mathrm{d}t \sum_{j=1}^N \left(\int_\Omega \hat{\phi}_i\phi_j \,\mathrm{d}x\right) F_j^k.\end{aligned} \tag{1.103}$$

Introducing $M_{ij} = \int_\Omega \hat{\phi}_i\phi_j \,\mathrm{d}x$, we see that the last expression can be written

$$\sum_{j=1}^N M_{ij}U_j^{k-1} + \mathrm{d}t \sum_{j=1}^N M_{ij}F_j^k, \tag{1.104}$$

which is nothing but two matrix-vector products,

$$MU^{k-1} + \mathrm{d}t MF^k, \tag{1.105}$$

if M is the matrix with entries M_{ij} and

$$U^{k-1} = (U_1^{k-1}, \ldots, U_N^{k-1})^\top, \tag{1.106}$$

and

$$F^k = (F_1^k, \ldots, F_N^k)^\top. \tag{1.107}$$

We have immediate access to U^{k-1} in the program since that is the vector in the `u_1` function. The F^k vector can easily be computed by interpolating the prescribed f function (at each time level if f varies with time). Given M, U^{k-1}, and F^k, the right-hand side b can be calculated as

$$b = MU^{k-1} + \mathrm{d}t MF^k. \tag{1.108}$$

That is, no assembly is necessary to compute b.

The coefficient matrix A can also be split into two terms. We insert $v = \hat{\phi}_i$ and $u^k = \sum_{j=1}^N U_j^k\phi_j$ in the expression (1.100) to get

$$\sum_{j=1}^N \left(\int_\Omega \hat{\phi}_i\phi_j \,\mathrm{d}x\right) U_j^k + \mathrm{d}t \sum_{j=1}^N \left(\int_\Omega \nabla\hat{\phi}_i \cdot \nabla\phi_j \,\mathrm{d}x\right) U_j^k, \tag{1.109}$$

which can be written as a sum of matrix-vector products,

$$MU^k + \mathrm{d}tKU^k = (M + \mathrm{d}tK)U^k, \tag{1.110}$$

if we identify the matrix M with entries M_{ij} as above and the matrix K with entries

$$K_{ij} = \int_\Omega \nabla\hat{\phi}_i \cdot \nabla\phi_j \,\mathrm{d}x. \tag{1.111}$$

The matrix M is often called the "mass matrix" while "stiffness matrix" is a common nickname for K. The associated bilinear forms for these matrices, as we need them for the assembly process in a FEniCS program, become

$$a_K(u,v) = \int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x, \tag{1.112}$$

$$a_M(u,v) = \int_\Omega uv \,\mathrm{d}x. \tag{1.113}$$

The linear system at each time level, written as $AU^k = b$, can now be computed by first computing M and K, and then forming $A = M + \mathrm{d}tK$ at $t = 0$, while b is computed as $b = MU^{k-1} + \mathrm{d}tMF^k$ at each time level.

The following modifications are needed in the `d1_d2D.py` program from the previous section in order to implement the new strategy of avoiding assembly at each time level:

1. Define separate forms a_M and a_K
2. Assemble a_M to M and a_K to K
3. Compute $A = M + \mathrm{d}t\,K$
4. Define f as an `Expression`
5. Interpolate the formula for f to a finite element function F^k
6. Compute $b = MU^{k-1} + \mathrm{d}tMF^k$

The relevant code segments become

Python code

```
# 1.
a_K = inner(nabla_grad(u), nabla_grad(v))*dx
a_M = u*v*dx

# 2. and 3.
M = assemble(a_M)
K = assemble(a_K)
A = M + dt*K

# 4.
f = Expression("beta - 2 - 2*alpha", beta=beta, alpha=alpha)

# 5. and 6.
while t <= T:
    f_k = interpolate(f, V)
    F_k = fk.vector()
    b = M*u_1.vector() + dt*M*F_k
```

The complete program appears in the file `d2_d2D.py`.

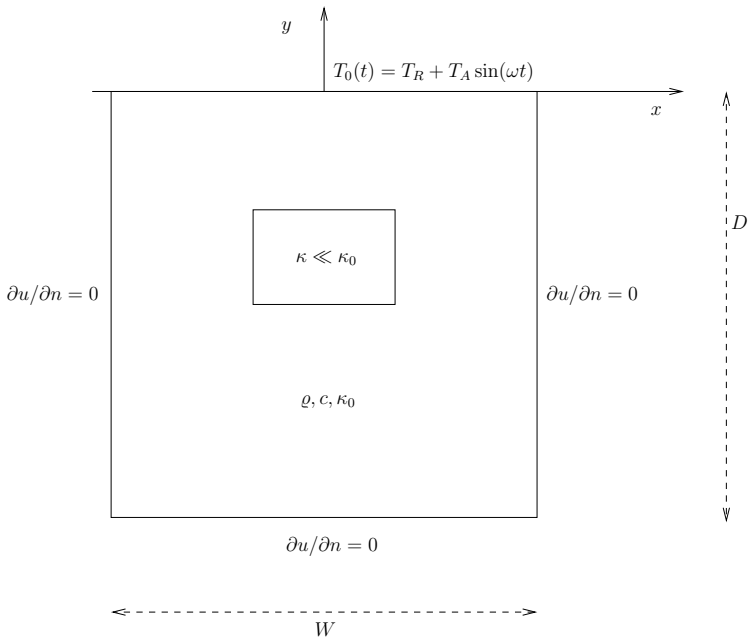


Figure 1.7: Sketch of a (2D) problem involving heating and cooling of the ground due to an oscillating surface temperature

1.3.4 A physical example

With the basic programming techniques for time-dependent problems from Sections 1.3.3–1.3.2 we are ready to attack more physically realistic examples. The next example concerns the question: How is the temperature in the ground affected by day and night variations at the earth's surface? We consider some box-shaped domain Ω in d dimensions with coordinates $x_0, \ldots, x_{d-1}$ (the problem is meaningful in 1D, 2D, and 3D). At the top of the domain, $x_{d-1} = 0$, we have an oscillating temperature

$$T_0(t) = T_R + T_A \sin(\omega t), \tag{1.114}$$

where T_R is some reference temperature, T_A is the amplitude of the temperature variations at the surface, and ω is the frequency of the temperature oscillations. At all other boundaries we assume that the temperature does not change anymore when we move away from the boundary; that is, the normal derivative is zero. Initially, the temperature can be taken as T_R everywhere. The heat conductivity properties of the soil in the ground may vary with space so we introduce a variable coefficient κ reflecting this property. Figure 1.7 shows a sketch of the problem, with a small region where the heat conductivity is much lower.

The initial-boundary value problem for this problem reads

$$\varrho c \frac{\partial T}{\partial t} = \nabla \cdot (\kappa \nabla T) \text{ in } \Omega \times (0, t_{\text{stop}}], \tag{1.115}$$

$$T = T_0(t) \text{ on } \Gamma_0, \tag{1.116}$$

$$\frac{\partial T}{\partial n} = 0 \text{ on } \partial\Omega \backslash \Gamma_0, \tag{1.117}$$

$$T = T_R \text{ at } t = 0. \tag{1.118}$$

Here, ϱ is the density of the soil, c is the heat capacity, κ is the thermal conductivity (heat conduction coefficient) in the soil, and Γ_0 is the surface boundary $x_{d-1} = 0$.

We use a θ-scheme in time; that is, the evolution equation $\partial P/\partial t = Q(t)$ is discretized as

$$\frac{P^k - P^{k-1}}{\mathrm{d}t} = \theta Q^k + (1-\theta)Q^{k-1}, \tag{1.119}$$

where $\theta \in [0,1]$ is a weighting factor: $\theta = 1$ corresponds to the backward difference scheme, $\theta = 1/2$ to the Crank–Nicolson scheme, and $\theta = 0$ to a forward difference scheme. The θ-scheme applied to our PDE results in

$$\varrho c \frac{T^k - T^{k-1}}{\mathrm{d}t} = \theta \nabla \cdot \left(\kappa \nabla T^k\right) + (1-\theta)\nabla \cdot \left(k \nabla T^{k-1}\right). \tag{1.120}$$

Bringing this time-discrete PDE into weak form follows the technique shown many times earlier in this tutorial. In the standard notation $a(T,v) = L(v)$ the weak form has

$$a(T,v) = \int_\Omega \left(\varrho c T v + \theta\, \mathrm{d}t \kappa \nabla T \cdot \nabla v\right)\, \mathrm{d}x, \tag{1.121}$$

$$L(v) = \int_\Omega \left(\varrho c T^{k-1} v - (1-\theta)\, \mathrm{d}t \kappa \nabla T^{k-1} \cdot \nabla v\right)\, \mathrm{d}x. \tag{1.122}$$

Observe that boundary integrals vanish because of the Neumann boundary conditions.

The size of a 3D box is taken as $W \times W \times D$, where D is the depth and $W = D/2$ is the width. We give the degree of the basis functions at the command-line, then D, and then the divisions of the domain in the various directions. To make a box, rectangle, or interval of arbitrary (not unit) size, we have the DOLFIN classes `Box`, `Rectangle`, and `Interval` at our disposal. The mesh and the function space can be created by the following code:

Python code

```
degree = int(sys.argv[1])
D = float(sys.argv[2])
W = D/2.0
divisions = [int(arg) for arg in sys.argv[3:]]
d = len(divisions)  # no of space dimensions
if d == 1:
    mesh = Interval(divisions[0], -D, 0)
elif d == 2:
    mesh = Rectangle(-W/2, -D, W/2, 0, divisions[0], divisions[1])
elif d == 3:
    mesh = Box(-W/2, -W/2, -D, W/2, W/2, 0,
               divisions[0], divisions[1], divisions[2])
V = FunctionSpace(mesh, "Lagrange", degree)
```

The `Rectangle` and `Box` objects are defined by the coordinates of the "minimum" and "maximum" corners.

Setting Dirichlet conditions at the upper boundary can be done by

Python code

```
T_R = 0; T_A = 1.0; omega = 2*pi

T_0 = Expression("T_R + T_A*sin(omega*t)",
                 T_R=T_R, T_A=T_A, omega=omega, t=0.0)

def surface(x, on_boundary):
    return on_boundary and abs(x[d-1]) < 1E-14

bc = DirichletBC(V, T_0, surface)
```

The κ function can be defined as a constant κ_1 inside the particular rectangular area with a special soil composition, as indicated in Figure 1.7. Outside this area κ is a constant κ_0. The domain of the rectangular area is taken as

$$[-W/4, W/4] \times [-W/4, W/4] \times [-D/2, -D/2 + D/4]$$

in 3D, with $[-W/4, W/4] \times [-D/2, -D/2 + D/4]$ in 2D and $[-D/2, -D/2 + D/4]$ in 1D. Since we need some testing in the definition of the $\kappa(x)$ function, the most straightforward approach is to define a subclass of `Expression`, where we can use a full Python method instead of just a C++ string formula for specifying a function. The method that defines the function is called `eval`:

Python code

```
class Kappa(Function):
    def eval(self, value, x):
        """x: spatial point, value[0]: function value."""
        d = len(x)  # no of space dimensions
        material = 0  # 0: outside, 1: inside
        if d == 1:
            if -D/2. < x[d-1] < -D/2. + D/4.:
                material = 1
        elif d == 2:
            if -D/2. < x[d-1] < -D/2. + D/4. and \
               -W/4. < x[0] < W/4.:
                material = 1
        elif d == 3:
            if -D/2. < x[d-1] < -D/2. + D/4. and \
               -W/4. < x[0] < W/4. and -W/4. < x[1] < W/4.:
                material = 1
        value[0] = kappa_0 if material == 0 else kappa_1
```

The `eval` method gives great flexibility in defining functions, but a downside is that C++ calls up `eval` in Python for each point `x`, which is a slow process, and the number of calls is proportional to the number of nodes in the mesh. Function expressions in terms of strings are compiled to efficient C++ functions, being called from C++, so we should try to express functions as string expressions if possible. (The `eval` method can also be defined through C++ code, but this is more complicated and not covered here.) Using inline if-tests in C++, we can make string expressions for κ:

Python code

```
kappa_str = {}
kappa_str[1] = "x[0] > -D/2 && x[0] < -D/2 + D/4 ? kappa_1 : kappa_0"
kappa_str[2] = "x[0] > -W/4 && x[0] < W/4 "\
               "&& x[1] > -D/2 && x[1] < -D/2 + D/4 ? "\
               "kappa_1 : kappa_0"
kappa_str[3] = "x[0] > -W/4 && x[0] < W/4 "\
               "x[1] > -W/4 && x[1] < W/4 "\
               "&& x[2] > -D/2 && x[2] < -D/2 + D/4 ?"\
               "kappa_1 : kappa_0"

kappa = Expression(kappa_str[d],
                   D=D, W=W, kappa_0=kappa_0, kappa_1=kappa_1)
```

Let `T` denote the unknown spatial temperature function at the current time level, and let `T_1` be the corresponding function at one earlier time level. We are now ready to define the initial condition and the `a` and `L` forms of our problem:

Python code

```
T_1 = interpolate(Constant(T_R), V)

rho = 1
c = 1
period = 2*pi/omega
t_stop = 5*period
dt = period/20  # 20 time steps per period
theta = 1

T = TrialFunction(V)
v = TestFunction(V)
f = Constant(0)
a = rho*c*T*v*dx + theta*dt*kappa*\
    inner(nabla_grad(T), nabla_grad(v))*dx
L = (rho*c*T_prev*v + dt*f*v -
     (1-theta)*dt*kappa*inner(nabla_grad(T), nabla_grad(v)))*dx

A = assemble(a)
b = None  # variable used for memory savings in assemble calls
T = Function(V)   # unknown at the current time level
```

We could, alternatively, break `a` and `L` up in subexpressions and assemble a mass matrix and stiffness matrix, as exemplified in Section 1.3.3, to avoid assembly of `b` at every time level. This modification is straightforward and left as an exercise. The speed-up can be significant in 3D problems.

The time loop is very similar to what we have displayed in Section 1.3.2:

Python code

```
t = dt
while t <= t_stop:
    b = assemble(L, tensor=b)
    T_0.t = t
    bc.apply(A, b)
    solve(A, T.vector(), b)
    # visualization statements
    t += dt
    T_prev.assign(T)
```

The complete code in `sin_daD.py` contains several statements related to visualization and animation of the solution, both as a finite element field (`plot` calls) and as a curve in the vertical direction. The code also plots the exact analytical solution,

$$T(x,t) = T_R + T_A e^{ax}\sin(\omega t + ax), \quad a = \sqrt{\frac{\omega\varrho c}{2\kappa}}, \tag{1.123}$$

which is valid when $\kappa = \kappa_0 = \kappa_1$.

Implementing this analytical solution as a Python function taking scalars and numpy arrays as arguments requires a word of caution. A straightforward function like

Python code

```
def T_exact(x):
    a = sqrt(omega*rho*c/(2*kappa_0))
    return T_R + T_A*exp(a*x)*sin(omega*t + a*x)
```

will not work and result in an error message from UFL. The reason is that the names `exp` and `sin` are those imported by the `from dolfin import *` statement, and these names come from UFL and are

aimed at being used in variational forms. In the `T_exact` function where `x` may be a scalar or a `numpy` array, we therefore need to explicitly specify `numpy.exp` and `numpy.sin`:

Python code

```
def T_exact(x):
    a = sqrt(omega*rho*c/(2*kappa_0))
    return T_R + T_A*numpy.exp(a*x)*numpy.sin(omega*t + a*x)
```

The reader is encouraged to play around with the code and test out various parameter sets:

1. $T_R = 0$, $T_A = 1$, $\kappa_0 = \kappa_1 = 0.2$, $\varrho = c = 1$, $\omega = 2\pi$
2. $T_R = 0$, $T_A = 1$, $\kappa_0 = 0.2$, $\kappa_1 = 0.01$, $\varrho = c = 1$, $\omega = 2\pi$
3. $T_R = 0$, $T_A = 1$, $\kappa_0 = 0.2$, $\kappa_1 = 0.001$, $\varrho = c = 1$, $\omega = 2\pi$
4. $T_R = 10$ C, $T_A = 10$ C, $\kappa_0 = 2.3\ \mathrm{K^{-1}Ns^{-1}}$, $\kappa_1 = 100\ \mathrm{K^{-1}Ns^{-1}}$, $\varrho = 1500\ \mathrm{kg/m^3}$, $c = 1600\ \mathrm{Nm\,kg^{-1}K^{-1}}$, $\omega = 2\pi/24\ 1/\mathrm{h} = 7.27 \cdot 10^{-5}\ 1/\mathrm{s}$, $D = 1.5$ m
5. As above, but $\kappa_0 = 12.3\ \mathrm{K^{-1}Ns^{-1}}$ and $\kappa_1 = 10^4\ \mathrm{K^{-1}Ns^{-1}}$

Data set no. 4 is relevant for real temperature variations in the ground (not necessarily the large value of κ_1), while data set no. 5 exaggerates the effect of a large heat conduction contrast so that it becomes clearly visible in an animation.

1.4 Creating more complex domains

Up to now we have been very fond of the unit square as domain, which is an appropriate choice for initial versions of a PDE solver. The strength of the finite element method, however, is its ease of handling domains with complex shapes. This section shows some methods that can be used to create different types of domains and meshes.

Domains of complex shape must normally be constructed in separate preprocessor programs. Two relevant preprocessors are Triangle for 2D domains and NETGEN for 3D domains.

1.4.1 Built-in mesh generation tools

DOLFIN has a few tools for creating various types of meshes over domains with simple shape: `UnitInterval`, `UnitSquare`, `UnitCube`, `Interval`, `Rectangle`, `Box`, `UnitCircle`, and `UnitSphere`. Some of these names have been briefly met in previous sections. The hopefully self-explanatory code snippet below summarizes typical constructions of meshes with the aid of these tools:

Python code

```
# 1D domains
mesh = UnitInterval(20)     # 20 cells, 21 vertices
mesh = Interval(20, -1, 1)  # domain [-1,1]

# 2D domains (6x10 divisions, 120 cells, 77 vertices)
mesh = UnitSquare(6, 10)  # "right" diagonal is default
# The diagonals can be right, left or crossed
mesh = UnitSquare(6, 10, "left")
mesh = UnitSquare(6, 10, "crossed")

# Domain [0,3]x[0,2] with 6x10 divisions and left diagonals
mesh = Rectangle(0, 0, 3, 2, 6, 10, "left")
```

```
# 6x10x5 boxes in the unit cube, each box gets 6 tetrahedra:
mesh = UnitCube(6, 10, 5)

# Domain [-1,1]x[-1,0]x[-1,2] with 6x10x5 divisions
mesh = Box(-1, -1, -1, 1, 0, 2, 6, 10, 5)

# 10 divisions in radial directions
mesh = UnitCircle(10)
mesh = UnitSphere(10)
```

1.4.2 Transforming mesh coordinates

A mesh that is denser toward a boundary is often desired to increase accuracy in that region. Given a mesh with uniformly spaced coordinates $x_0, \ldots, x_{M-1}$ in $[a, b]$, the coordinate transformation $\xi = (x-a)/(b-a)$ maps x onto $\xi \in [0,1]$. A new mapping $\eta = \xi^s$, for some $s > 1$, stretches the mesh toward $\xi = 0$ ($x = a$), while $\eta = \xi^{1/s}$ makes a stretching toward $\xi = 1$ ($x = b$). Mapping the $\eta \in [0,1]$ coordinates back to $[a, b]$ gives new, stretched x coordinates,

$$\bar{x} = a + (b-a)\left((x-a)\,b-a\right)^s \tag{1.124}$$

toward $x = a$, or

$$\bar{x} = a + (b-a)\left(\frac{x-a}{b-a}\right)^{1/s} \tag{1.125}$$

toward $x = b$

One way of creating more complex geometries is to transform the vertex coordinates in a rectangular mesh according to some formula. Say we want to create a part of a hollow cylinder of Θ degrees, with inner radius a and outer radius b. A standard mapping from polar coordinates to Cartesian coordinates can be used to generate the hollow cylinder. Given a rectangle in $(\bar{x}, \bar{y})$ space such that $a \leqslant \bar{x} \leqslant b$ and $0 \leqslant \bar{y} \leqslant 1$, the mapping

$$\hat{x} = \bar{x}\cos(\Theta\bar{y}), \quad \hat{y} = \bar{x}\sin(\Theta\bar{y}), \tag{1.126}$$

takes a point in the rectangular $(\bar{x}, \bar{y})$ geometry and maps it to a point $(\hat{x}, \hat{y})$ in a hollow cylinder.

The corresponding Python code for first stretching the mesh and then mapping it onto a hollow cylinder looks as follows:

Python code

```
Theta = pi/2
a, b = 1, 5.0
nr = 10  # divisions in r direction
nt = 20  # divisions in theta direction
mesh = Rectangle(a, 0, b, 1, nr, nt, "crossed")

# First make a denser mesh towards r=a
x = mesh.coordinates()[:,0]
y = mesh.coordinates()[:,1]
s = 1.3

def denser(x, y):
    return [a + (b-a)*((x-a)/(b-a))**s, y]

x_bar, y_bar = denser(x, y)
xy_bar_coor = numpy.array([x_bar, y_bar]).transpose()
mesh.coordinates()[:] = xy_bar_coor
```

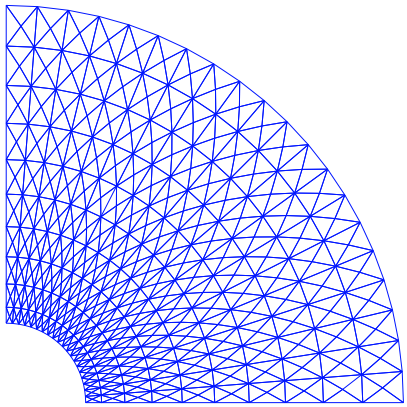


Figure 1.8: A hollow cylinder generated by mapping a rectangular mesh, stretched toward the left side.

```
plot(mesh, title="stretched mesh")

def cylinder(r, s):
    return [r*numpy.cos(Theta*s), r*numpy.sin(Theta*s)]

x_hat, y_hat = cylinder(x_bar, y_bar)
xy_hat_coor = numpy.array([x_hat, y_hat]).transpose()
mesh.coordinates()[:] = xy_hat_coor
plot(mesh, title="hollow cylinder")
interactive()
```

The result of calling `denser` and `cylinder` above is a list of two vectors, with the x and y coordinates, respectively. Turning this list into a `numpy` array object results in a $2 \times M$ array, M being the number of vertices in the mesh. However, `mesh.coordinates()` is by a convention an $M \times 2$ array so we need to take the transpose. The resulting mesh is displayed in Figure 1.8.

Setting boundary conditions in meshes created from mappings like the one illustrated above is most conveniently done by using a mesh function to mark parts of the boundary. The marking is easiest to perform before the mesh is mapped since one can then conceptually work with the sides in a pure rectangle.

1.5 Handling domains with different materials

Solving PDEs in domains made up of different materials is a frequently encountered task. In FEniCS, these kind of problems are handled by defining subdomains inside the domain. The subdomains may represent the various materials. We can thereafter define material properties through functions, known in FEniCS as *mesh functions*, that are piecewise constant in each subdomain. A simple example with two materials (subdomains) in 2D will demonstrate the basic steps in the process.

1.5.1 Working with two subdomains

Suppose we want to solve

$$\nabla \cdot [k(x,y)\nabla u(x,y)] = 0, \tag{1.127}$$

in a domain Ω consisting of two subdomains where k takes on a different value in each subdomain. For simplicity, yet without loss of generality, we choose for the current implementation the domain

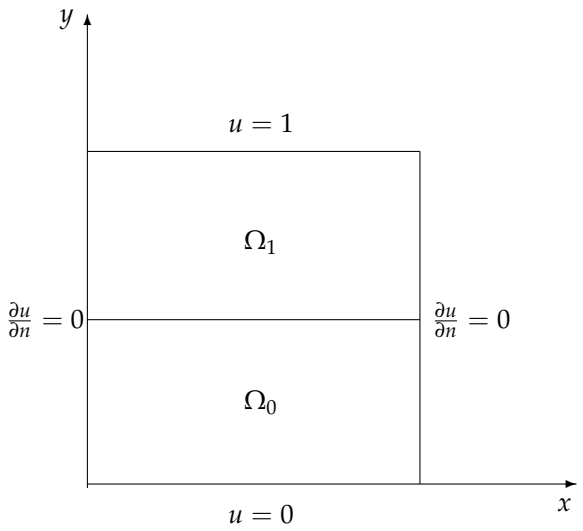


Figure 1.9: Sketch of a Poisson problem with a variable coefficient that is constant in each of the two subdomains Ω_0 and Ω_1.

$\Omega = [0,1] \times [0,1]$ and divide it into two equal subdomains, as depicted in Figure 1.9,

$$\Omega_0 = [0,1] \times [0,1/2], \quad \Omega_1 = [0,1] \times (1/2,1]. \tag{1.128}$$

We define $k(x,y) = k_0$ in Ω_0 and $k(x,y) = k_1$ in Ω_1, where $k_0 > 0$ and $k_1 > 0$ are given constants. As boundary conditions, we choose $u = 0$ at $y = 0$, $u = 1$ at $y = 1$, and $\partial u/\partial n = 0$ at $x = 0$ and $x = 1$. One can show that the exact solution is now given by

$$u(x,y) = \begin{cases} \frac{2yk_1}{k_0+k_1}, & y \leqslant 1/2 \\ \frac{(2y-1)k_0+k_1}{k_0+k_1}, & y \geqslant 1/2 \end{cases} \tag{1.129}$$

As long as the element boundaries coincide with the internal boundary $y = 1/2$, this piecewise linear solution should be exactly recovered by Lagrange elements of any degree. We use this property to verify the implementation.

Physically, the present problem may correspond to heat conduction, where the heat conduction in Ω_1 is ten times more efficient than in Ω_0. An alternative interpretation is flow in porous media with two geological layers, where the layers' ability to transport the fluid differs by a factor of 10.

1.5.2 Implementation

The new functionality in this subsection regards how to define the subdomains Ω_0 and Ω_1. For this purpose we need to use subclasses of class `SubDomain`, not only plain functions as we have used so far for specifying boundaries. Consider the boundary function

Python code

```
def boundary(x, on_boundary):
    tol = 1E-14
    return on_boundary and abs(x[0]) < tol
```

for defining the boundary $x = 0$. Instead of using such a stand-alone function, we can create an instance[2] of a subclass of `SubDomain`, which implements the `inside` method as an alternative to the

[2]The term *instance* means a Python object of a particular type (such as `SubDomain`, `Function`, `FunctionSpace`, etc.). Many use *instance* and *object* as interchangeable terms. In other computer programming languages one may also use the term *variable* for

`boundary` function:

Python code

```
class Boundary(SubDomain):
    def inside(x, on_boundary):
        tol = 1E-14
        return on_boundary and abs(x[0]) < tol

boundary = Boundary()
bc = DirichletBC(V, Constant(0), boundary)
```

A subclass of `SubDomain` with an `inside` method offers functionality for marking parts of the domain or the boundary. Now we need to define one class for the subdomain Ω_0 where $y \leqslant 1/2$ and another for the subdomain Ω_1 where $y \geqslant 1/2$:

Python code

```
class Omega0(SubDomain):
    def inside(self, x, on_boundary):
        return True if x[1] <= 0.5 else False

class Omega1(SubDomain):
    def inside(self, x, on_boundary):
        return True if x[1] >= 0.5 else False
```

Notice the use of `<=` and `>=` in both tests. For a cell to belong to, e.g., Ω_1, the `inside` method must return `True` for all the vertices `x` of the cell. So to make the cells at the internal boundary $y = 1/2$ belong to Ω_1, we need the test `x[1] >= 0.5`.

The next task is to use a `MeshFunction` to mark all cells in Ω_0 with the subdomain number 0 and all cells in Ω_1 with the subdomain number 1. Our convention is to number subdomains as $0, 1, 2, \ldots$.

A `MeshFunction` is a discrete function that can be evaluated at a set of so-called *mesh entities*. Examples of mesh entities are cells, facets, and vertices. A `MeshFunction` over cells is suitable to represent subdomains (materials), while a `MeshFunction` over facets is used to represent pieces of external or internal boundaries. Mesh functions over vertices can be used to describe continuous fields.

Since we need to define subdomains of Ω in the present example, we must make use of a `MeshFunction` over cells. The `MeshFunction` constructor is fed with three arguments: 1) the type of value: `"int"` for integers, `"uint"` for positive (unsigned) integers, `"double"` for real numbers, and `"bool"` for logical values; 2) a `Mesh` object, and 3) the topological dimension of the mesh entity in question: cells have topological dimension equal to the number of space dimensions in the PDE problem, and facets have one dimension lower. Alternatively, the constructor can take just a filename and initialize the `MeshFunction` from data in a file.

We start with creating a `MeshFunction` whose values are non-negative integers (`"uint"`) for numbering the subdomains. The mesh entities of interest are the cells, which have dimension 2 in a two-dimensional problem (1 in 1D, 3 in 3D). The appropriate code for defining the `MeshFunction` for two subdomains then reads

Python code

```
subdomains = MeshFunction("uint", mesh, 2)
# Mark subdomains with numbers 0 and 1
subdomain0 = Omega0()
subdomain0.mark(subdomains, 0)
subdomain1 = Omega1()
subdomain1.mark(subdomains, 1)
```

the same thing. We mostly use the well-known term *object* in this text.

Calling `subdomains.array()` returns a `numpy` array of the subdomain values. That is, `subdomain.array()[i]` is the subdomain value of cell number `i`. This array is used to look up the subdomain or material number of a specific element.

We need a function `k` that is constant in each subdomain Ω_0 and Ω_1. Since we want `k` to be a finite element function, it is natural to choose a space of functions that are constant over each element. The family of discontinuous Galerkin methods, in FEniCS denoted by `"DG"`, is suitable for this purpose. Since we want functions that are piecewise constant, the value of the degree parameter is zero:

Python code

```
V0 = FunctionSpace(mesh, "DG", 0)
k  = Function(V0)
```

To fill `k` with the right values in each element, we loop over all cells (the indices in `subdomain.array()`), extract the corresponding subdomain number of a cell, and assign the corresponding k value to the `k.vector()` array:

Python code

```
k_values = [1.5, 50]  # values of k in the two subdomains
for cell_no in range(len(subdomains.array())):
    subdomain_no = subdomains.array()[cell_no]
    k.vector()[cell_no] = k_values[subdomain_no]
```

Long loops in Python are known to be slow, so for large meshes it is preferable to avoid such loops and instead use *vectorized code*. Normally this implies that the loop must be replaced by calls to functions from the `numpy` library that operate on complete arrays (in efficient C code). The functionality we want in the present case is to compute an array of the same size as `subdomain.array()`, but where the value `i` of an entry in `subdomain.array()` is replaced by `k_values[i]`. Such an operation is carried out by the `numpy` function `choose`:

Python code

```
help = numpy.asarray(subdomains.array(), dtype=numpy.int32)
k.vector()[:] = numpy.choose(help, k_values)
```

The `help` array is required since `choose` cannot work with `subdomain.array()` because this array has elements of type `uint32`. We must therefore transform this array to an array `help` with standard `int32` integers.

Having the `k` function ready for finite element computations, we can proceed in the normal manner with defining essential boundary conditions, as in Section 1.1.14, and the $a(u, v)$ and $L(v)$ forms, as in Section 1.1.10. All the details can be found in the file `mat2_p2D.py`.

1.5.3 Multiple Neumann, Robin, and Dirichlet conditions

Let us go back to the model problem from Section 1.1.14 where we had both Dirichlet and Neumann conditions. The term `v*g*ds` in the expression for `L` implies a boundary integral over the complete boundary, or in FEniCS terms, an integral over all exterior facets. However, the contributions from the parts of the boundary where we have Dirichlet conditions are erased when the linear system is modified by the Dirichlet conditions. We would like, from an efficiency point of view, to integrate `v*g*ds` only over the parts of the boundary where we actually have Neumann conditions. And more importantly, in other problems one may have different Neumann conditions or other conditions like the Robin type condition. With the mesh function concept we can mark different parts of the boundary and integrate over specific parts. The same concept can also be used to treat multiple Dirichlet conditions. The forthcoming text illustrates how this is done.

Essentially, we still stick to the model problem from Section 1.1.14, but replace the Neumann condition at $y = 0$ by a *Robin condition*[3]:

$$-\frac{\partial u}{\partial n} = p(u - q), \tag{1.130}$$

where p and q are specified functions. Since we have prescribed a simple solution in our model problem, $u = 1 + x^2 + 2y^2$, we adjust p and q such that the condition holds at $y = 0$. This implies that $q = 1 + x^2 + 2y^2$ and p can be arbitrary (the normal derivative at $y = 0$: $\partial u/\partial n = -\partial u/\partial y = -4y = 0$).

Now we have four parts of the boundary: Γ_{N} which corresponds to the upper side $y = 1$, Γ_R which corresponds to the lower part $y = 0$, Γ_0 which corresponds to the left part $x = 0$, and Γ_1 which corresponds to the right part $x = 1$. The complete boundary-value problem reads

$$-\Delta u = -6 \text{ in } \Omega, \tag{1.131}$$
$$u = u_L \text{ on } \Gamma_0, \tag{1.132}$$
$$u = u_R \text{ on } \Gamma_1, \tag{1.133}$$
$$-\frac{\partial u}{\partial n} = p(u - q) \text{ on } \Gamma_R, \tag{1.134}$$
$$-\frac{\partial u}{\partial n} = g \text{ on } \Gamma_{\mathrm{N}}. \tag{1.135}$$

The involved prescribed functions are $u_L = 1 + 2y^2$, $u_R = 2 + 2y^2$, $q = 1 + x^2 + 2y^2$, p is arbitrary, and $g = -4y$.

Integration by parts of $-\int_\Omega v\Delta u \,\mathrm{d}x$ becomes as usual

$$-\int_\Omega v\Delta u \,\mathrm{d}x = \int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x - \int_{\partial\Omega} \frac{\partial u}{\partial n} v \,\mathrm{d}s. \tag{1.136}$$

The boundary integral vanishes on $\Gamma_0 \cup \Gamma_1$, and we split the parts over Γ_{N} and Γ_R since we have different conditions at those parts:

$$-\int_{\partial\Omega} v\frac{\partial u}{\partial n} \,\mathrm{d}s = -\int_{\Gamma_{\mathrm{N}}} v\frac{\partial u}{\partial n} \,\mathrm{d}s - \int_{\Gamma_R} v\frac{\partial u}{\partial n} \,\mathrm{d}s = \int_{\Gamma_{\mathrm{N}}} vg \,\mathrm{d}s + \int_{\Gamma_R} vp(u - q) \,\mathrm{d}s. \tag{1.137}$$

The weak form then becomes

$$\int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x + \int_{\Gamma_{\mathrm{N}}} gv \,\mathrm{d}s + \int_{\Gamma_R} p(u - q)v \,\mathrm{d}s = \int_\Omega fv \,\mathrm{d}x, \tag{1.138}$$

We want to write this weak form in the standard notation $a(u, v) = L(v)$, which requires that we identify all integrals with *both* u and v, and collect these in $a(u, v)$, while the remaining integrals with v and not u go into $L(v)$. The integral from the Robin condition must of this reason be split in two parts:

$$\int_{\Gamma_R} p(u - q)v \,\mathrm{d}s = \int_{\Gamma_R} puv \,\mathrm{d}s - \int_{\Gamma_R} pqv \,\mathrm{d}s. \tag{1.139}$$

[3]The Robin condition is most often used to model heat transfer to the surroundings and arise naturally from Newton's cooling law.

We then have

$$a(u,v) = \int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x + \int_{\Gamma_R} puv \,\mathrm{d}s, \tag{1.140}$$

$$L(v) = \int_\Omega fv \,\mathrm{d}x - \int_{\Gamma_N} gv \,\mathrm{d}s + \int_{\Gamma_R} pqv \,\mathrm{d}s. \tag{1.141}$$

A natural starting point for implementation is the file `stationary/poisson/dn2_p2D.py`. The new aspects are

1. definition of a mesh function over the boundary,
2. marking each side as a subdomain, using the mesh function,
3. splitting a boundary integral into parts.

Task 1 makes use of the `MeshFunction` object, but contrary to Section 1.5.2, this is not a function over cells, but a function over cell facets. The topological dimension of cell facets is one lower than the cell interiors, so in a two-dimensional problem the dimension becomes 1. In general, the facet dimension is given as `mesh.topology().dim()-1`, which we use in the code for ease of direct reuse in other problems. The construction of a `MeshFunction` object to mark boundary parts now reads

Python code

```
boundary_parts = \
  MeshFunction("uint", mesh, mesh.topology().dim()-1)
```

As in Section 1.5.2 we use a subclass of `SubDomain` to identify the various parts of the mesh function. Problems with domains of more complicated geometries may set the mesh function for marking boundaries as part of the mesh generation. In our case, the $y = 0$ boundary can be marked by

Python code

```
class LowerRobinBoundary(SubDomain):
    def inside(self, x, on_boundary):
        tol = 1E-14   # tolerance for coordinate comparisons
        return on_boundary and abs(x[1]) < tol

Gamma_R = LowerRobinBoundary()
Gamma_R.mark(boundary_parts, 0)
```

The code for the $y = 1$ boundary is similar and is seen in `dnr_p2D`.

The Dirichlet boundaries are marked similarly, using subdomain number 2 for Γ_0 and 3 for Γ_1:

Python code

```
class LeftBoundary(SubDomain):
    def inside(self, x, on_boundary):
        tol = 1E-14   # tolerance for coordinate comparisons
        return on_boundary and abs(x[0]) < tol

Gamma_0 = LeftBoundary()
Gamma_0.mark(boundary_parts, 2)

class RightBoundary(SubDomain):
    def inside(self, x, on_boundary):
        tol = 1E-14   # tolerance for coordinate comparisons
        return on_boundary and abs(x[0] - 1) < tol

Gamma_1 = RightBoundary()
Gamma_1.mark(boundary_parts, 3)
```

Specifying the `DirichletBC` objects may now make use of the mesh function (instead of a `SubDomain` subclass object) and an indicator for which subdomain each condition should be applied to:

Python code

```
u_L = Expression("1 + 2*x[1]*x[1]")
u_R = Expression("2 + 2*x[1]*x[1]")
bcs = [DirichletBC(V, u_L, boundary_parts, 2),
       DirichletBC(V, u_R, boundary_parts, 3)]
```

Some functions need to be defined before we can go on with the `a` and `L` of the variational problem:

Python code

```
g = Expression("-4*x[1]")
q = Expression("1 + x[0]*x[0] + 2*x[1]*x[1]")
p = Constant(100)  # arbitrary function can go here
u = TrialFunction(V)
v = TestFunction(V)
f = Constant(-6.0)
```

The new aspect of the variational problem is the two distinct boundary integrals. Having a mesh function over exterior cell facets (our `boundary_parts` object), where subdomains (boundary parts) are numbered as $0, 1, 2, \ldots$, the special symbol `ds(0)` implies integration over subdomain (part) 0, `ds(1)` denotes integration over subdomain (part) 1, and so on. The idea of multiple `ds`-type objects generalizes to volume integrals too: `dx(0)`, `dx(1)`, etc., are used to integrate over subdomain 0, 1, etc., inside Ω.

The variational problem can be defined as

Python code

```
a = inner(nabla_grad(u), nabla_grad(v))*dx + p*u*v*ds(0)
L = f*v*dx - g*v*ds(1) + p*q*v*ds(0)
```

For the `ds(0)` and `ds(1)` symbols to work we must obviously connect them (or `a` and `L`) to the mesh function marking parts of the boundary. This is done by a certain keyword argument to the `assemble` function:

Python code

```
A = assemble(a, exterior_facet_domains=boundary_parts)
b = assemble(L, exterior_facet_domains=boundary_parts)
```

Then essential boundary conditions are enforced, and the system can be solved in the usual way:

Python code

```
for bc in bcs: bc.apply(A, b)
u = Function(V)
U = u.vector()
solve(A, U, b)
```

The complete code is in the `dnr_p2D.py` file in the `stationary/poisson` directory.

1.6 More examples

Many more topics could be treated in a FEniCS tutorial, e.g., how to solve systems of PDEs, how to work with mixed finite element methods, how to create more complicated meshes and mark boundaries, and how to create more advanced visualizations. However, to limit the size of this tutorial,

the examples end here. There are, fortunately, a rich set of FEniCS demos. The FEniCS documentation explains a collection of PDE solvers in detail: the Poisson equation, the mixed formulation for the Poisson equation, the Biharmonic equation, the equations of hyperelasticity, the Cahn-Hilliard equation, and the incompressible Navier–Stokes equations. Both Python and C++ versions of these solvers are explained. An eigenvalue solver is also documented. In the `dolfin/demo` directory of the DOLFIN source code tree you can find programs for these and many other examples, including the advection-diffusion equation, the equations of elastodynamics, a reaction-diffusion equation, various finite element methods for the Stokes problem, discontinuous Galerkin methods for the Poisson and advection-diffusion equations, and an eigenvalue problem arising from electromagnetic waveguide problem with Nédélec elements. There are also numerous demos on how to apply various functionality in FEniCS, e.g., mesh refinement and error control, moving meshes (for ALE methods), computing functionals over subsets of the mesh (such as lift and drag on bodies in flow), and creating separate subdomain meshes from a parent mesh.

The project CBC.Solve (`https://launchpad.net/cbc.solve`) offers more complete PDE solvers for the Navier–Stokes equations (Chapter 29), the equations of hyperelasticity (Chapter 27), fluid–structure interaction (Chapter 29), viscous mantle flow (Chapter 31), and the bidomain model of electrophysiology. Another project, CBC.RANS (`https://launchpad.net/cbc.rans`), offers an environment for very flexible and easy implementation of Navier–Stokes solvers and turbulence (Mortensen et al., 2011b,a). For example, CBC.RANS contains an elliptic relaxation model for turbulent flow involving 18 nonlinear PDEs. FEniCS proved to be an ideal environment for implementing such complicated PDE models. The easy construction of systems of nonlinear PDEs in CBC.RANS has been further generalized to simplify the implementation of large systems of nonlinear PDEs in general. The functionality is found in the CBC.PDESys package (`https://launchpad.net/cbcpdesys`).

1.7 Miscellaneous topics

1.7.1 Glossary

Below we explain some key terms used in this tutorial.

FEniCS: name of a software suite composed of many individual software components (see `fenicsproject.org`). Some components are DOLFIN and Viper, explicitly referred to in this tutorial. Others are FFC and FIAT, heavily used by the programs appearing in this tutorial, but never explicitly used from the programs.

DOLFIN: a FEniCS component, more precisely a C++ library, with a Python interface, for performing important actions in finite element programs. DOLFIN makes use of many other FEniCS components and many external software packages.

Viper: a FEniCS component for quick visualization of finite element meshes and solutions.

UFL: a FEniCS component implementing the *unified form language* for specifying finite element forms in FEniCS programs. The definition of the forms, typically called `a` and `L` in this tutorial, must have legal UFL syntax. The same applies to the definition of functionals (see Section 1.1.11).

Class (Python): a programming construction for creating objects containing a set of variables and functions. Most types of FEniCS objects are defined through the class concept.

Instance (Python): an object of a particular type, where the type is implemented as a class. For instance, `mesh = UnitInterval(10)` creates an instance of class `UnitInterval`, which is reached by the name `mesh`. (Class `UnitInterval` is actually just an interface to a corresponding C++ class in the DOLFIN C++ library.)

Class method (Python): a function in a class, reached by dot notation: `instance_name.method_name`.

`self` parameter (Python): required first parameter in class methods, representing a particular object of the class. Used in method definitions, but never in calls to a method. For example, if `method(self, x)` is the definition of `method` in a class `Y`, `method` is called as `y.method(x)`, where `y` is an instance of class `Y`. In a call like `y.method(x)`, `method` is invoked with `self=y`.

Class attribute (Python): a variable in a class, reached by dot notation: `instance_name.attribute_name`.

1.7.2 Overview of objects and functions

Most classes in FEniCS have an explanation of the purpose and usage that can be seen by using the general documentation command `pydoc` for Python objects. You can type

Output

```
pydoc dolfin.X
```

to look up documentation of a Python class `X` from the DOLFIN library (`X` can be `UnitSquare`, `Function`, `FunctionSpace`, etc.). Below is an overview of the most important classes and functions in FEniCS programs, in the order they typically appear within programs.

`UnitSquare(nx, ny)`: generate mesh over the unit square $[0,1] \times [0,1]$ using `nx` divisions in x direction and `ny` divisions in y direction. Each of the `nx*ny` squares are divided into two cells of triangular shape.

`UnitInterval`, `UnitCube`, `UnitCircle`, `UnitSphere`, `Interval`, `Rectangle`, and `Box`: generate mesh over domains of simple geometric shape, see Section 1.4.

`FunctionSpace(mesh, element_type, degree)`: a function space defined over a mesh, with a given element type (e.g., `"Lagrange"` or `"DG"`), with basis functions as polynomials of a specified degree.

`Expression(formula, p1=v1, p2=v2, ...)`: a scalar- or vector-valued function, given as a mathematical expression `formula` (string) written in C++ syntax. The spatial coordinates in the expression are named `x[0]`, `x[1]`, and `x[2]`, while time and other physical parameters can be represented as symbols `p1`, `p2`, etc., with corresponding values `v1`, `v2`, etc., initialized through keyword arguments. These parameters become attributes, whose values can be modified when desired.

`Function(V)`: a scalar- or vector-valued finite element field in the function space `V`. If `V` is a `FunctionSpace` object, `Function(V)` becomes a scalar field, and with `V` as a `VectorFunctionSpace` object, `Function(V)` becomes a vector field.

`SubDomain`: class for defining a subdomain, either a part of the boundary, an internal boundary, or a part of the domain. The programmer must subclass `SubDomain` and implement the `inside(self, x, on_boundary)` function (see Section 1.1.3) for telling whether a point `x` is inside the subdomain or not.

`Mesh`: class for representing a finite element mesh, consisting of cells, vertices, and optionally faces, edges, and facets.

`MeshFunction`: tool for marking parts of the domain or the boundary. Used for variable coefficients ("material properties", see Section 1.5.1) or for boundary conditions (see Section 1.5.3).

`DirichletBC(V, value, where)`: specification of Dirichlet (essential) boundary conditions via a function space `V`, a function `value(x)` for computing the value of the condition at a point `x`, and a specification `where` of the boundary, either as a `SubDomain` subclass instance, a plain function, or as a `MeshFunction` instance. In the latter case, a 4th argument is provided to describe which subdomain number that describes the relevant boundary.

`TrialFunction(V)`: define a trial function on a space `V` to be used in a variational form to represent the unknown in a finite element problem.

`TestFunction(V)`: define a test function on a space `V` to be used in a variational form.

`assemble(X)`: assemble a matrix, a right-hand side, or a functional, given a from `X` written with UFL syntax.

`assemble_system(a, L, bcs)`: assemble the matrix and the right-hand side from a bilinear (`a`) and linear (`L`) form written with UFL syntax. The `bcs` parameter holds one or more `DirichletBC` objects.

`LinearVariationalProblem(a, L, u, bcs)`: define a variational problem, given a bilinear (`a`) and linear (`L`) form, written with UFL syntax, and one or more `DirichletBC` objects stored in `bcs`.

`LinearVariationalSolver(problem)`: create solver object for a a linear variational problem object (`problem`).

`solve(A, U, b)`: solve a linear system with `A` as coefficient matrix (`Matrix` object), `U` as unknown (`Vector` object), and `b` as right-hand side (`Vector` object). Usually, `U = u.vector()`, where `u` is a `Function` object representing the unknown finite element function of the problem, while `A` and `b` are computed by calls to `assemble` or `assemble_system`.

`plot(q)`: quick visualization of a mesh, function, or mesh function `q`, using the Viper component in FEniCS.

`interpolate(func, V)`: interpolate a formula or finite element function `func` onto the function space `V`.

`project(func, V)`: project a formula or finite element function `func` onto the function space `V`.

1.7.3 User-defined functions

When defining a function in terms of a mathematical expression inside a string formula, e.g.,

Python code

```
myfunc = Expression("sin(x[0])*cos(x[1])")
```

the expression contained in the first argument will be turned into a C++ function and compiled to gain efficiency. Therefore, the syntax used in the expression must be valid C++ syntax. Most Python syntax for mathematical expressions are also valid C++ syntax, but power expressions make an exception: `p**a` must be written as `pow(p,a)` in C++ (this is also an alternative Python syntax). The following mathematical functions can be used directly in C++ expressions when defining `Expression` objects: `cos`, `sin`, `tan`, `acos`, `asin`, `atan`, `atan2`, `cosh`, `sinh`, `tanh`, `exp`, `frexp`, `ldexp`, `log`, `log10`, `modf`, `pow`, `sqrt`, `ceil`, `fabs`, `floor`, and `fmod`. Moreover, the number π is available as the symbol `pi`. All the listed functions are taken from the `cmath` C++ header file, and one may hence consult documentation of `cmath` for more information on the various functions.

1.7.4 Linear solvers and preconditioners

The following solution methods for linear systems can be accessed in FEniCS programs:

Name	Method
`"lu"`	sparse LU factorization (Gaussian elim.)
`"cholesky"`	sparse Cholesky factorization
`"cg"`	Conjugate gradient method
`"gmres"`	Generalized minimal residual method
`"bicgstab"`	Biconjugate gradient stabilized method
`"minres"`	Minimal residual method
`"tfqmr"`	Transpose-free quasi-minimal residual method
`"richardson"`	Richardson method

Possible choices of preconditioners include

Name	Method
`"none"`	No preconditioner
`"ilu"`	Incomplete LU factorization
`"icc"`	Incomplete Cholesky factorization
`"jacobi"`	Jacobi iteration
`"bjacobi"`	Block Jacobi iteration
`"sor"`	Successive over-relaxation
`"amg"`	Algebraic multigrid (BoomerAMG or ML)
`"additive_schwarz"`	Additive Schwarz
`"hypre_amg"`	Hypre algebraic multigrid (BoomerAMG)
`"hypre_euclid"`	Hypre parallel incomplete LU factorization
`"hypre_parasails"`	Hypre parallel sparse approximate inverse
`"ml_amg"`	ML algebraic multigrid

Many of the choices listed above are only offered by a specific backend, so setting the backend appropriately is necessary for being able to choose a desired linear solver or preconditioner.

An up-to-date list of the available solvers and preconditioners in FEniCS can be produced by

Python code

```
list_linear_solver_methods()
list_krylov_solver_preconditioners()
```

1.7.5 Installing FEniCS

The FEniCS software components are available for Linux, Windows and Mac OS X platforms. Detailed information on how to get FEniCS running on such machines are available at the `fenicsproject.org` website. Here are just some quick descriptions and recommendations by the author.

To make the installation of FEniCS as painless and reliable as possible, the reader is strongly recommended to use Ubuntu Linux[4]. Any standard PC can easily be equipped with Ubuntu Linux, which may live side by side with either Windows or Mac OS X or another Linux installation. Basically, you download Ubuntu from `http://www.ubuntu.com/getubuntu/download`, burn the file on a CD or copy it to a memory stick, reboot the machine with the CD or memory stick, and answer some usually straightforward questions (if necessary). On Windows, Wubi is a tool that automatically installs Ubuntu on the machine. Just give a user name and password for the Ubuntu installation, and Wubi performs the rest. The graphical user interface (GUI) of Ubuntu is quite similar to both Windows 7 and Mac OS X, but to be efficient when doing science with FEniCS this author recommends to run programs in a terminal window and write them in a text editor like Emacs or Vim. You can employ integrated development environment such as Eclipse, but intensive FEniCS developers and users tend to find terminal windows and plain text editors more user friendly.

Instead of making it possible to boot your machine with the Linux Ubuntu operating system, you can run Ubuntu in a separate window in your existing operation system. There are several solutions to chose among: the free *VirtualBox* and *VMWare Player*, or the commercial tools *VMWare Fusion* and *Parallels* (just search for the names to download the programs).

Once the Ubuntu window is up and running, FEniCS is painlessly installed by

[4]Even though Mac users now can get FEniCS by a one-click install, I recommend using Ubuntu on Mac, unless you have high Unix competence and much experience with compiling and linking C++ libraries on Mac OS X.

Bash code

```
sudo apt-get install fenics
```

Sometimes the FEniCS software in a standard Ubuntu installation lacks some recent features and bug fixes. Visiting the detailed download page on `fenicsproject.org` and copying a few Unix commands is all you have to do to install a newer version of the software.

1.7.6 Books on the finite element method

There are a large number of books on the finite element method. The books typically fall in either of two categories: the abstract mathematical version of the method and the engineering "structural analysis" formulation. FEniCS builds heavily on concepts in the abstract mathematical exposition. An easy-to-read book, which provides a good general background for using FEniCS, is Gockenbach (2006). The book Donea and Huerta (2003) has a similar style, but aims at readers with interest in fluid flow problems. Hughes (1987) is also highly recommended, especially for those interested in solid mechanics and heat transfer applications.

Readers with background in the engineering "structural analysis" version of the finite element method may find Bickford (1994) as an attractive bridge over to the abstract mathematical formulation that FEniCS builds upon. Those who have a weak background in differential equations in general should consult a more fundamental book, and Eriksson et al. (1996) is a very good choice. On the other hand, FEniCS users with a strong background in mathematics and interest in the mathematical properties of the finite element method, will appreciate the texts Brenner and Scott (2008), Braess (2007), Ern and Guermond (2004), Quarteroni and Valli (2008), or Ciarlet (2002).

1.7.7 Books on Python

Two very popular introductory books on Python are "Learning Python" (Lutz, 2007) and "Practical Python" (Hetland, 2002). More advanced and comprehensive books include "Programming Python" (Lutz, 2006), and "Python Cookbook" (Martelli and Ascher, 2005) and "Python in a Nutshell" (Martelli, 2006). The web page `http://wiki.python.org/moin/PythonBooks` lists numerous additional books. Very few texts teach Python in a mathematical and numerical context, but the references Langtangen (2008, 2011); Kiusalaas (2009) are exceptions.

Acknowledgments. The author is very thankful to Johan Hake, Anders Logg, Kent-Andre Mardal, and Kristian Valen-Sendstad for promptly answering all my questions about FEniCS functionality and for implementing all my requests. I will in particular thank Professor Douglas Arnold for very valuable feedback on the text. Øystein Sørensen pointed out a lot of typos and contributed with many helpful comments. Many errors and typos were also reported by Mauricio Angeles, Ida Drøsdal, Hans Ekkehard Plesser, and Marie Rognes. Ekkehard Ellmann as well as two anonymous reviewers provided a series of suggestions and improvements.

Part I

Methodology

2 *The finite element method*

By Robert C. Kirby and Anders Logg

The finite element method has emerged as a universal method for the solution of differential equations. Much of the success of the finite element method can be attributed to its generality and elegance, allowing a wide range of differential equations from all areas of science to be analyzed and solved within a common framework. Another contributing factor to the success of the finite element method is the flexibility of formulation, allowing the properties of the discretization to be controlled by the choice of approximating finite element spaces.

In this chapter, we review the finite element method and summarize some basic concepts and notation used throughout this book. In the coming chapters, we discuss these concepts in more detail, with a particular focus on the implementation and automation of the finite element method as part of the FEniCS Project.

2.1 A simple model problem

In 1813, Siméon Denis Poisson published in *Bulletin de la société philomatique* his famous equation as a correction of an equation published earlier by Pierre-Simon Laplace. Poisson's equation is a second-order partial differential equation stating that the negative Laplacian $-\Delta u$ of some unknown field $u = u(x)$ is equal to a given function $f = f(x)$ on a domain $\Omega \subset \mathbb{R}^d$, possibly amended by a set of boundary conditions for the solution u on the boundary $\partial\Omega$ of Ω:

$$\begin{aligned} -\Delta u &= f \quad && \text{in } \Omega, \\ u &= u_0 \quad && \text{on } \Gamma_{\mathrm{D}} \subset \partial\Omega, \\ -\partial_n u &= g \quad && \text{on } \Gamma_{\mathrm{N}} \subset \partial\Omega. \end{aligned} \tag{2.1}$$

The Dirichlet boundary condition $u = u_0$ signifies a prescribed value for the unknown u on a subset Γ_{D} of the boundary, and the Neumann boundary condition $-\partial_n u = g$ signifies a prescribed value for the (negative) normal derivative of u on the remaining boundary $\Gamma_{\mathrm{N}} = \partial\Omega \setminus \Gamma_{\mathrm{D}}$. Poisson's equation is a simple model for gravity, electromagnetism, heat transfer, fluid flow, and many other physical processes. It also appears as the basic building block in a large number of more complex physical models, including the Navier–Stokes equations which we return to in Chapters 20, 21, 22, 23, 24, 25, 28 and 29.

To derive Poisson's equation (2.1), we may consider a model for the temperature u in a body occupying a domain Ω subject to a heat source f. Letting $\sigma = \sigma(x)$ denote heat flux, it follows by conservation of energy that the outflow of energy over the boundary $\partial\omega$ of any test volume $\omega \subset \Omega$

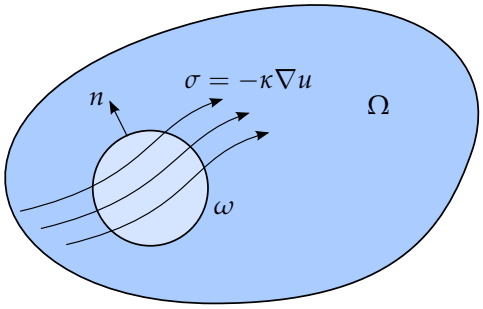


Figure 2.1: Poisson's equation is a simple consequence of balance of energy in an arbitrary test volume $\omega \subset \Omega$.

must be balanced by the energy emitted by the heat source f:

$$\int_{\partial\omega} \sigma \cdot n \, \mathrm{d}s = \int_{\omega} f \, \mathrm{d}x. \tag{2.2}$$

Integrating by parts, we find that

$$\int_{\omega} \nabla \cdot \sigma \, \mathrm{d}x = \int_{\omega} f \, \mathrm{d}x. \tag{2.3}$$

Since (2.3) holds for all test volumes $\omega \subset \Omega$, it follows that $\nabla \cdot \sigma = f$ throughout Ω (with suitable regularity assumptions on σ and f). If we now make the assumption that the heat flux σ is proportional to the negative gradient of the temperature u (Fourier's law),

$$\sigma = -\kappa\nabla u, \tag{2.4}$$

we arrive at the following system of equations:

$$\begin{aligned} \nabla \cdot \sigma &= f \quad \text{in } \Omega, \\ \sigma + \nabla u &= 0 \quad \text{in } \Omega, \end{aligned} \tag{2.5}$$

where we have assumed that the heat conductivity is $\kappa = 1$. Replacing σ in the first of these equations by $-\nabla u$, we arrive at Poisson's equation (2.1). Note that one may as well arrive at the system of first-order equations (2.5) by introducing $\sigma = -\nabla u$ as an auxiliary variable in the second-order equation (2.1). We also note that the Dirichlet and Neumann boundary conditions in (2.1) correspond to prescribed values for the temperature and heat flux, respectively.

2.2 Finite element discretization

2.2.1 Discretizing Poisson's equation

To discretize Poisson's equation (2.1) by the finite element method, we first multiply by a test function v and integrate by parts to obtain

$$\int_{\Omega} \nabla u \cdot \nabla v \, \mathrm{d}x - \int_{\partial\Omega} \partial_n u \, v \, \mathrm{d}s = \int_{\Omega} f v \, \mathrm{d}x. \tag{2.6}$$

Letting the test function v vanish on the Dirichlet boundary Γ_{D} where the solution u is known, we arrive at the following classical variational problem: find $u \in V$ such that

$$\int_{\Omega} \nabla u \cdot \nabla v \, \mathrm{d}x = \int_{\Omega} f v \, \mathrm{d}x - \int_{\Gamma_{\mathrm{N}}} g v \, \mathrm{d}s \quad \forall\, v \in \hat{V}. \tag{2.7}$$

The test space $\hat{V}$ is defined by

$$\hat{V} = \{v \in H^1(\Omega) : v = 0 \text{ on } \Gamma_{\mathrm{D}}\}, \tag{2.8}$$

and the trial space V contains members of $\hat{V}$ shifted by the Dirichlet condition:

$$V = \{v \in H^1(\Omega) : v = u_0 \text{ on } \Gamma_{\mathrm{D}}\}. \tag{2.9}$$

We may now discretize Poisson's equation by restricting the variational problem (2.7) to a pair of discrete spaces: find $u_h \in V_h \subset V$ such that

$$\int_\Omega \nabla u_h \cdot \nabla v \,\mathrm{d}x = \int_\Omega fv \,\mathrm{d}x - \int_{\Gamma_{\mathrm{N}}} gv \,\mathrm{d}s \quad \forall\, v \in \hat{V}_h \subset \hat{V}. \tag{2.10}$$

We note here that the Dirichlet condition $u = u_0$ on Γ_{D} enters directly into the definition of the trial space V_h (it is an *essential* boundary condition), whereas the Neumann condition $-\partial_n u = g$ on Γ_{N} enters into the variational problem (it is a *natural* boundary condition).

To solve the discrete variational problem (2.10), we must construct a suitable pair of discrete trial and test spaces V_h and $\hat{V}_h$. We return to this issue below, but assume for now that we have a basis $\{\phi_j\}_{j=1}^N$ for V_h and a basis $\{\hat{\phi}_i\}_{i=1}^N$ for $\hat{V}_h$. Here, N denotes the dimension of the spaces V_h and $\hat{V}_h$. We may then make an Ansatz for u_h in terms of the basis functions of the trial space,

$$u_h(x) = \sum_{j=1}^N U_j \phi_j(x), \tag{2.11}$$

where $U \in \mathbb{R}^N$ is the vector of degrees of freedom to be computed. Inserting this into (2.10) and varying the test function v over the basis functions of the discrete test space $\hat{V}_h$, we obtain

$$\sum_{j=1}^N U_j \int_\Omega \nabla \phi_j \cdot \nabla \hat{\phi}_i \,\mathrm{d}x = \int_\Omega f\hat{\phi}_i \,\mathrm{d}x - \int_{\Gamma_{\mathrm{N}}} g\hat{\phi}_i \,\mathrm{d}s, \quad i = 1, 2, \ldots, N. \tag{2.12}$$

We may thus compute the finite element solution $u_h = \sum_{j=1}^N U_j \phi_j$ by solving the linear system

$$AU = b, \tag{2.13}$$

where

$$\begin{aligned} A_{ij} &= \int_\Omega \nabla \phi_j \cdot \nabla \hat{\phi}_i \,\mathrm{d}x, \\ b_i &= \int_\Omega f\hat{\phi}_i \,\mathrm{d}x - \int_{\Gamma_{\mathrm{N}}} g\hat{\phi}_i \,\mathrm{d}s. \end{aligned} \tag{2.14}$$

2.2.2 *Discretizing the first-order system*

We may similarly discretize the first-order system (2.5) by multiplying the first equation by a test function v and the second equation by a test function τ. Summing up and integrating by parts, we find that

$$\int_\Omega (\nabla \cdot \sigma)\, v + \sigma \cdot \tau - u \nabla \cdot \tau \,\mathrm{d}x + \int_{\partial\Omega} u\tau \cdot n \,\mathrm{d}s = \int_\Omega fv \,\mathrm{d}x \quad \forall\, (v, \tau) \in \hat{V}. \tag{2.15}$$

The normal flux $\sigma \cdot n = g$ is known on the Neumann boundary Γ_{N} so we may take $\tau \cdot n = 0$ on Γ_{N}. Inserting the value for u on the Dirichlet boundary Γ_{D}, we arrive at the following variational problem:

find $(u, \sigma) \in V$ such that

$$\int_\Omega (\nabla \cdot \sigma)\, v + \sigma \cdot \tau - u \nabla \cdot \tau \,\mathrm{d}x = \int_\Omega f v \,\mathrm{d}x - \int_{\Gamma_\mathrm{D}} u_0 \tau \cdot n \,\mathrm{d}s \quad \forall\, (v, \tau) \in \hat{V}. \tag{2.16}$$

A suitable choice of trial and test spaces is

$$\begin{aligned} V &= \{(v, \tau) : v \in L^2(\Omega), \tau \in H(\mathrm{div}, \Omega), \tau \cdot n = g \text{ on } \Gamma_\mathrm{N}\}, \\ \hat{V} &= \{(v, \tau) : v \in L^2(\Omega), \tau \in H(\mathrm{div}, \Omega), \tau \cdot n = 0 \text{ on } \Gamma_\mathrm{N}\}. \end{aligned} \tag{2.17}$$

Note that the variational problem (2.16) differs from the variational problem (2.7) in that the Dirichlet condition $u = u_0$ on Γ_D enters into the variational formulation (it is now a natural boundary condition), whereas the Neumann condition $\sigma \cdot n = g$ on Γ_N enters into the definition of the trial space V (it is now an essential boundary condition).

As above, we restrict the variational problem to a pair of discrete trial and test spaces $V_h \subset V$ and $\hat{V}_h \subset \hat{V}$ and make an Ansatz for the finite element solution of the form

$$(u_h, \sigma_h) = \sum_{j=1}^{N} U_j (\phi_j, \psi_j), \tag{2.18}$$

where $\{(\phi_j, \psi_j)\}_{j=1}^N$ is a basis for the trial space V_h. Typically, either ϕ_j or ψ_j will vanish, so that the basis is really the tensor product of a basis for the L^2 space with a basis for the $H(\mathrm{div})$ space. We thus obtain a linear system for the degrees of freedom $U \in \mathbb{R}^N$ by solving a linear system $AU = b$, where now

$$\begin{aligned} A_{ij} &= \int_\Omega (\nabla \cdot \psi_j)\, \hat{\phi}_i + \psi_j \cdot \hat{\psi}_i - \phi_j \nabla \cdot \hat{\psi}_i \,\mathrm{d}x, \\ b_i &= \int_\Omega f \hat{\phi}_i \,\mathrm{d}x - \int_{\Gamma_\mathrm{D}} u_0\, \hat{\psi}_i \cdot n \,\mathrm{d}s. \end{aligned} \tag{2.19}$$

The finite element discretization (2.19) is an example of a *mixed method*. Such formulations require some care in selecting spaces that discretize the different function spaces, here L^2 and $H(\mathrm{div})$, in a compatible way. Stable discretizations must satisfy the so-called *inf–sup* or Ladyzhenskaya–Babuška–Brezzi (LBB) condition(s). This theory explains why many of the finite element spaces for mixed methods seem complicated compared to those for standard methods. In Chapter 3 below, we give several examples of such finite element spaces.

2.3 Finite element abstract formalism

2.3.1 Linear problems

We saw above that the finite element solution of Poisson's equation (2.1) or (2.5) can be obtained by restricting an infinite-dimensional (continuous) variational problem to a finite-dimensional (discrete) variational problem and solving a linear system.

To formalize this, we consider a general linear variational problem written in the following canonical form: find $u \in V$ such that

$$a(u, v) = L(v) \quad \forall\, v \in \hat{V}, \tag{2.20}$$

where V is the trial space and $\hat{V}$ is the test space. We thus express the variational problem in terms of

a *bilinear form a* and a *linear form* (functional) L:

$$\begin{aligned} a &: V \times \hat{V} \to \mathbb{R}, \\ L &: \hat{V} \to \mathbb{R}. \end{aligned} \tag{2.21}$$

As above, we discretize the variational problem (2.20) by restricting to a pair of discrete trial and test spaces: find $u_h \in V_h \subset V$ such that

$$a(u_h, v) = L(v) \quad \forall\, v \in \hat{V}_h \subset \hat{V}. \tag{2.22}$$

To solve the discrete variational problem (2.22), we make an Ansatz of the form

$$u_h = \sum_{j=1}^{N} U_j \phi_j, \tag{2.23}$$

and take $v = \hat{\phi}_i$ for $i = 1, 2, \ldots, N$. As before, $\{\phi_j\}_{j=1}^N$ is a basis for the discrete trial space V_h and $\{\hat{\phi}_i\}_{i=1}^N$ is a basis for the discrete test space $\hat{V}_h$. It follows that

$$\sum_{j=1}^{N} U_j\, a(\phi_j, \hat{\phi}_i) = L(\hat{\phi}_i), \quad i = 1, 2, \ldots, N. \tag{2.24}$$

The degrees of freedom U of the finite element solution u_h may then be computed by solving a linear system $AU = b$, where

$$\begin{aligned} A_{ij} &= a(\phi_j, \hat{\phi}_i), \quad i, j = 1, 2, \ldots, N, \\ b_i &= L(\hat{\phi}_i). \end{aligned} \tag{2.25}$$

2.3.2 Nonlinear problems

We also consider nonlinear variational problems written in the following canonical form: find $u \in V$ such that

$$F(u; v) = 0 \quad \forall\, v \in \hat{V}, \tag{2.26}$$

where now $F : V \times \hat{V} \to \mathbb{R}$ is a *semilinear* form, linear in the argument(s) subsequent to the semicolon. As above, we discretize the variational problem (2.26) by restricting to a pair of discrete trial and test spaces: find $u_h \in V_h \subset V$ such that

$$F(u_h; v) = 0 \quad \forall\, v \in \hat{V}_h \subset \hat{V}. \tag{2.27}$$

The finite element solution $u_h = \sum_{j=1}^N U_j \phi_j$ may then be computed by solving a nonlinear system of equations,

$$b(U) = 0, \tag{2.28}$$

where $b : \mathbb{R}^N \to \mathbb{R}^N$ and

$$b_i(U) = F(u_h; \hat{\phi}_i), \quad i = 1, 2, \ldots, N. \tag{2.29}$$

To solve the nonlinear system (2.28) by Newton's method or some variant of Newton's method, we compute the Jacobian $A = b'$. We note that if the semilinear form F is differentiable in u, then the

entries of the Jacobian A are given by

$$A_{ij}(u_h) = \frac{\partial b_i(U)}{\partial U_j} = \frac{\partial}{\partial U_j} F(u_h; \hat{\phi}_i) = F'(u_h; \hat{\phi}_i) \frac{\partial u_h}{\partial U_j} = F'(u_h; \hat{\phi}_i)\, \phi_j \equiv F'(u_h; \phi_j, \hat{\phi}_i). \tag{2.30}$$

In each Newton iteration, we must then evaluate (assemble) the matrix A and the vector b, and update the solution vector U by

$$U^{k+1} = U^k - \delta U^k, \quad k = 0, 1, \ldots, \tag{2.31}$$

where δU^k solves the linear system

$$A(u_h^k)\, \delta U^k = b(u_h^k). \tag{2.32}$$

We note that for each fixed u_h, $a = F'(u_h; \cdot, \cdot)$ is a bilinear form and $L = F(u_h; \cdot)$ is a linear form. In each Newton iteration, we thus solve a linear variational problem of the canonical form (2.20): find $\delta u \in V_{h,0}$ such that

$$F'(u_h; \delta u, v) = F(u_h; v) \quad \forall\, v \in \hat{V}_h, \tag{2.33}$$

where $V_{h,0} = \{v - w : v, w \in V_h\}$. Discretizing (2.33) as in Section 2.3.1, we recover the linear system (2.32).

Example 2.1 (Nonlinear Poisson equation) *As an example, consider the following nonlinear Poisson equation:*

$$\begin{aligned} -\nabla \cdot ((1+u)\nabla u) &= f \quad \text{in } \Omega, \\ u &= 0 \quad \text{on } \partial\Omega. \end{aligned} \tag{2.34}$$

Multiplying (2.34) with a test function v and integrating by parts, we obtain

$$\int_\Omega ((1+u)\nabla u) \cdot \nabla v \,\mathrm{d}x = \int_\Omega f v \,\mathrm{d}x, \tag{2.35}$$

which is a nonlinear variational problem of the form (2.26), with

$$F(u; v) = \int_\Omega ((1+u)\nabla u) \cdot \nabla v \,\mathrm{d}x - \int_\Omega f v \,\mathrm{d}x. \tag{2.36}$$

Linearizing the semilinear form F around $u = u_h$, we obtain

$$F'(u_h; \delta u, v) = \int_\Omega (\delta u \nabla u_h) \cdot \nabla v \,\mathrm{d}x + \int_\Omega ((1+u_h)\nabla \delta u) \cdot \nabla v \,\mathrm{d}x. \tag{2.37}$$

We may thus compute the entries of the Jacobian matrix $A(u_h)$ by

$$A_{ij}(u_h) = F'(u_h; \phi_j, \hat{\phi}_i) = \int_\Omega (\phi_j \nabla u_h) \cdot \nabla \hat{\phi}_i \,\mathrm{d}x + \int_\Omega ((1+u_h)\nabla \phi_j) \cdot \nabla \hat{\phi}_i \,\mathrm{d}x. \tag{2.38}$$

2.4 Finite element function spaces

In the above discussion, we assumed that we could construct discrete subspaces $V_h \subset V$ of infinite-dimensional function spaces. A central aspect of the finite element method is the construction of such subspaces by patching together local function spaces defined by a set of *finite elements*. We here give a general overview of the construction of finite element function spaces and return in Chapters 3 and 4 to the construction of specific function spaces as subsets of H^1, $H(\mathrm{curl})$, $H(\mathrm{div})$ and L^2.

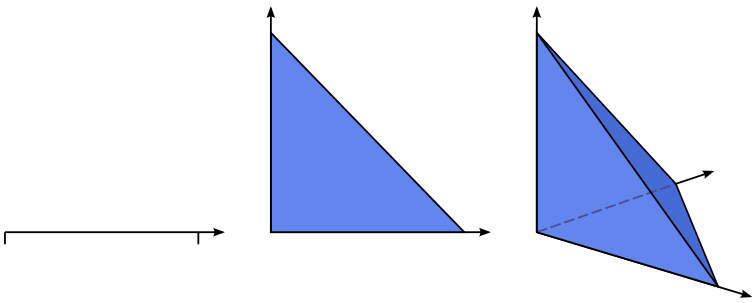


Figure 2.2: Examples of finite element cells in one, two and three space dimensions.

2.4.1 The mesh

To define V_h, we first partition the domain Ω into a finite set of cells $\mathcal{T}_h = \{T\}$ with disjoint interiors such that

$$\cup_{T \in \mathcal{T}_h} T = \Omega. \tag{2.39}$$

Together, these cells form a *mesh* of the domain Ω. The cells are typically simple polygonal shapes like intervals, triangles, quadrilaterals, tetrahedra or hexahedra as shown in Figure 2.2. But other shapes are possible, in particular curved cells to capture the boundary of a non-polygonal domain correctly.

2.4.2 The finite element definition

Once a domain Ω has been partitioned into cells, one may define a local function space $\mathcal{V}$ on each cell T and use these local function spaces to build the global function space V_h. A cell T together with a local function space $\mathcal{V}$ and a set of rules for describing the functions in $\mathcal{V}$ is called a *finite element*. This definition was first formalized by Ciarlet (1976) and it remains the standard formulation today (Brenner and Scott, 2008). The formal definition reads as follows: a finite element is a triple $(T, \mathcal{V}, \mathcal{L})$, where

- the domain T is a bounded, closed subset of $\mathbb{R}^d$ (for $d = 1, 2, 3, \dots$) with nonempty interior and piecewise smooth boundary;
- the space $\mathcal{V} = \mathcal{V}(T)$ is a finite dimensional function space on T of dimension n;
- the set of degrees of freedom (nodes) $\mathcal{L} = \{\ell_1, \ell_2, \dots, \ell_n\}$ is a basis for the dual space $\mathcal{V}'$; that is, the space of bounded linear functionals on $\mathcal{V}$.

As an example, consider the standard linear Lagrange finite element on the triangle in Figure 2.3. The cell T is given by the triangle and the space $\mathcal{V}$ is given by the space of first degree polynomials on T (a space of dimension three). As a basis for $\mathcal{V}'$, we may take point evaluation at the three vertices of T; that is,

$$\begin{aligned} \ell_i &: \mathcal{V} \to \mathbb{R}, \\ \ell_i(v) &= v(x^i), \end{aligned} \tag{2.40}$$

for $i = 1, 2, 3$ where x^i is the coordinate of the ith vertex. To check that this is indeed a finite element, we need to verify that $\mathcal{L}$ is a basis for $\mathcal{V}'$. This is equivalent to the unisolvence of $\mathcal{L}$; that is, if $v \in \mathcal{V}$ and $\ell_i(v) = 0$ for all ℓ_i, then $v = 0$ (Brenner and Scott, 2008). For the linear Lagrange triangle, we note

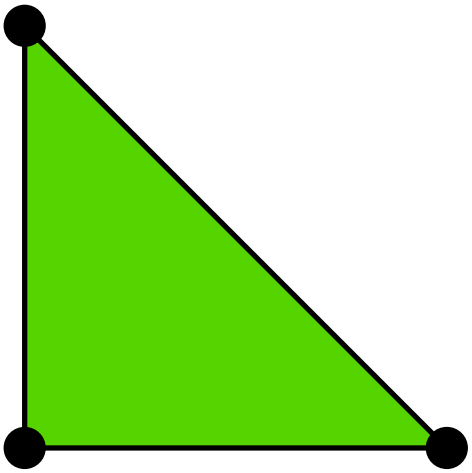


Figure 2.3: The degrees of freedom of the linear Lagrange (Courant) triangle are given by point evaluation at the three vertices of the triangle.

that if v is zero at each vertex, then v must be zero everywhere, since a plane is uniquely determined by its values at three non-collinear points. It follows that the linear Lagrange triangle is indeed a finite element. In general, determining the unisolvence of $\mathcal{L}$ may be non-trivial.

2.4.3 The nodal basis

Expressing finite element solutions in V_h in terms of basis functions for the local function spaces $\mathcal{V}$ may be greatly simplified by introducing a *nodal basis* for $\mathcal{V}$. A nodal basis $\{\phi_i\}_{i=1}^n$ for $\mathcal{V}$ is a basis for $\mathcal{V}$ that satisfies

$$\ell_i(\phi_j) = \delta_{ij}, \quad i, j = 1, 2, \ldots, n. \tag{2.41}$$

It follows that any $v \in \mathcal{V}$ may be expressed by

$$v = \sum_{i=1}^{n} \ell_i(v)\phi_i. \tag{2.42}$$

In particular, any function v in $\mathcal{V}$ for the linear Lagrange triangle is given by $v = \sum_{i=1}^{3} v(x^i)\phi_i$. In other words, the expansion coefficients of any function v may be obtained by evaluating the linear functionals in $\mathcal{L}$ at v. We shall therefore interchangeably refer to both the expansion coefficients U of u_h and the linear functionals of $\mathcal{L}$ as the *degrees of freedom*.

Example 2.2 (Nodal basis for the linear Lagrange simplices) *The nodal basis for the linear Lagrange interval with vertices at $x^1 = 0$ and $x^2 = 1$ is given by*

$$\phi_1(x) = 1 - x, \quad \phi_2(x) = x. \tag{2.43}$$

The nodal basis for the linear Lagrange triangle with vertices at $x^1 = (0, 0)$, $x^2 = (1, 0)$ and $x^3 = (0, 1)$ is given by

$$\phi_1(x) = 1 - x_1 - x_2, \quad \phi_2(x) = x_1, \quad \phi_3(x) = x_2. \tag{2.44}$$

The nodal basis for the linear Lagrange tetrahedron with vertices at $x^1 = (0, 0, 0)$, $x^2 = (1, 0, 0)$, $x^3 = (0, 1, 0)$ and $x^4 = (0, 0, 1)$ is given by

$$\begin{aligned} \phi_1(x) &= 1 - x_1 - x_2 - x_3, & \phi_2(x) &= x_1, \\ \phi_3(x) &= x_2, & \phi_4(x) &= x_3. \end{aligned} \tag{2.45}$$

For any finite element $(T, \mathcal{V}, \mathcal{L})$, the nodal basis may be computed by solving a linear system of size $n \times n$. To see this, let $\{\psi_i\}_{i=1}^n$ be any basis (the *prime* basis) for $\mathcal{V}$. Such a basis is easy to construct if $\mathcal{V}$ is a full polynomial space or may otherwise be computed by a singular-value decomposition or a Gram–Schmidt procedure; see Kirby (2004). We may then make an Ansatz for the nodal basis in terms of the prime basis:

$$\phi_j = \sum_{k=1}^{n} \alpha_{jk} \psi_k, \quad j = 1, 2, \ldots, n. \tag{2.46}$$

Inserting this into (2.41), we find that

$$\sum_{k=1}^{n} \alpha_{jk} \ell_i(\psi_k) = \delta_{ij}, \quad i, j = 1, 2, \ldots, n. \tag{2.47}$$

In other words, the coefficients α expanding the nodal basis functions in the prime basis may be computed by solving the linear system

$$B\alpha^\top = I, \tag{2.48}$$

where $B_{ij} = \ell_i(\psi_j)$.

2.4.4 The local-to-global mapping

Now, to define a global function space $V_h = \text{span}\{\phi_i\}_{i=1}^N$ on Ω from a given set $\{(T, \mathcal{V}_T, \mathcal{L}_T)\}_{T \in \mathcal{T}_h}$ of finite elements, we also need to specify how the local function spaces are patched together. We do this by specifying for each cell $T \in \mathcal{T}_h$ a *local-to-global mapping*:

$$\iota_T : [1, n_T] \to [1, N]. \tag{2.49}$$

This mapping specifies how the local degrees of freedom $\mathcal{L}_T = \{\ell_i^T\}_{i=1}^{n_T}$ are mapped to global degrees of freedom $\mathcal{L} = \{\ell_i\}_{i=1}^N$. More precisely, the global degrees of freedom are defined by

$$\ell_{\iota_T(i)}(v) = \ell_i^T(v|_T), \quad i = 1, 2, \ldots, n_T, \tag{2.50}$$

for any $v \in V_h$. Thus, each local degree of freedom $\ell_i^T \in \mathcal{L}_T$ corresponds to a global degree of freedom $\ell_{\iota_T(i)} \in \mathcal{L}$ determined by the local-to-global mapping ι_T. As we shall see, the local-to-global mapping together with the choice of degrees of freedom determine the continuity of the global function space V_h.

For standard continuous piecewise linears, one may define the local-to-global mapping by simply mapping each local vertex number i for $i = 1, 2, 3$ to the corresponding global vertex number $\iota_T(i)$. For continuous piecewise quadratics, one can base the local-to-global mapping on global vertex and edge numbers as illustrated in Figure 2.4 for a simple mesh consisting of two triangles.

2.4.5 The global function space

One may now define the global function space V_h as the set of functions on Ω satisfying the following pair of conditions. We first require that

$$v|_T \in \mathcal{V}_T \quad \forall\, T \in \mathcal{T}_h; \tag{2.51}$$

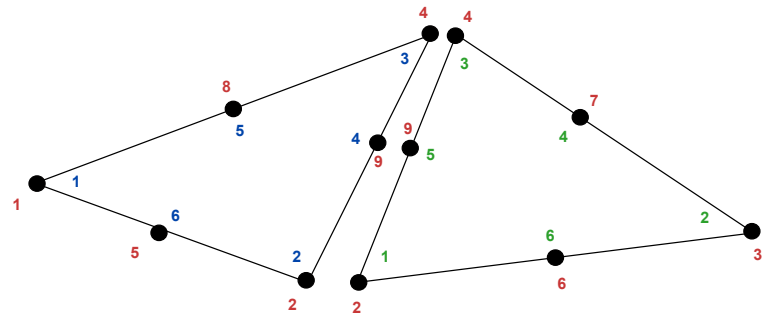


Figure 2.4: Local-to-global mapping for a simple mesh consisting of two triangles. The six local degrees of freedom of the left triangle (T) are mapped to the global degrees of freedom $\iota_T(i) = 1, 2, 4, 9, 8, 5$ for $i = 1, 2, \ldots, 6$, and the six local degrees of freedom of the right triangle (T') are mapped to $\iota_{T'}(i) = 2, 3, 4, 7, 9, 6$ for $i = 1, 2, \ldots, 6$.

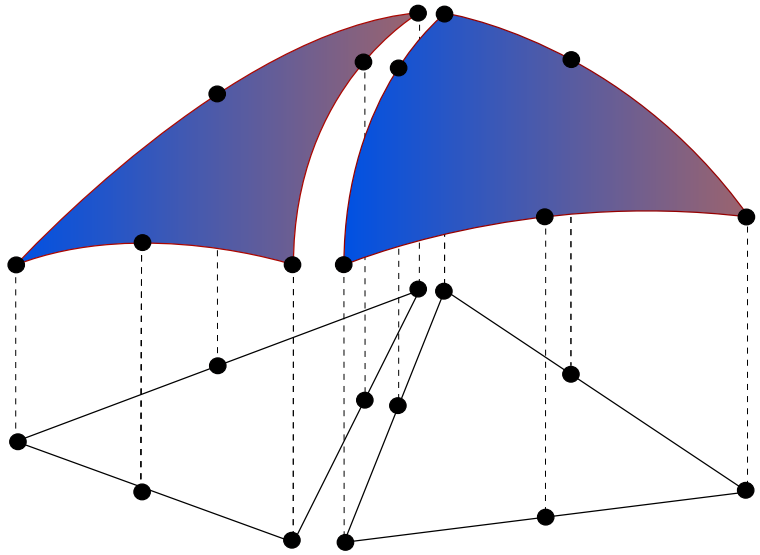


Figure 2.5: Patching together a pair of quadratic local function spaces on a pair of cells (T, T') to form a global continuous piecewise quadratic function space on $\Omega = T \cup T'$.

that is, the restriction of v to each cell T lies in the local function space $\mathcal{V}_T$. Second, we require that for any pair of cells $(T, T') \in \mathcal{T}_h \times \mathcal{T}_h$ and any pair $(i, i') \in [1, n_T] \times [1, n_{T'}]$ satisfying

$$\iota_T(i) = \iota_{T'}(i'), \tag{2.52}$$

it holds that

$$\ell_i^T(v|_T) = \ell_{i'}^{T'}(v|_{T'}). \tag{2.53}$$

In other words, if two local degrees of freedom ℓ_i^T and $\ell_{i'}^{T'}$ are mapped to the same global degree of freedom, then they must agree for each function $v \in V_h$. Here, $v|_T$ denotes (the continuous extension of the) restriction of v to the interior of T. This is illustrated in Figure 2.5 for the space of continuous piecewise quadratics obtained by patching together two quadratic Lagrange triangles.

Note that by this construction, the functions in V_h are undefined on cell boundaries, unless the constraints (2.53) force the functions in V_h to be continuous on cell boundaries. However, this is usually not a problem, since we can perform all operations on the restrictions of functions to the local cells.

The local-to-global mapping together with the choice of degrees of freedom determine the continuity of the global function space V_h. For the linear Lagrange triangle, choosing the degrees of freedom as point evaluation at the vertices ensures that all functions in V_h must be continuous at the two vertices of the common edge of any pair of adjacent triangles, and therefore along the entire common edge. It follows that the functions in V_h are continuous throughout the domain Ω. As a

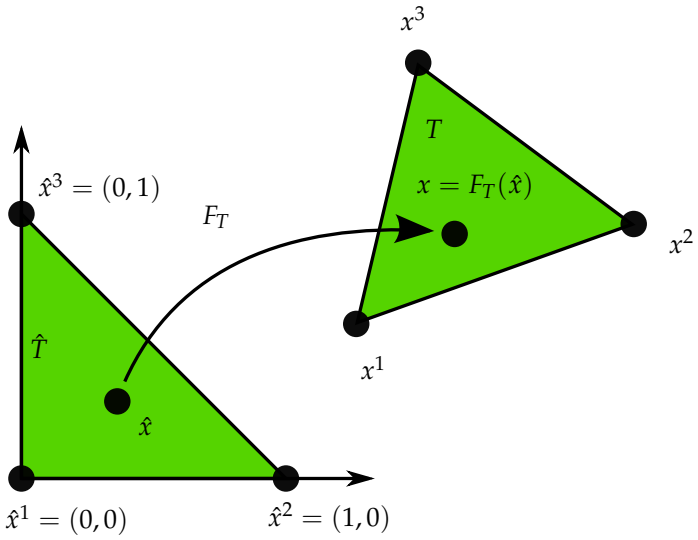


Figure 2.6: The (affine) map F_T from a reference cell $\hat{T}$ to a cell $T \in \mathcal{T}_h$.

consequence, the space of piecewise linears generated by the Lagrange triangle is H^1-conforming; that is, $V_h \subset H^1(\Omega)$.

One may also consider degrees of freedom defined by point evaluation at the midpoint of each edge. This is the so-called Crouzeix–Raviart triangle. The corresponding global Crouzeix–Raviart space V_h is consequently continuous only at edge midpoints. The Crouzeix–Raviart triangle is an example of an H^1-*nonconforming* element; that is, the function space V_h constructed from a set of Crouzeix–Raviart elements is not a subspace of H^1. Other choices of degrees of freedom may ensure continuity of normal components, like for the $H(\mathrm{div})$-conforming Brezzi–Douglas–Marini elements, or tangential components, as for the $H(\mathrm{curl})$-conforming Nédélec elements. In Chapter 3, other examples of elements are given which ensure different kinds of continuity by the choice of degrees of freedom and local-to-global mapping.

2.4.6 *The mapping from the reference element*

As we have seen, the global function space V_h may be described by a mesh $\mathcal{T}_h$, a set of finite elements $\{(T, \mathcal{V}_T, \mathcal{L}_T)\}_{T\in\mathcal{T}_h}$ and a set of local-to-global mappings $\{\iota_T\}_{T\in\mathcal{T}_h}$. We may simplify this description further by introducing a *reference finite element* $(\hat{T}, \hat{\mathcal{V}}, \hat{\mathcal{L}})$, where $\hat{\mathcal{L}} = \{\hat{\ell}_1, \hat{\ell}_2, \ldots, \hat{\ell}_{\hat{n}}\}$, and a set of invertible mappings $\{F_T\}_{T\in\mathcal{T}_h}$ that map the reference cell $\hat{T}$ to the cells of the mesh:

$$T = F_T(\hat{T}) \quad \forall\, T \in \mathcal{T}_h. \tag{2.54}$$

This is illustrated in Figure 2.6. Note that $\hat{T}$ is generally not part of the mesh.

For function spaces discretizing H^1 as in (2.7), the mapping F_T is typically *affine*; that is, F_T can be written in the form $F_T(\hat{x}) = A_T\hat{x} + b_T$ for some matrix $A_T \in \mathbb{R}^{d\times d}$ and some vector $b_T \in \mathbb{R}^d$, or else *isoparametric*, in which case the components of F_T are functions in $\hat{\mathcal{V}}$. For function spaces discretizing $H(\mathrm{div})$ like in (2.16) or $H(\mathrm{curl})$, the appropriate mappings are the contravariant and covariant Piola mappings which preserve normal and tangential components, respectively; see Rognes et al. (2009). For simplicity, we restrict the following discussion to the case when F_T is affine or isoparametric.

For each cell $T \in \mathcal{T}_h$, the mapping F_T generates a function space on T given by

$$\mathcal{V}_T = \{v : v = \hat{v} \circ F_T^{-1}, \quad \hat{v} \in \hat{\mathcal{V}}\}; \tag{2.55}$$

that is, each function $v = v(x)$ may be expressed as $v(x) = \hat{v}(F_T^{-1}(x)) = \hat{v} \circ F_T^{-1}(x)$ for some $\hat{v} \in \hat{\mathcal{V}}$.

The mapping F_T also generates a set of degrees of freedom $\mathcal{L}_T$ on $\mathcal{V}_T$ given by

$$\mathcal{L}_T = \{\ell_i : \ell_i(v) = \hat{\ell}_i(v \circ F_T), \quad i = 1, 2, \ldots, \hat{n}\}. \tag{2.56}$$

The mappings $\{F_T\}_{T \in \mathcal{T}_h}$ thus generate from the reference finite element $(\hat{T}, \hat{\mathcal{V}}, \hat{\mathcal{L}})$ a set of finite elements $\{(T, \mathcal{V}_T, \mathcal{L}_T)\}_{T \in \mathcal{T}_h}$ given by

$$\begin{aligned} T &= F_T(\hat{T}), \\ \mathcal{V}_T &= \{v : v = \hat{v} \circ F_T^{-1}, \quad \hat{v} \in \hat{\mathcal{V}}\}, \\ \mathcal{L}_T &= \{\ell_i : \ell_i(v) = \hat{\ell}_i(v \circ F_T), \quad i = 1, 2, \ldots, \hat{n} = n_T\}. \end{aligned} \tag{2.57}$$

By this construction, we also obtain the nodal basis functions $\{\phi_i^T\}_{i=1}^{n_T}$ on T from a set of nodal basis functions $\{\hat{\phi}_i\}_{i=1}^{\hat{n}}$ on the reference element satisfying $\hat{\ell}_i(\hat{\phi}_j) = \delta_{ij}$. To see this, we let $\phi_i^T = \hat{\phi}_i \circ F_T^{-1}$ for $i = 1, 2, \ldots, n_T$ and find that

$$\ell_i^T(\phi_j^T) = \hat{\ell}_i(\phi_j^T \circ F_T) = \hat{\ell}_i(\hat{\phi}_j \circ F_T^{-1} \circ F_T) = \hat{\ell}_i(\hat{\phi}_j) = \delta_{ij}, \tag{2.58}$$

so $\{\phi_i^T\}_{i=1}^{n_T}$ is a nodal basis for $\mathcal{V}_T$.

We may therefore define the function space V_h by specifying a mesh $\mathcal{T}_h$, a reference finite element $(\hat{T}, \hat{\mathcal{V}}, \hat{\mathcal{L}})$, a set of local-to-global mappings $\{\iota_T\}_{T \in \mathcal{T}_h}$ and a set of mappings $\{F_T\}_{T \in \mathcal{T}_h}$ from the reference cell $\hat{T}$. Note that in general, the mappings need not be of the same type for all cells T and not all finite elements need to be generated from the same reference finite element. In particular, one could employ a different (higher-degree) isoparametric mapping for cells on a curved boundary.

The above construction is valid for so-called affine-equivalent elements (Brenner and Scott, 2008) like the family of H^1-conforming Lagrange finite elements. A similar construction is possible for $H(\mathrm{div})$- and $H(\mathrm{curl})$-conforming elements, like the Brezzi–Douglas–Marini and Nédélec elements, where an appropriate Piola mapping must be used to map the basis functions (while an affine map may still be used to map the geometry). However, not all finite elements may be generated from a reference finite element using this simple construction. For example, this construction fails for the family of Hermite finite elements (Ciarlet, 2002; Brenner and Scott, 2008).

2.5 Finite element solvers

Finite elements provide a powerful methodology for discretizing differential equations, but solving the resulting algebraic systems also presents a challenge, even for linear systems. Good solvers must handle the sparsity and possible ill-conditioning of the algebraic system, and also scale well on parallel computers. The linear solve is a fundamental operation not only in linear problems, but also within each iteration of a nonlinear solve via Newton's method, an eigenvalue solve, or time-stepping.

A classical approach that has been revived recently is direct solution, based on Gaussian elimination. Thanks to techniques enabling parallel scalability and recognizing block structure, packages such as UMFPACK (Davis, 2004) and SuperLU (Li, 2005) have made direct methods competitive for quite large problems.

The 1970s and 1980s saw the advent of modern iterative methods. These grew out of classical iterative methods such as relaxation methods and the conjugate gradient iteration of Hestenes and Stiefel (1952). These techniques can use much less memory than direct methods and are easier to parallelize.

Multigrid methods (Brandt, 1977; Wesseling, 1992) use relaxation techniques on a hierarchy of meshes to solve elliptic equations, typically for symmetric problems, in nearly linear time. However, they require a hierarchy of meshes that may not always be available. This motivated the introduction of *algebraic* multigrid methods (AMG) that mimic mesh coarsening, working only on the matrix entries. Successful AMG distributions include the Hypre package (Falgout and Yang, 2002) and the ML package distributed as part of Trilinos (Heroux et al., 2005).

Krylov methods such as conjugate gradients and GMRES (Saad and Schultz, 1986) generate a sequence of approximations converging to the solution of the linear system. These methods are based only on the matrix–vector product. The performance of these methods is significantly improved by use of *preconditioners*, which transform the linear system

$$AU = b \tag{2.59}$$

into

$$P^{-1}AU = P^{-1}b, \tag{2.60}$$

which is known as left preconditioning. The preconditioner P^{-1} may also be applied from the right by recognizing that $AU = (AP^{-1})(PU)$ and solving the modified system for the matrix AP^{-1}, followed by an additional solve to obtain U from the solution PU. To ensure good convergence, the preconditioner P^{-1} should be a good approximation of A^{-1}. Some preconditioners are strictly algebraic, meaning they only use information available from the entries of A. Classical relaxation methods such as Gauss–Seidel may be used as preconditioners, as can so-called incomplete factorizations (Manteuffel, 1980; Axelsson, 1986; Saad, 1994). Multigrid, whether geometric or algebraic, also can serve as a powerful preconditioner. Other kinds of preconditioners require special knowledge about the differential equation being solved and may require new matrices modeling related physical processes. Such methods are sometimes called *physics-based* preconditioners. An automated system, such as FEniCS, provides an interesting opportunity to assist with the development and implementation of these powerful but less widely used methods.

Fortunately, many of the methods discussed here are included in modern libraries such as PETSc (Balay et al., 2004) and Trilinos (Heroux et al., 2005). FEniCS typically interacts with the solvers discussed here through these packages and so mainly need to be aware of the various methods at a high level, such as when the various methods are appropriate and how to access them.

2.6 Finite element error estimation and adaptivity

The error $e = u - u_h$ in a computed finite element solution u_h approximating the exact solution u of (2.20) may be estimated either *a priori* or *a posteriori*.

A priori error estimates express the error in terms of the regularity of the exact (unknown) solution and may give useful information about the order of convergence of a finite element method. *A posteriori* error estimates express the error in terms of computable quantities like the residual and (possibly) the solution of an auxiliary dual problem, as described below.

2.6.1 A priori *error analysis*

We consider the linear variational problem (2.20). We first assume that the bilinear form a and the linear form L are continuous (bounded); that is, there exists a constant $C > 0$ such that

$$\begin{aligned} a(v, w) &\leqslant C\|v\|_V\|w\|_V, \\ L(v) &\leqslant C\|v\|_V, \end{aligned} \tag{2.61}$$

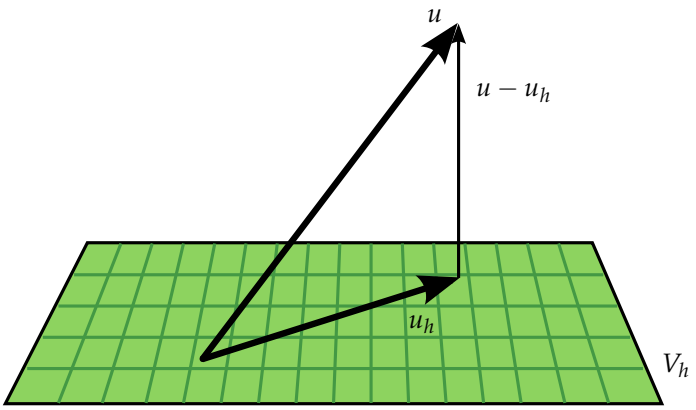


Figure 2.7: If the bilinear form a is symmetric, then the finite element solution $u_h \in V_h \subset V$ is the a-projection of $u \in V$ onto the subspace V_h and is consequently the best possible approximation of u in the subspace V_h (in the norm defined by the bilinear form a). This follows by the Galerkin orthogonality $\langle u - u_h, v\rangle_a \equiv a(u - u_h, v) = 0$ for all $v \in V_h$.

for all $v, w \in V$. For simplicity, we assume in this section that $V = \hat{V}$ is a Hilbert space. For (2.1), this corresponds to the case of homogeneous Dirichlet boundary conditions and $V = H_0^1(\Omega)$. Extensions to the general case $V \neq \hat{V}$ are possible; see for example Oden and Demkowicz (1996). We further assume that the bilinear form a is coercive (V-elliptic); that is, there exists a constant $\alpha > 0$ such that

$$a(v, v) \geqslant \alpha \|v\|_V^2, \tag{2.62}$$

for all $v \in V$. It then follows by the Lax–Milgram theorem (Lax and Milgram, 1954) that there exists a unique solution $u \in V$ to the variational problem (2.20).

To derive an *a priori* error estimate for the approximate solution u_h defined by the discrete variational problem (2.22), we first note that

$$a(u - u_h, v) = a(u, v) - a(u_h, v) = L(v) - L(v) = 0 \tag{2.63}$$

for all $v \in V_h \subset V$ (the Galerkin orthogonality). By the coercivity and continuity of the bilinear form a, we find that

$$\begin{aligned} \alpha \|u - u_h\|_V^2 &\leqslant a(u - u_h, u - u_h) = a(u - u_h, u - v) + a(u_h - u, v - u_h) \\ &= a(u - u_h, u - v) \leqslant C \|u - u_h\|_V \, \|u - v\|_V. \end{aligned} \tag{2.64}$$

for all $v \in V_h$. It follows that

$$\|u - u_h\|_V \leqslant \frac{C}{\alpha} \|u - v\|_V \quad \forall\, v \in V_h. \tag{2.65}$$

The estimate (2.65) is referred to as Cea's lemma. We note that when the bilinear form a is symmetric, it is also an inner product. We may then take $\|v\|_V = \sqrt{a(v, v)}$ and $C = \alpha = 1$. In this case, u_h is the a-projection onto V_h and Cea's lemma states that

$$\|u - u_h\|_V \leqslant \|u - v\|_V \quad \forall\, v \in V_h; \tag{2.66}$$

that is, u_h is the best possible solution of the variational problem (2.20) in the subspace V_h. This is illustrated in Figure 2.7.

Cea's lemma together with a suitable interpolation estimate now yields an *a priori* error estimate

for u_h. By choosing $v = \pi_h u$, where $\pi_h : V \to V_h$ is an interpolation operator into V_h, we find that

$$\|u - u_h\|_V \leqslant \frac{C}{\alpha}\|u - \pi_h u\|_V \leqslant \frac{CC_i}{\alpha}\|h^p D^{q+1} u\|_{L^2}, \tag{2.67}$$

where C_i is an interpolation constant and the values of p and q depend on the accuracy of interpolation and the definition of $\|\cdot\|_V$. For the solution of Poisson's equation in $V = H_0^1$, we have $C = \alpha = 1$ and $p = q = 1$.

2.6.2 A posteriori *error analysis*

Energy norm error estimates. The continuity and coercivity of the bilinear form a also allow the derivation of an *a posteriori* error estimate. This type of error estimate is obtained by relating the size of the error to the size of the (weak) residual $r : \hat{V} \to \mathbb{R}$ defined by

$$r(v) = L(v) - a(u_h, v). \tag{2.68}$$

Note that the weak residual is formally related to the *strong residual* $R \in \hat{V}'$ by $r(v) = \langle R, v\rangle$ for all $v \in \hat{V}$.

We first note that the V-norm of the error $e = u - u_h$ is equivalent to the V'-norm of the residual r. To see this, note that by the continuity of the bilinear form a, we have

$$r(v) = L(v) - a(u_h, v) = a(u, v) - a(u_h, v) = a(u - u_h, v) \leqslant C\|u - u_h\|_V \, \|v\|_V. \tag{2.69}$$

Furthermore, by coercivity, we find that

$$\alpha\|u - u_h\|_V^2 \leqslant a(u - u_h, u - u_h) = a(u, u - u_h) - a(u_h, u - u_h) = L(u - u_h) - a(u_h, u - u_h) = r(u - u_h). \tag{2.70}$$

It follows that

$$\alpha\|u - u_h\|_V \leqslant \|r\|_{V'} \leqslant C\|u - u_h\|_V, \tag{2.71}$$

where $\|r\|_{V'} = \sup_{v\in V, v\neq 0} r(v)/\|v\|_V$.

The estimates (2.67) and (2.71) are sometimes referred to as *energy norm* error estimates. This is the case when the bilinear form a is symmetric and thus defines an inner product. One may then take $\|v\|_V = \sqrt{a(v, v)}$ and $C = \alpha = 1$. In this case, it follows that

$$\eta \equiv \|e\|_V = \|r\|_{V'}. \tag{2.72}$$

The term energy norm refers to $a(v, v)$ corresponding to physical energy in many applications.

Goal-oriented error estimates. The classical *a priori* and *a posteriori* error estimates (2.67) and (2.71) relate the V-norm of the error $e = u - u_h$ to the regularity of the exact solution u and the residual $r = L(v) - a(u_h, v)$ of the finite element solution u_h, respectively. However, in applications it is often necessary to control the error in a certain *output functional* $\mathcal{M} : V \to \mathbb{R}$ of the computed solution to within some given tolerance $\epsilon > 0$. Typical functionals are average values of the computed solution, such as the lift or drag of an object immersed in a flow field. In these situations, one would ideally like to choose the finite element space $V_h \subset V$ such that the finite element solution u_h satisfies

$$\eta \equiv |\mathcal{M}(u) - \mathcal{M}(u_h)| \leqslant \epsilon \tag{2.73}$$

with minimal computational work. We assume here that both the output functional and the variational problem are linear, but the analysis may be easily extended to the full nonlinear case [Eriksson

et al.1995, Becker and Rannacher 2001].

To estimate the error in the output functional $\mathcal{M}$, we introduce an auxiliary *dual* problem: find $z \in V^*$ such that

$$a^*(z, v) = \mathcal{M}(v) \quad \forall\, v \in \hat{V}^*. \tag{2.74}$$

We note here that the functional $\mathcal{M}$ enters as data in the dual problem. The dual (adjoint) bilinear form $a^* : V^* \times \hat{V}^* \to \mathbb{R}$ is defined by

$$a^*(v, w) = a(w, v) \quad \forall\, (v, w) \in V^* \times \hat{V}^*. \tag{2.75}$$

The dual trial and test spaces are given by

$$\begin{aligned} V^* &= \hat{V}, \\ \hat{V}^* &= V_0 = \{v - w : v, w \in V\}; \end{aligned} \tag{2.76}$$

that is, the dual trial space is the primal test space and the dual test space is the primal trial space modulo boundary conditions. In particular, if $V = u_0 + \hat{V}$ then $V^* = \hat{V}^* = \hat{V}$, and both the dual test and trial functions vanish at Dirichlet boundaries. The definition of the dual problem leads us to the following representation of the error:

$$\begin{aligned} \mathcal{M}(u) - \mathcal{M}(u_h) &= \mathcal{M}(u - u_h) \\ &= a^*(z, u - u_h) \\ &= a(u - u_h, z) \\ &= L(z) - a(u_h, z) \\ &= r(z). \end{aligned} \tag{2.77}$$

We find that the error is exactly represented by the residual of the dual solution:

$$\mathcal{M}(u) - \mathcal{M}(u_h) = r(z). \tag{2.78}$$

2.6.3 Adaptivity

As seen above, one may estimate the error in a computed finite element solution u_h in the V-norm or an output functional by estimating the size of the residual r. This may be done in several different ways. The estimate typically involves integration by parts to recover the strong element-wise residual of the original PDE, possibly in combination with the solution of local problems over cells or patches of cells. In the case of the standard piecewise linear finite element approximation of Poisson's equation (2.1), one may obtain the following estimate:

$$\|u - u_h\|_V \equiv \|\nabla e\|_{L^2} \leqslant C \left(\sum_{T \in \mathcal{T}_h} h_T^2 \|R\|_T^2 + h_T \|[\partial_n u_h]\|_{\partial T}^2 \right)^{1/2}, \tag{2.79}$$

where $R|_T = f|_T + \Delta u_h|_T$ is the strong residual, h_T denotes the mesh size (the diameter of the smallest circumscribed sphere around each cell T) and $[\partial_n u_h]$ denotes the jump of the normal derivative across mesh facets. For a derivation of this estimate, see for example Elman et al. (2005). Letting $\eta_T^2 = h_T^2 \|R\|_T^2 + h_T \|[\partial_n u_h]\|_{\partial T}^2$, we obtain the estimate

$$\|u - u_h\|_V \leqslant \eta_h \equiv C \left(\sum_T \eta_T^2 \right)^{1/2}. \tag{2.80}$$

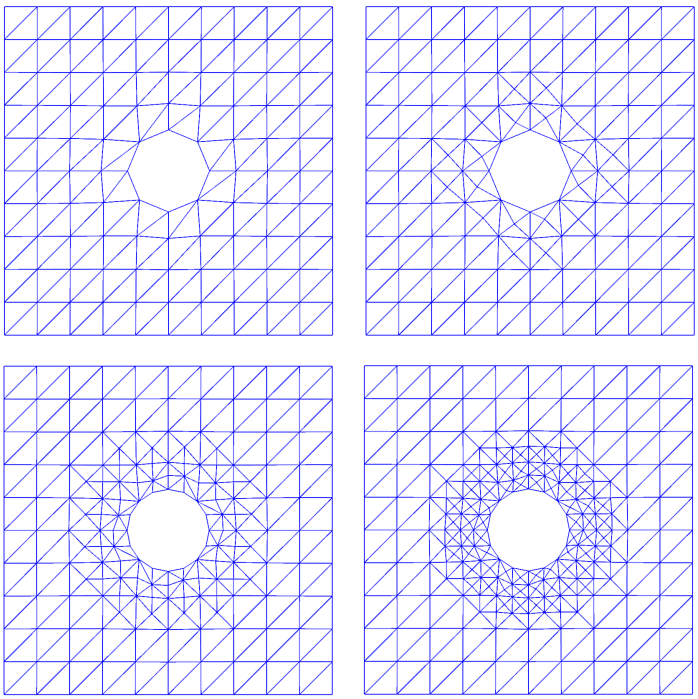


Figure 2.8: A sequence of adaptively refined meshes obtained by successive refinement of an original coarse mesh.

An adaptive algorithm seeks to determine a mesh size $h = h(x)$ such that $\eta_h \leqslant \epsilon$. Starting from an initial coarse mesh, the mesh is successively refined in those cells where the error indicator η_T is large. Several strategies are available, such as refining the top fraction of all cells where η_T is large, say the first 20% of all cells ordered by the size of η_T. Other strategies include refining all cells where η_T is above a certain fraction of $\max_{T \in \mathcal{T}_h} \eta_T$, or refining a top fraction of all cells such that the sum of their error indicators account for a significant fraction of η_h (so-called *Dörfler marking* (Dörfler, 1996)).

Once the mesh has been refined, a new solution and new error indicators can be computed. The process is then repeated until either $\eta_h \leqslant \epsilon$ (the stopping criterion) or the available resources (CPU time and memory) have been exhausted. The adaptive algorithm yields a sequence of successively refined meshes as illustrated in Figure 2.8. For time-dependent problems, an adaptive algorithm needs to decide both on the local mesh size and the size of the (local) time step as functions of space *and* time. Ideally, the error estimate η_h is close to the actual error, as measured by the *efficiency index* η_h / η which should be close to and bounded below by one.

2.7 *Automating the finite element method*

The FEniCS Project seeks to automate the solution of differential equations. This is a formidable task, but it may be approached by an automation of the finite element method. In particular, this automation relies on the following key steps:

(i) automation of discretization,

(ii) automation of discrete solution,

(iii) automation of error control.

Since its inception in 2003, the FEniCS Project has been concerned mainly with the automation of discretization, resulting in the development of the form compilers FFC and SyFi/SFC, the code generation interface UFC, the form language UFL, and a generic assembler implemented as part of DOLFIN. As a result, variational problems for a large class of partial differential equations may now be automatically discretized by the finite element method using FEniCS. For the automation of discrete solution; that is, the solution of linear and nonlinear systems arising from the automated discretization of variational problems, interfaces to state-of-the-art libraries for linear algebra have been implemented as part of DOLFIN. Ongoing work is now seeking to automate error control by automated error estimation and adaptivity. In the following chapters, we return to specific aspects of the automation of the finite element method developed as part of the FEniCS Project. The mathematical methodology behind the FEniCS Project has also been described in a number of scientific works. For further reading, we refer to Logg (2007); Logg and Wells (2010); Kirby (2004); Kirby and Logg (2006); Alnæs et al. (2009); Alnæs and Mardal (2010); Kirby et al. (2005, 2006); Kirby and Logg (2007, 2008); Kirby and Scott (2007); Kirby (2006b); Ølgaard et al. (2008); Rognes et al. (2009); Ølgaard and Wells (2010); Logg (2009).

2.8 Historical notes

In 1915, Boris Grigoryevich Galerkin formulated a general method for solving differential equations (Galerkin, 1915). A similar approach was presented sometime earlier by Bubnov. Galerkin's method, or the Bubnov–Galerkin method, was originally formulated with global polynomials and goes back to the variational principles of Leibniz, Euler, Lagrange, Dirichlet, Hamilton, Castigliano (Castigliano, 1879), Rayleigh (Rayleigh, 1870) and Ritz (Ritz, 1908). Galerkin's method with piecewise polynomial spaces $(V_h, \hat{V}_h)$ is known as the *finite element method*. The finite element method was introduced by engineers for structural analysis in the 1950s and was independently proposed by Courant (Courant, 1943). The exploitation of the finite element method among engineers and mathematicians exploded in the 1960s. Since then, the machinery of the finite element method has been expanded and refined into a comprehensive framework for the design and analysis of numerical methods for differential equations; see Zienkiewicz et al. (2005); Strang and Fix (1973); Ciarlet (1976); Becker et al. (1981); Hughes (1987); Brenner and Scott (2008). Recently, the quest for compatible (stable) discretizations of mixed variational problems has led to the development of finite element exterior calculus (Arnold et al., 2006a).

Work on *a posteriori* error analysis of finite element methods dates back to the pioneering work of Babuška and Rheinboldt (1978). Important references include the works by Bank and Weiser (1985); Zienkiewicz and Zhu (1987); Eriksson and Johnson (1991, 1995a); Eriksson and Johnson, 1995b,c); Eriksson et al. (1998); Ainsworth and Oden (1993) and the reviews papers (Eriksson et al., 1995; Verfürth, 1994, 1999; Ainsworth and Oden, 2000; Becker and Rannacher, 2001).

3 Common and unusual finite elements

By Robert C. Kirby, Anders Logg, Marie E. Rognes and Andy R. Terrel

This chapter provides a glimpse of the considerable range of finite elements in the literature. Many of the elements presented here are implemented as part of the FEniCS Project already; some are future work. The universe of finite elements extends far beyond what we consider here. In particular, we consider only simplicial, polynomial-based elements. We thus bypass elements defined on quadrilaterals and hexahedra, composite and macro-element techniques, as well as XFEM-type methods. Even among polynomial-based elements on simplices, the list of elements can be extended. Nonetheless, this chapter presents a comprehensive collection of some of the most common, and some more unusual, finite elements.

3.1 The finite element definition

The Ciarlet definition of a *finite element* was first introduced in a set of lecture notes by Ciarlet (1975) and became popular after his 1978 book (Ciarlet, 2002). It remains the standard definition today, see for example Brenner and Scott (2008). The definition, which was also presented in Chapter 2, reads as follows:

Definition 3.1 (Finite element (Ciarlet, 2002)) *A finite element is defined by a triple* $(T, \mathcal{V}, \mathcal{L})$*, where*

- *the domain* T *is a bounded, closed subset of* $\mathbb{R}^d$ *(for* $d = 1, 2, 3, \ldots$*) with nonempty interior and piecewise smooth boundary;*
- *the space* $\mathcal{V} = \mathcal{V}(T)$ *is a finite dimensional function space on* T *of dimension* n*;*
- *the set of degrees of freedom (nodes)* $\mathcal{L} = \{\ell_1, \ell_2, \ldots, \ell_n\}$ *is a basis for the dual space* $\mathcal{V}'$*; that is, the space of bounded linear functionals on* $\mathcal{V}$*.*

Similar ideas were introduced earlier in Ciarlet and Raviart (1972)[1], in which unisolvence[2] of a set of interpolation points $\{x^i\}_i$ was discussed. This is closely related to the unisolvence of $\mathcal{L}$ when the degrees of freedom are given by by $\ell_i(v) = v(x^i)$. Conditions for uniquely determining a polynomial based on interpolation of function values and derivatives at a set of points was also discussed in Bramble and Zlámal (1970), although the term unisolvence was not used.

For any finite element, one may define a local basis for $\mathcal{V}$ that is dual to the degrees of freedom. Such a basis $\{\phi_1^T, \phi_2^T \ldots, \phi_n^T\}$ satisfies $\ell_i(\phi_j^T) = \delta_{ij}$ for $1 \leqslant i, j \leqslant n$ and is called the *nodal basis*. It is typically this basis that is used in finite element computations.

[1] The Ciarlet triple was originally written as (K, P, Σ) with K denoting T, P denoting $\mathcal{V}$, and Σ denoting $\mathcal{L}$.

[2] To check whether a given set of linear functionals is a basis for $\mathcal{V}'$, one may check whether it is *unisolvent* for $\mathcal{V}$; that is, for $v \in \mathcal{V}$, $\ell_i(v) = 0$ for $i = 1, \ldots, n$ if and only if $v = 0$.

Also associated with a finite element is a *local interpolation operator*, sometimes called a *nodal interpolant*. Given some function f on T, the nodal interpolant is defined by

$$\Pi_T(f) = \sum_{i=1}^{n} \ell_i(f)\phi_i^T, \tag{3.1}$$

assuming that f is smooth enough for all of the degrees of freedom acting on it to be well-defined.

Once a local finite element space is defined, it is relatively straightforward to define a global finite element space over a tessellation $\mathcal{T}_h$. One defines the global space to consist of functions whose restrictions to each $T \in \mathcal{T}_h$ lie in the local space $\mathcal{V}(T)$ and that also satisfy any required continuity requirements. Typically, the degrees of freedom for each local element are chosen such that if the degrees of freedom on a common interface between two adjacent cells T and T' agree, then a function will satisfy the required continuity condition.

When constructing a global finite element space, it is common to construct a single *reference finite element* $(\hat{T}, \hat{\mathcal{V}}, \hat{\mathcal{L}})$ and map it to each cell in the mesh. As we are dealing with a simplicial geometry, the mapping between $\hat{T}$ and each $T \in \mathcal{T}_h$ will be affine. Originally defined for the purpose of error estimation, but also useful for computation, is the notion of *affine equivalence*. Let $F_T : \hat{T} \to T$ denote this affine map. Let $v \in \mathcal{V}$. The *pullback* associated with the affine map is given by $\mathcal{F}^*(v)(\hat{x}) = v(F_T(\hat{x}))$ for all $\hat{x} \in \hat{T}$. Given a functional $\hat{\ell} \in \hat{\mathcal{V}}'$, its *pushforward* acts on a function in $v \in \mathcal{V}$ by $\mathcal{F}_*(\hat{\ell})(v) = \hat{\ell}(\mathcal{F}^*(v))$.

Definition 3.2 (Affine equivalence) *Let $(\hat{T}, \hat{\mathcal{V}}, \hat{\mathcal{L}})$ and $(T, \mathcal{V}, \mathcal{L})$ be finite elements and $F_T : \hat{T} \to T$ be a non-degenerate affine map. The finite elements are* affine equivalent *if $\mathcal{F}^*(\mathcal{V}) = \hat{\mathcal{V}}$ and $\mathcal{F}_*(\hat{\mathcal{L}}) = \mathcal{L}$.*

One consequence of affine equivalence is that only a single nodal basis needs to be constructed, and then it can be mapped to each cell in a mesh. Moreover, this idea of equivalence can be extended to some vector-valued elements when certain kinds of Piola mappings are used. In this case, the affine map is the same, but the pull-back and push-forward are appropriately modified. It is also worth stating that not all finite elements generate affine equivalent or Piola-equivalent families. The Lagrange elements are affine equivalent in H^1, but the Hermite and Argyris elements are not. The Raviart–Thomas elements are Piola-equivalent in $H(\mathrm{div})$, while the Mardal–Tai–Winther elements are not.

A dictionary of the finite elements discussed in this chapter is presented in Table 3.1.

3.2 Notation

- The space of polynomials of degree up to and including q on a domain $T \subset \mathbb{R}^d$ is denoted by $\mathcal{P}_q(T)$ and the corresponding d-vector fields by $[\mathcal{P}_q(T)]^d$.
- A finite element space E is called V-conforming if $E \subseteq V$. If not, it is called (V-) nonconforming.
- The elements of $\mathcal{L}$ are usually referred to as the *degrees of freedom* of the element $(T, \mathcal{V}, \mathcal{L})$. When describing finite element families, it is usual to illustrate the degrees of freedom with a certain schematic notation. We summarize the notation used here in the list below and in Figure 3.1.

  Point evaluation. A black sphere (disc) at a point x denotes point evaluation of the function v at that point:

  $$\ell(v) = v(x). \tag{3.2}$$

  For a vector valued function v with d components, a black sphere denotes evaluation of all components and thus corresponds to d degrees of freedom.

Finite element	Short name	Sobolev space	Conforming
(Quintic) Argyris	ARG	H^2	Yes
Arnold–Winther	AW	$H(\mathrm{div}; \mathbb{S})$	Yes
Brezzi–Douglas–Marini	BDM	$H(\mathrm{div})$	Yes
Crouzeix–Raviart	CR	H^1	No
Discontinuous Lagrange	DG	L^2	Yes
(Cubic) Hermite	HER	H^2	No
Lagrange	CG	H^1	Yes
Mardal–Tai–Winther	MTW	$H^1 / H(\mathrm{div})$	No/Yes
(Quadratic) Morley	MOR	H^2	No
Nédélec first kind	NED^1	$H(\mathrm{curl})$	Yes
Nédélec second kind	NED^2	$H(\mathrm{curl})$	Yes
Raviart–Thomas	RT	$H(\mathrm{div})$	Yes

Table 3.1: A dictionary of the finite elements discussed in this chapter, including full name and the respective (highest order) Sobolev space to which the elements are conforming/nonconforming.

Evaluation of all first derivatives. A dark gray, slightly larger sphere (disc) at a point x denotes point evaluation of all first derivatives of the function v at that point:

$$\ell_i(v) = \frac{\partial v(x)}{\partial x_i}, \quad i = 1, \ldots, d, \tag{3.3}$$

thus corresponding to d degrees of freedom.

Evaluation of all second derivatives. A light gray, even larger sphere (disc) at a point x denotes point evaluation of all second derivatives of the function v at that point:

$$\ell_{ij}(v) = \frac{\partial^2 v(x)}{\partial x_i \partial x_j}, \quad 1 \leqslant i \leqslant j \leqslant d, \tag{3.4}$$

thus corresponding to $d(d+1)/2$ degrees of freedom.

Evaluation of directional component. An arrow at a point x in a direction n denotes evaluation of the vector-valued function v in the direction n at the point x:

$$\ell(v) = v(x) \cdot n. \tag{3.5}$$

The direction n is typically the normal direction of a facet, or a tangent direction of a facet or edge. We will sometimes use an arrow at a point to denote a moment (integration against a weight function) of a component of the function over a facet or edge.

Evaluation of directional derivative. A black line at a point x in a direction n denotes evaluation of the directional derivative of the scalar function v in the direction n at the point x:

$$\ell(v) = \nabla v(x) \cdot n. \tag{3.6}$$

Evaluation of interior moments. A set of concentric spheres (discs) denotes interior moment degrees of freedom; that is, degrees of freedom defined by integration against a weight function over the interior of the domain T. The spheres are colored white-black-white etc.

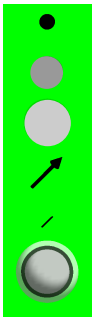


Figure 3.1: Summary of notation used for degrees of freedom. In this example, the three concentric spheres indicate a set of three degrees of freedom defined by interior moments.

We note that, for some of the finite elements presented below, the literature will use different notation and numbering schemes, so that our presentation may be quite different from the original presentation of the elements. In particular, the families of Raviart–Thomas and Nédélec spaces of the first kind are traditionally numbered from 0, while we have followed the more recent scheme from the finite element exterior calculus of numbering from 1.

3.3 H^1 finite elements

The space H^1 is fundamental in the analysis and discretization of weak forms for second-order elliptic problems, and finite element subspaces of H^1 give rise to some of the best-known finite elements. Typically, these elements use C^0 approximating spaces, since a piecewise smooth function on a bounded domain is H^1 if and only if it is continuous (Braess, 2007, Theorem 5.2). We consider the classic Lagrange element, as well as a nonconforming example, the Crouzeix–Raviart space. It is worth noting that the Hermite element considered later is technically only an H^1 element, but can be used as a nonconforming element for smoother spaces. Also, smoother elements such as Argyris may be used to discretize H^1, although this is less common in practice.

3.3.1 The Lagrange element

The best-known and most widely used finite element is the $\mathcal{P}_1$ Lagrange element. This lowest-degree triangle is sometimes called the *Courant* triangle, after the seminal paper by Courant (1943) in which variational techniques are used with the $\mathcal{P}_1$ triangle to derive a finite difference method. Sometimes this is viewed as "the" finite element method, but in fact there is a whole family of elements parametrized by polynomial degree that generalize the univariate Lagrange interpolating polynomials to simplices, boxes, and other shapes. The Lagrange elements of higher degree offer higher order approximation properties. Moreover, these can alleviate locking phenomena observed when using linear elements or give improved discrete stability properties; see Taylor and Hood (1973); Scott and Vogelius (1985).

Definition 3.3 (Lagrange element) *The Lagrange element* (CG_q) *is defined for* $q = 1, 2, \ldots$ *by*

$$T \in \{\text{interval}, \text{triangle}, \text{tetrahedron}\}, \tag{3.7}$$

$$\mathcal{V} = \mathcal{P}_q(T), \tag{3.8}$$

$$\ell_i(v) = v(x^i), \quad i = 1, \ldots, n(q), \tag{3.9}$$

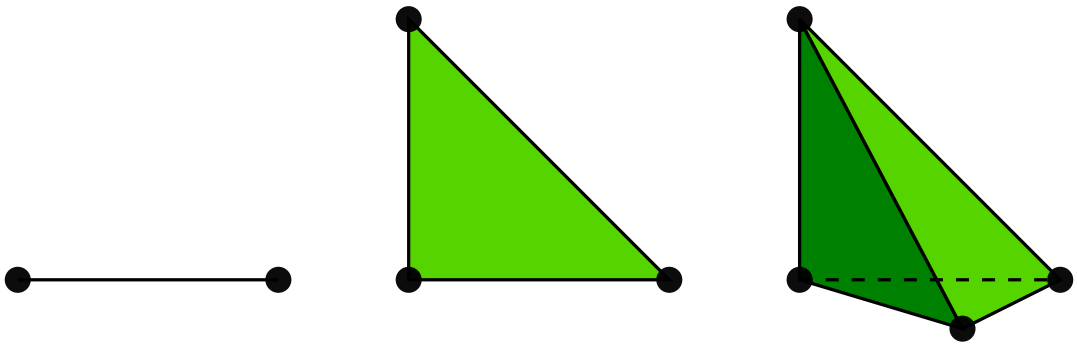


Figure 3.2: The linear Lagrange interval, triangle and tetrahedron.

where $\{x^i\}_{i=1}^{n(q)}$ *is an enumeration of points in* T *defined by*

$$x = \begin{cases} i/q, & 0 \leqslant i \leqslant q, & T \text{ interval}, \\ (i/q, j/q), & 0 \leqslant i+j \leqslant q, & T \text{ triangle}, \\ (i/q, j/q, k/q), & 0 \leqslant i+j+k \leqslant q, & T \text{ tetrahedron}. \end{cases} \tag{3.10}$$

The dimension of the Lagrange finite element thus corresponds to the dimension of the complete polynomials of degree q on T and is

$$n(q) = \begin{cases} q+1, & T \text{ interval}, \\ \frac{1}{2}(q+1)(q+2), & T \text{ triangle}, \\ \frac{1}{6}(q+1)(q+2)(q+3), & T \text{ tetrahedron}. \end{cases} \tag{3.11}$$

The definition above presents one choice for the set of points $\{x^i\}$. However, this is not the only possible choice. In general, it suffices that the set of points $\{x^i\}$ is unisolvent and that the boundary points are located so as to allow C^0 assembly. The point set must include the vertices, $q-1$ points on each edge, $\frac{(q-1)(q-2)}{2}$ points per face, and so forth. The boundary points should be placed symmetrically so that the points on adjacent cells match. While numerical conditioning and interpolation properties can be dramatically improved by choosing these points in a clever way (Warburton, 2005), for the purposes of this chapter the points may be assumed to lie on an equispaced lattice; see Figures 3.2, 3.3 and 3.4.

Letting Π_T^q denote the interpolant defined by the above degrees of freedom of the Lagrange element of degree q, we have from Brenner and Scott (2008) that

$$||u - \Pi_T^q u||_{H^1(T)} \leqslant C\, h_T^q |u|_{H^{q+1}(T)}, \quad ||u - \Pi_T^q u||_{L^2(T)} \leqslant C\, h_T^{q+1} |u|_{H^{q+1}(T)}. \tag{3.12}$$

where, here and throughout, C denotes a generic positive constant not depending on h_T but depending on the degree q and the aspect ratio of the simplex, and u is a sufficiently regular function (or vector-field).

Vector-valued or tensor-valued Lagrange elements are usually constructed by using a Lagrange element for each component.

3.3.2 *The Crouzeix–Raviart element*

The Crouzeix–Raviart element was introduced in Crouzeix and Raviart (1973) as a technique for solving the stationary Stokes equations. The global element space consists of piecewise linear

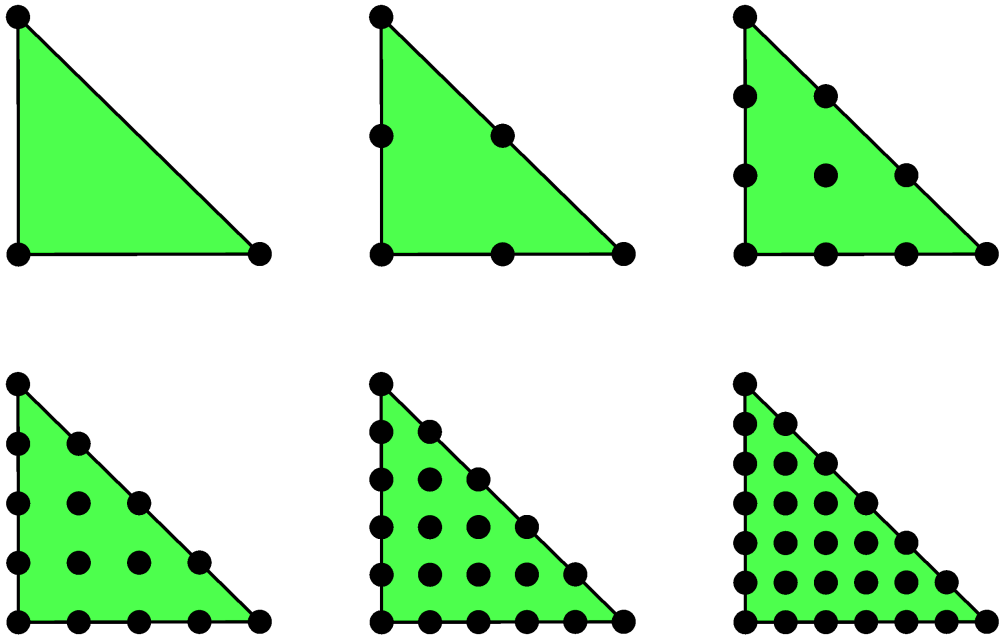


Figure 3.3: The Lagrange CG_q triangle for $q = 1, 2, 3, 4, 5, 6$.

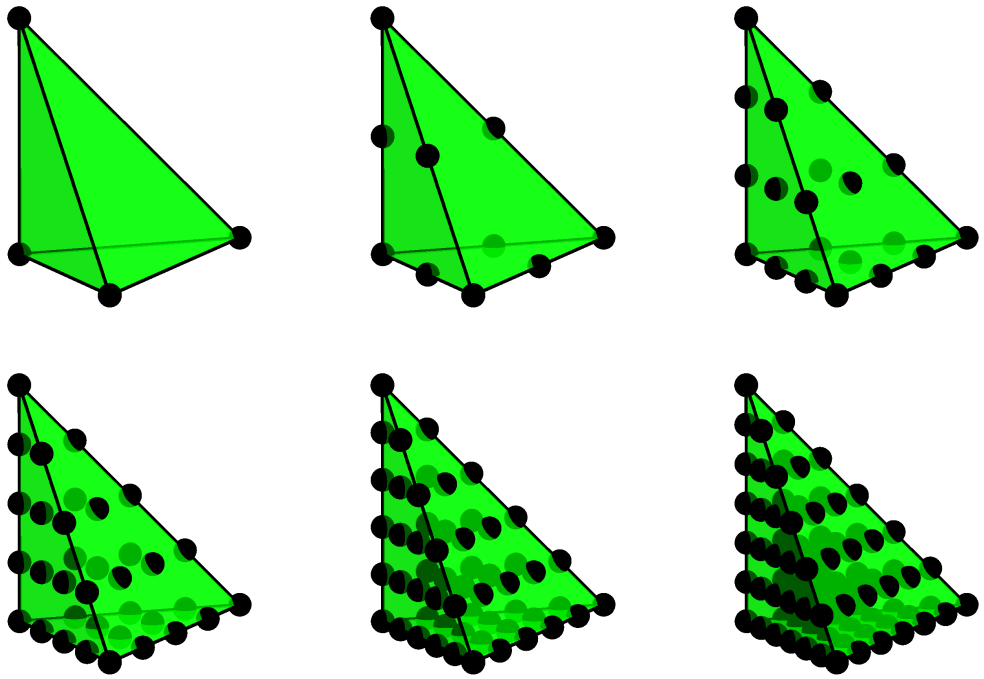


Figure 3.4: The Lagrange CG_q tetrahedron for $q = 1, 2, 3, 4, 5, 6$.

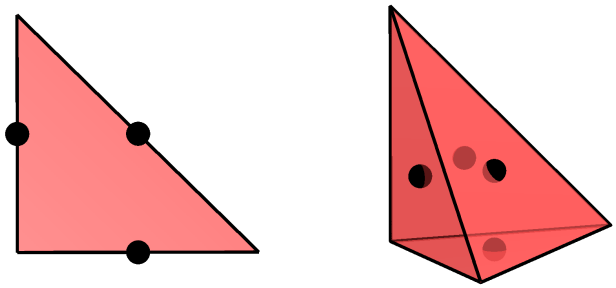


Figure 3.5: Illustration of the Crouzeix–Raviart elements on triangles and tetrahedra. The degrees of freedom are point evaluation at the midpoint of each facet.

polynomials, as for the linear Lagrange element. However, in contrast to the Lagrange element, the global basis functions are not required to be continuous at all points; continuity is only imposed at the midpoint of facets. The element is hence not H^1-conforming, but it is typically used for nonconforming approximations of H^1 functions (and vector fields). Other applications of the Crouzeix–Raviart element includes linear elasticity (Hansbo and Larson, 2003) and Reissner–Mindlin plates (Arnold and Falk, 1989).

Definition 3.4 (Crouzeix–Raviart element) *The (linear) Crouzeix–Raviart element* (CR) *is defined by*

$$T \in \{\text{triangle}, \text{tetrahedron}\}, \tag{3.13}$$

$$\mathcal{V} = \mathcal{P}_1(T), \tag{3.14}$$

$$\ell_i(v) = v(x^i), \quad i = 1, \ldots, n. \tag{3.15}$$

where $\{x^i\}$ *are the barycenters (midpoints) of each facet of the domain* T*.*

The dimension of the Crouzeix–Raviart element on $T \subset \mathbb{R}^d$ is thus

$$n = d + 1 \tag{3.16}$$

for $d = 2, 3$.

Letting Π_T denote the interpolation operator defined by the degrees of freedom, the Crouzeix–Raviart element interpolates as the linear Lagrange element (Braess, 2007, Chapter 3.I):

$$||u - \Pi_T u||_{H^1(T)} \leqslant C\, h_T |u|_{H^2(T)}, \quad ||u - \Pi_T u||_{L^2(T)} \leqslant C\, h_T^2 |u|_{H^2(T)}. \tag{3.17}$$

Vector-valued Crouzeix–Raviart elements can be defined by using a Crouzeix–Raviart element for each component, or by using facet normal and facet tangential components at the midpoints of each facet as degrees of freedom. The Crouzeix–Raviart element can be extended to higher odd degrees ($q = 3, 5, 7 \ldots$) (Crouzeix and Falk, 1989).

3.4 $H(\text{div})$ *finite elements*

The Sobolev space $H(\text{div})$ consists of vector fields for which the components and the weak divergence are square-integrable. This is a weaker requirement than for a d-vector field to be in $[H^1]^d$ (for $d \geqslant 2$). This space naturally occurs in connection with mixed formulations of second-order elliptic problems, porous media flow, and elasticity equations. For a finite element family to be $H(\text{div})$-conforming, each component need not be continuous, but the normal component must be continuous. In order

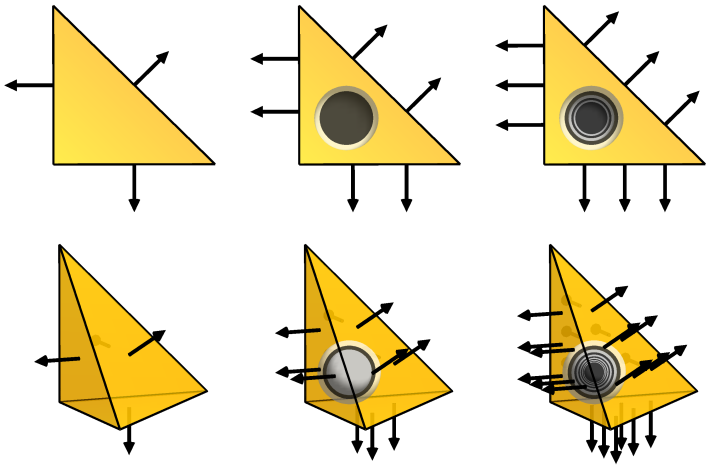


Figure 3.6: Illustration of the degrees of freedom for the first, second and third degree Raviart–Thomas elements on triangles and tetrahedra. The degrees of freedom are moments of the normal component against $\mathcal{P}_{q-1}$ on facets (edges and faces, respectively) and, for the higher degree elements, interior moments against $[\mathcal{P}_{q-2}]^d$. Alternatively, as indicated in this illustration, the moments of normal components may be replaced by point evaluation of normal components.

to ensure such continuity, the degrees of freedom of $H(\mathrm{div})$-conforming elements usually include normal components on element facets.

The two main families of $H(\mathrm{div})$-conforming elements are the Raviart–Thomas and Brezzi–Douglas–Marini elements. These two families are described below. In addition, the Arnold–Winther element discretizing the space of symmetric tensor fields with square-integrable row-wise divergence and the Mardal–Tai–Winther element are included.

3.4.1 *The Raviart–Thomas element*

The Raviart–Thomas element was introduced by Raviart and Thomas (1977). It was the first element to discretize the mixed form of second-order elliptic equations on triangles. Its element space $\mathcal{V}$ is designed so that it is the smallest polynomial space $\mathcal{V} \subset \mathcal{P}_q(T)$, for $q = 1, 2, \ldots$, from which the divergence maps onto $\mathcal{P}_{q-1}(T)$. Shortly thereafter, it was extended to tetrahedra and boxes by Nédélec (1980). It is therefore sometimes referred to as the Raviart–Thomas–Nédélec element. Here, we label both the two- and three-dimensional versions as the Raviart–Thomas element.

The definition given below is based on the one presented by Nédélec (1980) (and Brezzi and Fortin (1991)). The original Raviart–Thomas paper used a slightly different form. Moreover, Raviart and Thomas originally started counting at $q = 0$. Hence, the lowest degree element is traditionally called the RT_0 element. For the sake of consistency, such that a finite element of polynomial degree q is included in $\mathcal{P}_q(T)$, we here label the lowest degree elements by $q = 1$ instead (as did also Nédélec).

Definition 3.5 (Raviart–Thomas element) *The Raviart–Thomas element* (RT_q) *is defined for* $q = 1, 2, \ldots$ *by*

$$T \in \{\text{triangle}, \text{tetrahedron}\}, \tag{3.18}$$

$$\mathcal{V} = [\mathcal{P}_{q-1}(T)]^d + x\mathcal{P}_{q-1}(T), \tag{3.19}$$

$$\mathcal{L} = \begin{cases} \int_f v \cdot n\, p \,\mathrm{d}s, & \text{for a set of basis functions } p \in \mathcal{P}_{q-1}(f) \text{ for each facet } f, \\ \int_T v \cdot p \,\mathrm{d}x, & \text{for a set of basis functions } p \in [\mathcal{P}_{q-2}(T)]^d \text{ for } q \geqslant 2. \end{cases} \tag{3.20}$$

As an example, the lowest degree Raviart–Thomas space on triangles is a three-dimensional space and consists of vector fields of the form

$$v(x) = \alpha + \beta x, \tag{3.21}$$

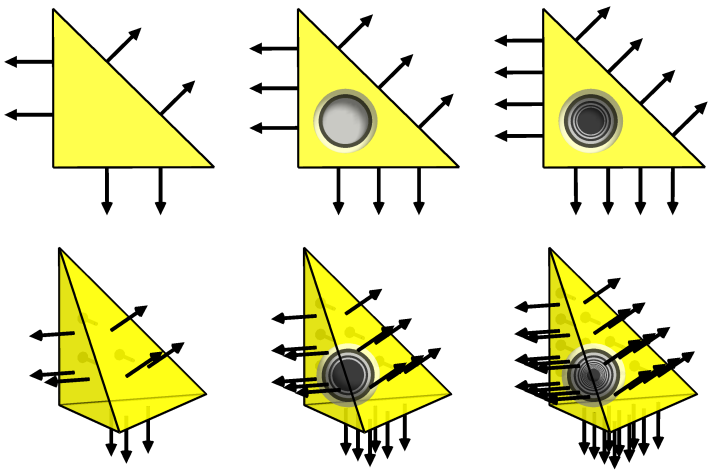


Figure 3.7: Illustration of the first, second and third degree Brezzi–Douglas–Marini elements on triangles and tetrahedra. The degrees of freedom are moments of the normal component against $\mathcal{P}_q$ on facets (edges and faces, respectively) and, for the higher degree elements, interior moments against NED^1_{q-1}. Alternatively, as indicated in this illustration, the moments of normal components may be replaced by point evaluation of normal components.

where α is a vector-valued constant, and β is a scalar constant.

The dimension of RT_q is

$$n(q) = \begin{cases} q(q+2), & T \text{ triangle}, \\ \frac{1}{2}q(q+1)(q+3), & T \text{ tetrahedron}. \end{cases} \tag{3.22}$$

Letting Π^q_T denote the interpolation operator defined by the degrees of freedom above for $q = 1, 2, \ldots$, we have that (Brezzi and Fortin, 1991, Chapter III.3)

$$||u - \Pi^q_T u||_{H(\mathrm{div})(T)} \leqslant C\, h^q_T |u|_{H^{q+1}(T)}, \quad ||u - \Pi^q_T u||_{L^2(T)} \leqslant C\, h^q_T |u|_{H^q(T)}. \tag{3.23}$$

3.4.2 The Brezzi–Douglas–Marini element

The Brezzi–Douglas–Marini element was introduced by Brezzi, Douglas and Marini in two dimensions (for triangles) in Brezzi et al. (1985a). The element can be viewed as an alternative to the Raviart–Thomas element using a complete polynomial space. It was later extended to three dimensions (tetrahedra, prisms and cubes) in Nédélec (1986) and Brezzi et al. (1987a). The definition given here is based on that of Nédélec (1986).

The Brezzi–Douglas–Marini element was introduced for mixed formulations of second-order elliptic equations. However, it is also useful for weakly symmetric discretizations of the elastic stress tensor; see Farhloul and Fortin (1997); Arnold et al. (2007).

Definition 3.6 (Brezzi–Douglas–Marini element) *The Brezzi–Douglas–Marini element* (BDM_q) *is defined for* $q = 1, 2, \ldots$ *by*

$$T \in \{\text{triangle}, \text{tetrahedron}\}, \tag{3.24}$$

$$\mathcal{V} = [\mathcal{P}_q(T)]^d, \tag{3.25}$$

$$\mathcal{L} = \begin{cases} \int_f v \cdot n p \,\mathrm{d}s, & \text{for a set of basis functions } p \in \mathcal{P}_q(f) \text{ for each facet } f, \\ \int_T v \cdot p \,\mathrm{d}x, & \text{for a set of basis functions } p \in \mathrm{NED}^1_{q-1}(T) \text{ for } q \geqslant 2. \end{cases} \tag{3.26}$$

where NED^1 *refers to the Nédélec* $H(\mathrm{curl})$ *elements of the first kind, defined below in Section 3.5.1.*

The dimension of BDM_q is

$$n(q) = \begin{cases} (q+1)(q+2), & T \text{ triangle}, \\ \frac{1}{2}(q+1)(q+2)(q+3), & T \text{ tetrahedron}. \end{cases} \tag{3.27}$$

Letting Π_T^q denote the interpolation operator defined by the degrees of freedom for $q = 1, 2, \ldots$, we have that (Brezzi and Fortin, 1991, Chapter III.3)

$$||u - \Pi_T^q u||_{H(\mathrm{div})(T)} \leqslant C\, h_T^q |u|_{H^{q+1}(T)}, \quad ||u - \Pi_T^q u||_{L^2(T)} \leqslant C\, h_T^{q+1} |u|_{H^{q+1}(T)}. \tag{3.28}$$

A slight modification of the Brezzi–Douglas–Marini element constrains the element space $\mathcal{V}$ by only allowing normal components on the boundary of polynomial degree $q-1$ (rather than the full polynomial degree q). Such an element was suggested on rectangles by Brezzi et al. (1987b), and the triangular analogue was given in Brezzi and Fortin (1991). In similar spirit, elements with differing degrees on the boundary suitable for varying the polynomial degree between triangles were derived in Brezzi et al. (1985b).

3.4.3 The Mardal-Tai-Winther element

The Mardal–Tai–Winther element was introduced in Mardal et al. (2002) as a finite element suitable for the velocity space for both Darcy and Stokes flow in two dimensions. In the Darcy flow equations, the velocity space only requires $H(\mathrm{div})$-regularity. Moreover, discretizations based on H^1-conforming finite elements are typically not stable. On the other hand, for the Stokes equations, the velocity space does stipulate H^1-regularity. The Mardal–Tai–Winther element is $H(\mathrm{div})$-conforming, but H^1-nonconforming. The element was extended to three dimensions in Tai and Winther (2006), but we only present the two-dimensional case here.

Definition 3.7 (Mardal–Tai–Winther element) *The Mardal–Tai–Winther element (*MTW*) is defined by*

$$T = \text{triangle}, \tag{3.29}$$

$$\mathcal{V} = \{v \in [\mathcal{P}_3(T)]^2, \text{ such that } \operatorname{div} v \in \mathcal{P}_0(T) \text{ and } v \cdot n|_f \in \mathcal{P}_1(T) \text{ for each facet } f\}, \tag{3.30}$$

$$\mathcal{L} = \begin{cases} \int_f v \cdot n\, p \,\mathrm{d}s, & \text{for a set of basis functions } p \in \mathcal{P}_1(f) \text{ for each facet } f, \\ \int_f v \cdot t \,\mathrm{d}s, & \text{for each facet } f. \end{cases} \tag{3.31}$$

The dimension of MTW is

$$n = 9. \tag{3.32}$$

Letting Π_T denote the interpolation operator defined by the degrees of freedom, we have that

$$||u - \Pi_T u||_{H^1(T)} \leqslant C\, h_T |u|_{H^2(T)}, \; ||u - \Pi_T u||_{H(\mathrm{div})(T)} \leqslant C\, h_T |u|_{H^2(T)}, \; ||u - \Pi_T u||_{L^2(T)} \leqslant C\, h_T^2 |u|_{H^2(T)}. \tag{3.33}$$

3.4.4 The Arnold–Winther element

The Arnold–Winther element was introduced by Arnold and Winther (2002). This paper presented the first stable mixed (non-composite) finite element for the stress–displacement formulation of linear elasticity. The finite element used for the stress space is what is presented as the Arnold–Winther element here. This finite element is a symmetric tensor element that is row-wise $H(\mathrm{div})$-conforming. The finite element was introduced for a hierarchy of polynomial degrees and extended

Figure 3.8: Illustration of the Mardal–Tai–Winther element. The degrees of freedom are two moments of the normal component on each facet and one moment of the tangential component on each facet. In this figure, the moments of normal components are illustrated by point evaluation of normal components.

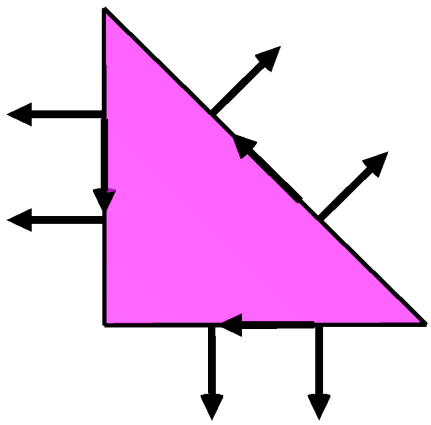
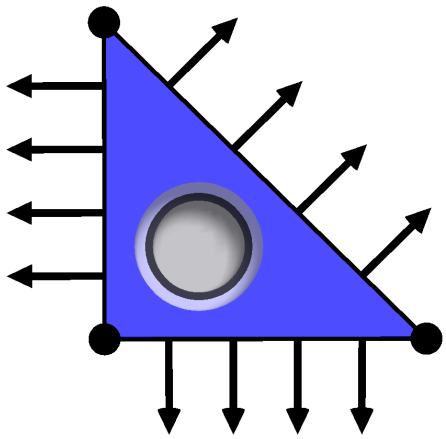


Figure 3.9: Illustration of the Arnold–Winther element. The 24 degrees of freedom are point evaluation at the vertices, the two first moments of the normal component of each row of the tensor field on each facet, and three interior moments.



to three-dimensions in Adams and Cockburn (2005) and Arnold et al. (2008), but we only present the lowest degree two-dimensional case here.

Definition 3.8 (Arnold–Winther element) *The (lowest degree) Arnold–Winther element* (AW) *is defined by*

$$T = \text{triangle}, \tag{3.34}$$

$$\mathcal{V} = \{v \in \mathcal{P}_3(T;\mathbb{S}) : \operatorname{div} v \in \mathcal{P}_1(T;\mathbb{R}^2)\}, \tag{3.35}$$

$$\mathcal{L} = \begin{cases} v(x^k)_{ij}, & \text{for } 1 \leqslant i \leqslant j \leqslant 2 \text{ at each vertex } x^k \\ \int_f \sum_{j=1}^2 v_{ij} n_j \, p \, \mathrm{d}s, & \text{for a set of basis functions } p \in \mathcal{P}_1(f), \text{ on each facet } f,\ 1 \leqslant i \leqslant 2, \\ \int_T v_{ij} \, \mathrm{d}x, & \text{for } 1 \leqslant i \leqslant j \leqslant 2. \end{cases} \tag{3.36}$$

The dimension of AW is

$$n = 24. \tag{3.37}$$

Letting Π_T denote the interpolation operator defined by the degrees of freedom, we have that

$$||u - \Pi_T u||_{H(\operatorname{div})(T)} \leqslant C\, h_T^2 |u|_{H^3(T)}, \quad ||u - \Pi_T u||_{L^2(T)} \leqslant C\, h_T^3 |u|_{H^3(T)}. \tag{3.38}$$

3.5 $H(\mathrm{curl})$ *finite elements*

The Sobolev space $H(\mathrm{curl})$ arises frequently in problems associated with electromagnetism. The Nédélec elements, also colloquially referred to as *edge elements*, are much used for such problems, and stand as a premier example of the power of "nonstandard" (meaning not lowest-degree Lagrange) finite elements (Nédélec, 1980, 1986). For a piecewise polynomial to be $H(\mathrm{curl})$-conforming, the tangential component must be continuous. Therefore, the degrees of freedom for $H(\mathrm{curl})$-conforming finite elements typically include tangential components.

There are four families of finite element spaces due to Nédélec, introduced in the papers Nédélec (1980) and Nédélec (1986). The first (1980) paper introduced two families of finite element spaces on tetrahedra, cubes and prisms: one $H(\mathrm{div})$-conforming family and one $H(\mathrm{curl})$-conforming family. These families are known as Nédélec $H(\mathrm{div})$ elements of the *first kind* and Nédélec $H(\mathrm{curl})$ elements of the *first kind*, respectively. The $H(\mathrm{div})$ elements can be viewed as the three-dimensional extension of the Raviart–Thomas elements. (These are therefore presented as Raviart–Thomas elements above.) The first kind Nédélec $H(\mathrm{curl})$ elements are presented below.

The second (1986) paper introduced two more families of finite element spaces: again, one $H(\mathrm{div})$-conforming family and one $H(\mathrm{curl})$-conforming family. These families are known as Nédélec $H(\mathrm{div})$ elements of the *second kind* and Nédélec $H(\mathrm{curl})$ elements of the *second kind*, respectively. The $H(\mathrm{div})$ elements can be viewed as the three-dimensional extension of the Brezzi–Douglas–Marini elements. (These are therefore presented as Brezzi–Douglas–Marini elements above.) The second kind Nédélec $H(\mathrm{curl})$ elements are presented below.

In his two classic papers, Nédélec considered only the three-dimensional case. However, one can also define a two-dimensional curl, and two-dimensional $H(\mathrm{curl})$-conforming finite element spaces. We present such elements as Nédélec elements on triangles here. Although attributing these elements to Nédélec may be dubious, we include them for the sake of completeness.

In many ways, Nédélec's work anticipates the recently introduced finite element exterior calculus presented in Arnold et al. (2006a), where the first kind spaces appear as $\mathcal{P}_q^-\Lambda^k$ spaces and the second kind as $\mathcal{P}_q\Lambda^k$. Moreover, the use of a differential operator (the elastic strain) in Nédélec (1980) to characterize the function space foreshadows the use of differential complexes in Arnold et al. (2006b).

3.5.1 *The Nédélec $H(\mathrm{curl})$ element of the first kind*

Definition 3.9 (Nédélec $H(\mathrm{curl})$ element of the first kind) *For $q = 1, 2, \ldots$, define the space*

$$S_q(T) = \{s \in [\mathcal{P}_q(T)]^d \,:\, s(x) \cdot x = 0 \quad \forall\, x \in T\}. \tag{3.39}$$

The Nédélec element of the first kind (NED_q^1) is defined for $q = 1, 2, \ldots$ in two dimensions by

$$T = \text{triangle}, \tag{3.40}$$

$$\mathcal{V} = [\mathcal{P}_{q-1}(T)]^2 + S_q(T), \tag{3.41}$$

$$\mathcal{L} = \begin{cases} \int_e v \cdot t\, p \,\mathrm{d}s, & \text{for a set of basis functions } p \in \mathcal{P}_{q-1}(e) \text{ for each edge } e, \\ \int_T v \cdot p \,\mathrm{d}x, & \text{for a set of basis functions } p \in [\mathcal{P}_{q-2}(T)]^2, \text{ for } q \geqslant 2, \end{cases} \tag{3.42}$$

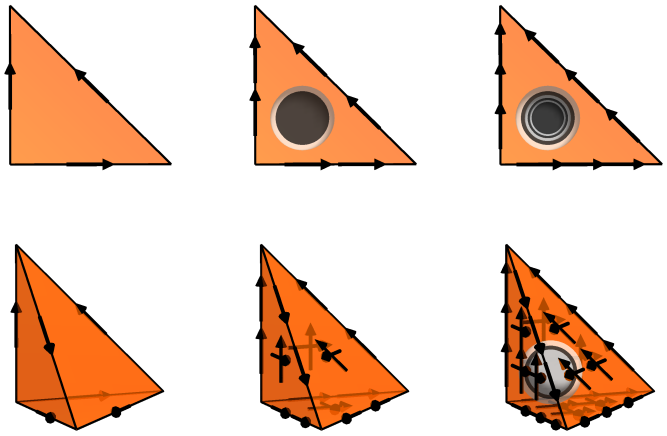


Figure 3.10: Illustration of first, second and third degree Nédélec $H(\mathrm{curl})$ elements of the first kind on triangles and tetrahedra. Note that these elements may be viewed as *rotated* Raviart–Thomas elements. For the first degree Nédélec elements, the degrees of freedom are the average value over edges or, alternatively, the value of the tangential component at the midpoint of edges. Hence the term "edge elements".

where t is the edge tangent; and in three dimensions by

$$T = \text{tetrahedron}, \tag{3.43}$$

$$\mathcal{V} = [\mathcal{P}_{q-1}(T)]^3 + S_q(T), \tag{3.44}$$

$$\mathcal{L} = \begin{cases} \int_e v \cdot t\, p \,\mathrm{d}l, & \text{for a set of basis functions } p \in \mathcal{P}_{q-1}(e) \text{ for each edge } e \\ \int_f v \times n \cdot p \,\mathrm{d}s, & \text{for a set of basis functions } p \in [\mathcal{P}_{q-2}(f)]^2 \text{ for each face } f, \text{ for } q \geqslant 2, \\ \int_T v \cdot p \,\mathrm{d}x, & \text{for a set of basis functions } p \in [\mathcal{P}_{q-3}]^3, \text{ for } q \geqslant 3. \end{cases} \tag{3.45}$$

The dimension of NED_q^1 is

$$n(q) = \begin{cases} q(q+2), & T \text{ triangle}, \\ \frac{1}{2}q(q+2)(q+3), & T \text{ tetrahedron}. \end{cases} \tag{3.46}$$

Letting Π_T^q denote the interpolation operator defined by the degrees of freedom above, we have that (Nédélec, 1980, Theorem 2)

$$||u - \Pi_T^q u||_{H(\mathrm{curl})(T)} \leqslant C\, h_T^q |u|_{H^{q+1}(T)}, \quad ||u - \Pi_T^q u||_{L^2(T)} \leqslant C\, h_T^q |u|_{H^q(T)}. \tag{3.47}$$

3.5.2 *The $H(\mathrm{curl})$ Nédélec element of the second kind*

Definition 3.10 (Nédélec $H(\mathrm{curl})$ element of the second kind) *The Nédélec element of the second kind (NED_q^2) is defined for $q = 1, 2, \ldots$ in two dimensions by*

$$T = \text{triangle}, \tag{3.48}$$

$$\mathcal{V} = [\mathcal{P}_q(T)]^2, \tag{3.49}$$

$$\mathcal{L} = \begin{cases} \int_e v \cdot t\, p \,\mathrm{d}s, & \text{for a set of basis functions } p \in \mathcal{P}_q(e) \text{ for each edge } e, \\ \int_T v \cdot p \,\mathrm{d}x, & \text{for a set of basis functions } p \in \mathrm{RT}_{q-1}(T), \text{ for } q \geqslant 2. \end{cases} \tag{3.50}$$

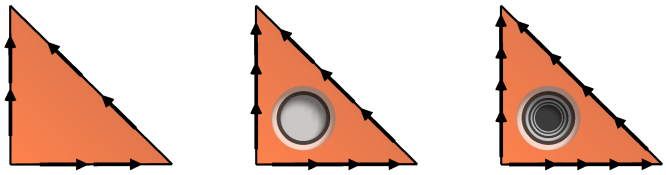


Figure 3.11: Illustration of first, second and third degree Nédélec $H(\mathrm{curl})$ elements of the second kind on triangles. Note that these elements may be viewed as *rotated* Brezzi–Douglas–Marini elements.

Figure 3.12: Illustration of the first degree Nédélec $H(\mathrm{curl})$ elements of the second kind on tetrahedra.

where t is the edge tangent, and in three dimensions by

$$T = \text{tetrahedron}, \tag{3.51}$$

$$\mathcal{V} = [\mathcal{P}_q(T)]^3, \tag{3.52}$$

$$\mathcal{L} = \begin{cases} \int_e v \cdot t\, p \,\mathrm{d}l, & \text{for a set of basis functions } p \in \mathcal{P}_q(e) \text{ for each edge } e, \\ \int_f v \cdot p \,\mathrm{d}s, & \text{for a set of basis functions } p \in \mathrm{RT}_{q-1}(f) \text{ for each face } f, \text{ for } q \geqslant 2 \\ \int_T v \cdot p \,\mathrm{d}x, & \text{for a set of basis functions } p \in \mathrm{RT}_{q-2}(T), \text{ for } q \geqslant 3. \end{cases} \tag{3.53}$$

The dimension of $\mathrm{NED}^2{}_q$ is

$$n(q) = \begin{cases} (q+1)(q+2), & T \text{ triangle}, \\ \frac{1}{2}(q+1)(q+2)(q+3), & T \text{ tetrahedron}. \end{cases} \tag{3.54}$$

Letting Π_T^q denote the interpolation operator defined by the degrees of freedom above, we have that (Nédélec, 1986, Proposition 3)

$$||u - \Pi_T^q u||_{H(\mathrm{curl})(T)} \leqslant C\, h_T^q |u|_{H^{q+1}(T)}, \quad ||u - \Pi_T^q u||_{L^2(T)} \leqslant C\, h_T^{q+1} |u|_{H^{q+1}(T)}. \tag{3.55}$$

3.6 L^2 finite elements

By L^2 elements, one usually refers to finite element spaces where the elements are not in C^0. Such elements naturally occur in mixed formulations of the Poisson equation, Stokes flow, and elasticity. Alternatively, such elements can be used for nonconforming methods imposing the desired continuity

weakly instead of directly. The discontinuous Galerkin (DG) methods provide a typical example. In this case, the numerical flux of element facets is assembled as part of the weak form; numerous variants of DG methods have been defined with different numerical fluxes. DG methods were originally developed for hyperbolic problems but have been successfully applied to many elliptic and parabolic problems. Moreover, the decoupling of each individual element provides an increased opportunity for parallelism and *hp*-adaptivity.

3.6.1 Discontinuous Lagrange

Definition 3.11 (Discontinuous Lagrange element) *The discontinuous Lagrange element* (DG_q) *is defined for* $q = 0, 1, 2, \ldots$ *by*

$$T \in \{\text{interval, triangle, tetrahedron}\}, \tag{3.56}$$
$$\mathcal{V} = \mathcal{P}_q(T), \tag{3.57}$$
$$\ell_i(v) = v(x^i), \tag{3.58}$$

where $\{x^i\}_{i=1}^{n(q)}$ *is an enumeration of points in* T *defined by*

$$x = \begin{cases} i/q, & 0 \leqslant i \leqslant q, & T \text{ interval}, \\ (i/q, j/q) & 0 \leqslant i+j \leqslant q, & T \text{ triangle}, \\ (i/q, j/q, k/q) & 0 \leqslant i+j+k \leqslant q, & T \text{ tetrahedron}. \end{cases} \tag{3.59}$$

The dimension of DG_q is

$$n(q) = \begin{cases} q+1, & T \text{ interval}, \\ \frac{1}{2}(q+1)(q+2), & T \text{ triangle}, \\ \frac{1}{6}(q+1)(q+2)(q+3), & T \text{ tetrahedron}. \end{cases} \tag{3.60}$$

Letting Π_T^q denote the interpolation operator defined by the degrees of freedom above, the interpolation properties of the DG_q elements of degree q are:

$$||u - \Pi_T^q u||_{L^2(T)} \leqslant C\, h_T^{q+1} |u|_{H^{q+1}(T)}. \tag{3.61}$$

3.7 H^2 finite elements

The H^2 elements are commonly used in the approximation of fourth-order problems, or for other spaces requiring at least C^1 continuity. Due to the restrictive nature of the continuity requirement, conforming elements are often of a high polynomial degree, but lower degree nonconforming elements have proven to be successful. Therefore, we here consider the conforming Argyris element and the nonconforming Hermite and Morley elements.

3.7.1 The Argyris element

The Argyris element (Argyris et al., 1968; Ciarlet, 2002) is based on the space $\mathcal{P}_5(T)$ of quintic polynomials over some triangle T. It can be pieced together with full C^1 continuity between elements and C^2 continuity at the vertices of a triangulation.

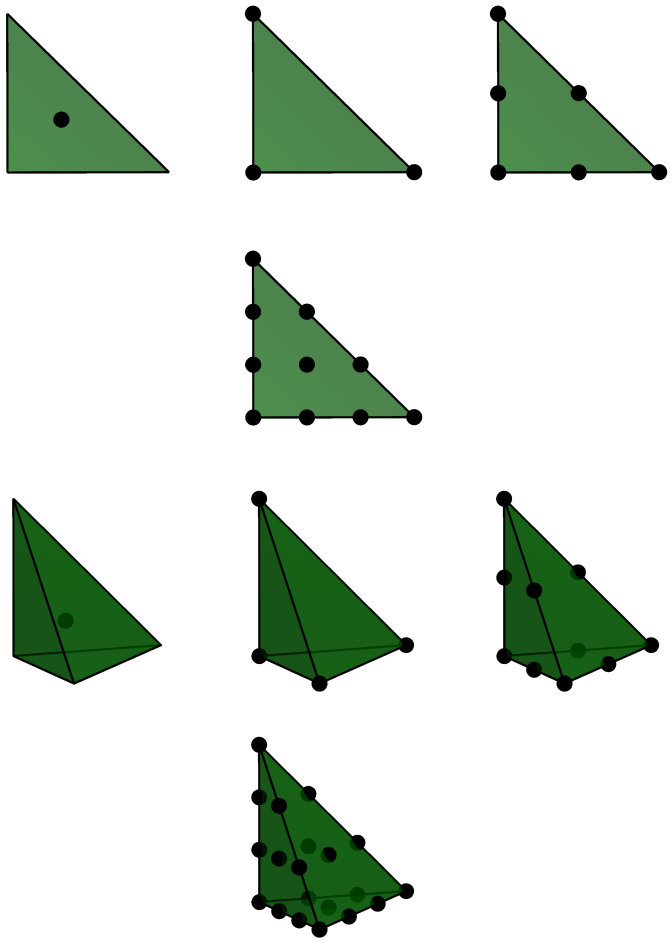


Figure 3.13: Illustration of the zeroth, first, second and third degree discontinuous Lagrange elements on triangles and tetrahedra. The degrees of freedom may be chosen arbitrarily as long as they span the dual space $\mathcal{V}'$. Here, the degrees of freedom have been chosen to be identical to those of the standard Lagrange finite element, with the difference that the degrees of freedom are viewed as *internal* to the element.

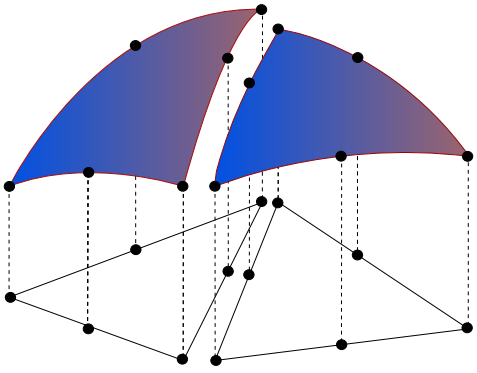


Figure 3.14: All degrees of freedom of a discontinuous Lagrange finite element are internal to the element, which means that no global continuity is imposed by these elements. This is illustrated here for discontinuous quadratic Lagrange elements.

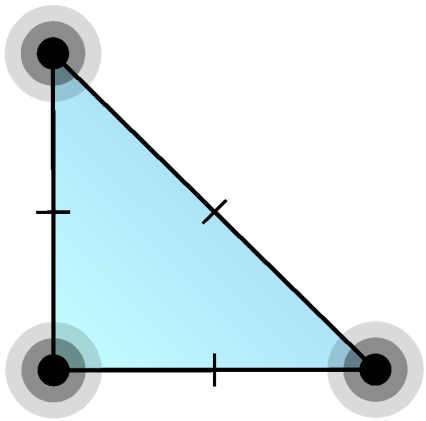


Figure 3.15: The quintic Argyris triangle. The degrees of freedom are point evaluation, point evaluation of both first derivatives and point evaluation of all three second derivatives at the vertices of the triangle, and evaluation of the normal derivative at the midpoint of each edge.

Definition 3.12 (Argyris element) *The (quintic) Argyris element* (ARG_5) *is defined by*

$$T = \text{triangle}, \tag{3.62}$$

$$\mathcal{V} = \mathcal{P}_5(T), \tag{3.63}$$

$$\mathcal{L} = \begin{cases} v(x^i), & \text{for each vertex } x^i, \\ \operatorname{grad} v(x^i)_j, & \text{for each vertex } x^i, \text{ and each component } j, \\ D^2 v(x^i)_{jk}, & \text{for each vertex } x^i, \text{ and each component } jk, j \leqslant k, \\ \operatorname{grad} v(m^i) \cdot n, & \text{for each edge midpoint } m^i. \end{cases} \tag{3.64}$$

The dimension of ARG_5 is

$$n = 21. \tag{3.65}$$

Letting Π_T denote the interpolation operator defined by the degrees of freedom above, the interpolation properties of the (quintic) Argyris elements are (Braess, 2007, Chapter II.6):

$$||u - \Pi_T u||_{H^2(T)} \leqslant C\, h_T^4 |u|_{H^6(T)}, \quad ||u - \Pi_T u||_{H^1(T)} \leqslant C\, h_T^5 |u|_{H^6(T)}, \quad ||u - \Pi_T u||_{L^2(T)} \leqslant C\, h_T^6 |u|_{H^6(T)}. \tag{3.66}$$

The normal derivatives in the dual basis for the Argyris element prevent it from being affine-interpolation equivalent. This prevents the nodal basis from being constructed on a reference cell and affinely mapped. Recent work by Domínguez and Sayas (2008) develops a transformation that corrects this issue and requires less computational effort than directly forming the basis on each cell in a mesh. The Argyris element can be generalized to polynomial degrees higher than quintic, still giving C^1 continuity with C^2 continuity at the vertices (Šolín et al., 2004).

3.7.2 The Hermite element

The Hermite element generalizes the classic cubic Hermite interpolating polynomials on the line segment (Ciarlet, 2002). Hermite-type elements appear in the finite element literature almost from the beginning, appearing at least as early as the classic paper by Ciarlet and Raviart (1972). They have long been known as useful C^1-nonconforming elements (Braess, 2007; Ciarlet, 2002). Under affine mappings, the Hermite elements form *affine-interpolation equivalent* families (Brenner and Scott, 2008).

On the triangle, the space of cubic polynomials is ten-dimensional, and the ten degrees of freedom for the Hermite element are point evaluation at the triangle vertices and barycenter, together with the

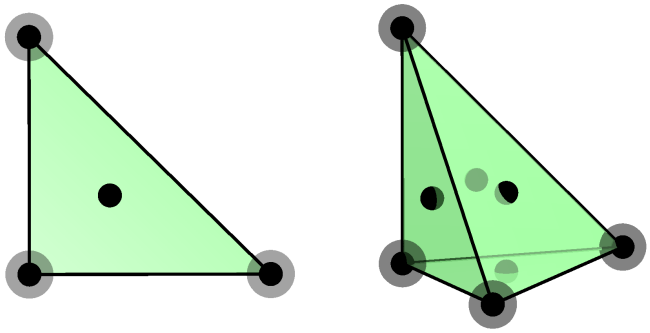


Figure 3.16: The cubic Hermite triangle and tetrahedron. The degrees of freedom are point evaluation at the vertices and the barycenter, and evaluation of both first derivatives at the vertices.

components of the gradient evaluated at the vertices. The generalization to tetrahedra is analogous.

Definition 3.13 (Hermite element) *The (cubic) Hermite element (*HER*) is defined by*

$$T \in \{\text{interval}, \text{triangle}, \text{tetrahedron}\}, \tag{3.67}$$

$$\mathcal{V} = \mathcal{P}_3(T), \tag{3.68}$$

$$\mathcal{L} = \begin{cases} v(x^i), & \text{for each vertex } x^i, \\ \operatorname{grad} v(x^i)_j, & \text{for each vertex } x^i, \text{ and each component } j, \\ v(b), & \text{for the barycenter } b \text{ (of the faces in 3D).} \end{cases} \tag{3.69}$$

The dimension of HER is

$$n = \begin{cases} 10, & T \text{ triangle}, \\ 20, & T \text{ tetrahedron}. \end{cases} \tag{3.70}$$

Letting Π_T denote the interpolation operator defined by the degrees of freedom above, the interpolation properties of the (cubic) Hermite elements are:

$$||u - \Pi_T u||_{H^1(T)} \leqslant C\, h_T^3 |u|_{H^4(T)}, \quad ||u - \Pi_T u||_{L^2(T)} \leqslant C\, h_T^4 |u|_{H^4(T)}. \tag{3.71}$$

Unlike the cubic Hermite functions on a line segment, the cubic Hermite triangle and tetrahedron cannot be patched together in a fully C^1 fashion. The cubic Hermite element can be extended to higher degree (Brenner and Scott, 2008).

3.7.3 The Morley element

The Morley triangle defined in Morley (1968) is a simple H^2-nonconforming quadratic element that is used in fourth-degree problems. The function space $\mathcal{V}$ is simply $\mathcal{P}_2(T)$, the six-dimensional space of quadratics. The degrees of freedom consist of pointwise evaluation at each vertex and the normal derivative at each edge midpoint. It is interesting to note that the Morley triangle is neither C^1 nor even C^0, yet it is suitable for fourth-order problems, and is the simplest known element for this purpose.

The Morley element was first introduced to the engineering literature by Morley (1968, 1971). In the mathematical literature, Lascaux and Lesaint (1975) considered it in the context of the patch test in a study of plate-bending elements. Recent applications of the Morley element include Huang et al. (2008); Ming and Xu (2006).

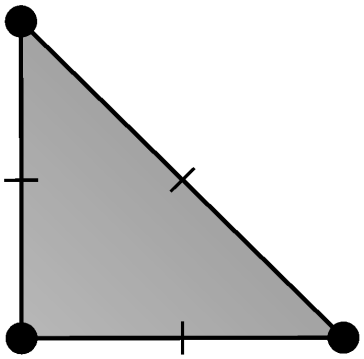


Figure 3.17: The quadratic Morley triangle. The degrees of freedom are point evaluation at the vertices and evaluation of the normal derivative at the midpoint on each edge.

Definition 3.14 (Morley element) *The (quadratic) Morley element (*MOR*) is defined by*

$$T = \text{triangle}, \tag{3.72}$$

$$\mathcal{V} = \mathcal{P}_2(T), \tag{3.73}$$

$$\mathcal{L} = \begin{cases} v(x^i), & \text{for each vertex } x^i, \\ \operatorname{grad} v(m^i) \cdot n, & \text{for each edge midpoint } m^i. \end{cases} \tag{3.74}$$

The dimension of the Morley element is

$$n = 6. \tag{3.75}$$

Letting Π_T denote the interpolation operator defined by the degrees of freedom above, the interpolation properties of the (quadratic) Morley elements are:

$$||u - \Pi_T u||_{H^1(T)} \leqslant C\, h_T^2 |u|_{H^3(T)}, \quad ||u - \Pi_T u||_{L^2(T)} \leqslant C\, h_T^3 |u|_{H^3(T)}. \tag{3.76}$$

3.8 Enriching finite elements

If U, V are linear spaces, one can define a new linear space W by

$$W = \{w = u + v \,:\, u \in U, v \in V\}. \tag{3.77}$$

Here, we choose to call such a space W an *enriched space*.

The enrichment of a finite element space can lead to improved stability properties, especially for mixed finite element methods. Examples include the enrichment of the Lagrange element with bubble functions for use with the Stokes equations or enriching the Raviart–Thomas element for linear elasticity (Arnold et al., 1984a,b). Bubble functions have since been used for many different applications. We here define a *bubble element* for easy reference. Notable examples of the use of a bubble element include:

The MINI element for the Stokes equations. In the lowest degree case, the linear vector Lagrange element is enriched with the cubic vector bubble element for the velocity approximation (Arnold et al., 1984b).

The PEERS element for weakly symmetric linear elasticity. Each row of the stress tensor is approximated by the lowest degree Raviart–Thomas element enriched by the curl of the cubic bubble element (Arnold et al., 1984a).

Definition 3.15 (Bubble element) *The bubble element (B_q) is defined for $q \geqslant (d+1)$ by*

$$T \in \{\text{interval}, \text{triangle}, \text{tetrahedron}\}, \tag{3.78}$$

$$\mathcal{V} = \{v \in \mathcal{P}_q(T) : v|_{\partial T} = 0\}, \tag{3.79}$$

$$\ell_i(v) = v(x^i), \quad i = 1, \ldots, n(q). \tag{3.80}$$

where $\{x^i\}_{i=1}^{n(q)}$ is an enumeration of the points[3] *in T defined by*

$$x = \begin{cases} (i+1)/q, & 0 \leqslant i \leqslant q-2, & T \text{ interval}, \\ ((i+1)/q, (j+1)/q), & 0 \leqslant i+j \leqslant q-3, & T \text{ triangle}, \\ ((i+1)/q, (j+1)/q, (k+1)/q), & 0 \leqslant i+j+k \leqslant q-4, & T \text{ tetrahedron}. \end{cases} \tag{3.81}$$

The dimension of the Bubble element is

$$n(q) = \begin{cases} q-1, & T \text{ interval}, \\ \frac{1}{2}(q-2)(q-1), & T \text{ triangle}, \\ \frac{1}{6}(q-3)(q-2)(q-1), & T \text{ tetrahedron}. \end{cases} \tag{3.82}$$

3.9 Finite element exterior calculus

It has recently been demonstrated that many of the finite elements that have been discovered or invented over the years can be formulated and analyzed in a common unifying framework as special cases of a more general class of finite elements. This new framework is known as *finite element exterior calculus* and is summarized in Arnold et al. (2006a). In finite element exterior calculus, two finite element spaces $\mathcal{P}_q\Lambda^k(T)$ and $\mathcal{P}_q^-\Lambda^k(T)$ are defined for general simplices T of dimension $d \geqslant 1$. The element $\mathcal{P}_q\Lambda^k(T)$ is the space of polynomial differential k-forms[4] on T with degrees of freedom chosen to ensure continuity of the trace on facets. When these elements are interpreted as regular elements, by a suitable identification between differential k-forms and scalar- or vector-valued functions, one obtains a series of well-known elements for $0 \leqslant k \leqslant d \leqslant 3$. In Table 3.2, we summarize the relation between these elements and the elements presented above in this chapter[5].

3.10 Summary

In the table below, we summarize the list of elements discussed in this chapter. For brevity, we include element degrees only up to and including $q = 3$. For higher degree elements, we refer to the script `dolfin-plot` available as part of FEniCS, which can be used to easily plot the degrees of freedom for a wide range of elements:

Bash code

[3] Any other basis for the dual space of $\mathcal{V}$ will work just as well.

[4] A differential k-form ω on a domain Ω maps each point $x \in \Omega$ to an alternating k-form ω_x on the tangent space $T_x(\Omega)$ of Ω at the point x. One can show that for $d = 3$, the differential k-forms correspond to scalar-, vector-, vector-, and scalar-valued functions for $k = 0, 1, 2, 3$ respectively. Thus, we may identify for example both $\mathcal{P}_q\Lambda^1$ and $\mathcal{P}_q\Lambda^2$ on a tetrahedron with the vector-valued polynomials of degree at most q on the tetrahedron.

[5] The finite elements $\mathcal{P}_q\Lambda^k(T)$ and $\mathcal{P}_q^-\Lambda^k(T)$ have been implemented for general values of k, q and $d = 1, 2, 3, 4, \ldots$ as part of the FEniCS *Exterior* package available from `http://launchpad.net/exterior`.

$\mathcal{P}_q\Lambda^k$

k	$d=1$	$d=2$	$d=3$
0	CG_q	CG_q	CG_q
1	DG_q	$\mathrm{NED}_q^{2,\mathrm{curl}}$	$\mathrm{NED}_q^{2,\mathrm{curl}}$
2	—	DG_q	BDM_q
3	—	—	DG_q

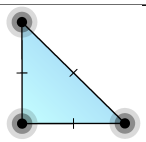
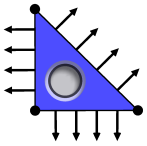
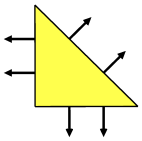
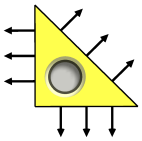
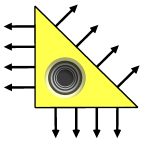
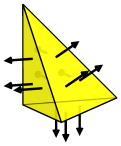
$\mathcal{P}_q^-\Lambda^k$

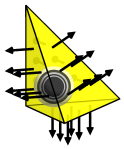
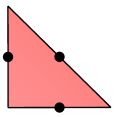
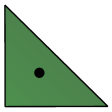
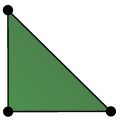
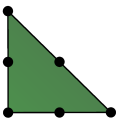
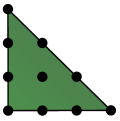
k	$d=1$	$d=2$	$d=3$
0	CG_q	CG_q	CG_q
1	DG_{q-1}	$\mathrm{NED}_q^{1,\mathrm{curl}}$	$\mathrm{NED}_q^{1,\mathrm{curl}}$
2	—	DG_{q-1}	RT_q
3	—	—	DG_{q-1}

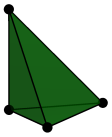
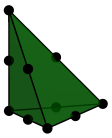
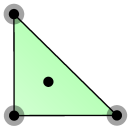
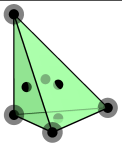
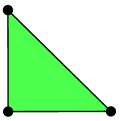
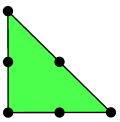
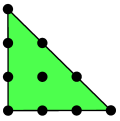
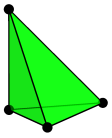
Table 3.2: Relationships between the finite elements $\mathcal{P}_q\Lambda^k$ and $\mathcal{P}_q^-\Lambda^k$ defined by finite element exterior calculus and their more traditional counterparts using the numbering and labeling of this chapter.

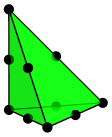
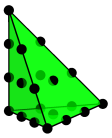
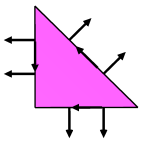
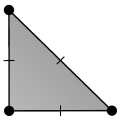
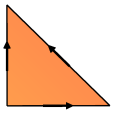
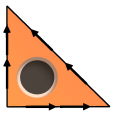
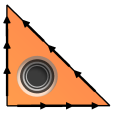
```
$ dolfin-plot BDM tetrahedron 3
$ dolfin-plot N1curl triangle 4
$ dolfin-plot CG tetrahedron 5
```

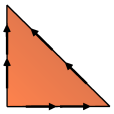
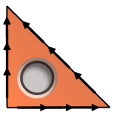
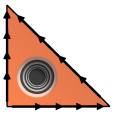
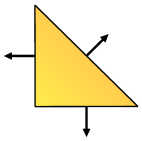
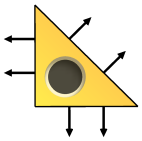
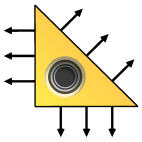
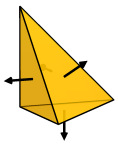
Elements indicated with at $(*)$ in the table below are fully supported by FEniCS.

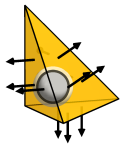
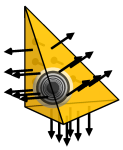
Element family	Notation	Illustration	Dimension	Description
(Quintic) Argyris	ARG_5 (2D)		$n=21$	$\mathcal{P}_5$ (scalar); 3 point values, 3×2 derivatives, 3×3 second derivatives, 3 directional derivatives
Arnold–Winther	AW (2D)		$n=24$	$\mathcal{P}_3(T;\mathbb{S})$ (matrix) with linear divergence; 3×3 point values, 12 normal components, 3 interior moments
Brezzi–Douglas–Marini $(*)$	BDM_1 (2D)		$n=6$	$[\mathcal{P}_1]^2$ (vector); 6 normal components
Brezzi–Douglas–Marini $(*)$	BDM_2 (2D)		$n=12$	$[\mathcal{P}_2]^2$ (vector); 9 normal components, 3 interior moments
Brezzi–Douglas–Marini $(*)$	BDM_3 (2D)		$n=20$	$[\mathcal{P}_3]^2$ (vector); 12 normal components, 8 interior moments
Brezzi–Douglas–Marini $(*)$	BDM_1 (3D)		$n=12$	$[\mathcal{P}_1]^3$ (vector); 12 normal components

Brezzi–Douglas–Marini $(*)$	BDM_2 (3D)		$n = 30$	$[\mathcal{P}_2]^3$ (vector); 24 normal components, 6 interior moments
Brezzi–Douglas–Marini $(*)$	BDM_3 (3D)		$n = 60$	$[\mathcal{P}_3]^3$ (vector); 40 normal components, 20 interior moments
Crouzeix–Raviart $(*)$	CR_1 (2D)		$n = 3$	$\mathcal{P}_1$ (scalar); 3 point values
Crouzeix–Raviart $(*)$	CR_1 (3D)		$n = 4$	$\mathcal{P}_1$ (scalar); 4 point values
Discontinuous Lagrange $(*)$	DG_0 (2D)		$n = 1$	$\mathcal{P}_0$ (scalar); 1 point value
Discontinuous Lagrange $(*)$	DG_1 (2D)		$n = 3$	$\mathcal{P}_1$ (scalar); 3 point values
Discontinuous Lagrange $(*)$	DG_2 (2D)		$n = 6$	$\mathcal{P}_2$ (scalar); 6 point values
Discontinuous Lagrange $(*)$	DG_3 (2D)		$n = 10$	$\mathcal{P}_3$ (scalar); 10 point values
Discontinuous Lagrange $(*)$	DG_0 (3D)		$n = 1$	$\mathcal{P}_0$ (scalar); 1 point value

Discontinuous Lagrange $(*)$	DG_1 (3D)		$n = 4$	$\mathcal{P}_1$ (scalar); 4 point values
Discontinuous Lagrange $(*)$	DG_2 (3D)		$n = 10$	$\mathcal{P}_2$ (scalar); 10 point values
Discontinuous Lagrange $(*)$	DG_3 (3D)		$n = 20$	$\mathcal{P}_3$ (scalar); 20 point values
(Cubic) Hermite	HER (2D)		$n = 10$	$\mathcal{P}_3$ (scalar); 4 point values, 3×2 derivatives
(Cubic) Hermite	HER (3D)		$n = 20$	$\mathcal{P}_3$ (scalar); 8 point values, 4×3 derivatives
Lagrange $(*)$	CG_1 (2D)		$n = 3$	$\mathcal{P}_1$ (scalar); 3 point values
Lagrange $(*)$	CG_2 (2D)		$n = 6$	$\mathcal{P}_2$ (scalar); 6 point values
Lagrange $(*)$	CG_3 (2D)		$n = 10$	$\mathcal{P}_3$ (scalar); 10 point values
Lagrange $(*)$	CG_1 (3D)		$n = 4$	$\mathcal{P}_1$ (scalar); 4 point values

Lagrange $(*)$	CG_2 (3D)		$n = 10$	$\mathcal{P}_2$ (scalar); 10 point values
Lagrange $(*)$	CG_3 (3D)		$n = 20$	$\mathcal{P}_2$ (scalar); 20 point values
Mardal–Tai–Winther	MTW (2D)		$n = 9$	$[\mathcal{P}_2]^2$ (vector); with constant divergence and linear normal components; 6 moments of normal components, 3 moments of tangential components
(Quadratic) Morley	MOR (2D)		$n = 6$	$\mathcal{P}_2$ (scalar); 3 point values, 3 directional derivatives
Nédélec 1st kind $H(\mathrm{curl})$ $(*)$	NED_1^1 (2D)		$n = 3$	$[\mathcal{P}_0]^2 + S_1$ (vector); 3 tangential components
Nédélec 1st kind $H(\mathrm{curl})$ $(*)$	NED_2^1 (2D)		$n = 8$	$[\mathcal{P}_1]^2 + S_2$ (vector); 6 tangential components, 2 interior moments
Nédélec 1st kind $H(\mathrm{curl})$ $(*)$	NED_3^1 (2D)		$n = 15$	$[\mathcal{P}_2]^2 + S_3$ (vector); 9 tangential components, 6 interior moments
Nédélec 1st kind $H(\mathrm{curl})$ $(*)$	NED_1^1 (3D)		$n = 6$	$[\mathcal{P}_0]^3 + S_1$ (vector); 6 tangential components
Nédélec 1st kind $H(\mathrm{curl})$ $(*)$	NED_2^1 (3D)		$n = 20$	$[\mathcal{P}_1]^3 + S_2$ (vector); 20 tangential components

Nédélec 1st kind $H(\mathrm{curl})$ $(*)$	NED_3^1 (3D)		$n = 45$	$[\mathcal{P}_2]^3 + S_3$ (vector); 42 tangential components, 3 interior moments
Nédélec 2nd kind $H(\mathrm{curl})$ $(*)$	NED_1^2 (2D)		$n = 6$	$[\mathcal{P}_1]^2$ (vector); 6 tangential components
Nédélec 2nd kind $H(\mathrm{curl})$ $(*)$	NED_2^2 (2D)		$n = 12$	$[\mathcal{P}_2]^2$ (vector); 9 tangential components, 3 interior moments
Nédélec 2nd kind $H(\mathrm{curl})$ $(*)$	NED_3^2 (2D)		$n = 20$	$[\mathcal{P}_3]^2$ (vector); 12 tangential components, 8 interior moments
Nédélec 2nd kind $H(\mathrm{curl})$ $(*)$	NED_1^2 (3D)		$n = 12$	$[\mathcal{P}_1]^3$ (vector); 12 tangential components
Raviart–Thomas $(*)$	RT_1 (2D)		$n = 3$	$[\mathcal{P}_0]^2 + x\mathcal{P}_0$ (vector); 3 normal components
Raviart–Thomas $(*)$	RT_2 (2D)		$n = 8$	$[\mathcal{P}_1]^2 + x\mathcal{P}_1$ (vector); 6 normal components, 2 interior moments
Raviart–Thomas $(*)$	RT_3 (2D)		$n = 15$	$[\mathcal{P}_2]^2 + x\mathcal{P}_2$ (vector); 9 normal components, 6 interior moments
Raviart–Thomas $(*)$	RT_1 (3D)		$n = 4$	$[\mathcal{P}_0]^3 + x\mathcal{P}_0$ (vector); 4 normal components

Raviart–Thomas $(*)$	RT_2 (3D)		$n = 15$	$[\mathcal{P}_1]^3 + x\mathcal{P}_1$ (vector); 12 normal components, 3 interior moments
Raviart–Thomas $(*)$	RT_3 (3D)		$n = 36$	$[\mathcal{P}_2]^3 + x\mathcal{P}_2$ (vector); 24 normal components, 12 interior moments

4 Constructing general reference finite elements

By Robert C. Kirby and Kent-Andre Mardal

This chapter describes the mathematical framework for constructing a general class of finite elements on reference domains. This framework is used by both the FIAT and SyFi projects, see the Chapters 13 and 15, respectively. Our goal is to provide a framework by which simplicial finite elements with very complicated bases can be constructed automatically. We work from the classic Ciarlet definition of the finite element and its "nodal" basis (an abstract notion far more general and powerful than standard node-oriented Lagrange polynomials).

To date, our methodology does not include spline-type spaces such as are becoming widely popular in isogeometric analysis (Hughes et al., 2005), nor does it entirely address XFEM (Chessa et al., 2002) or *hp*-type methods (Schwab, 1998). However, in isogeometric analysis, the basis functions are readily defined by simple recurrence relations from the theory of splines, so a tool like FIAT or SyFi is not necessary. XFEM typically works by enriching existing finite element spaces with special basis functions to capture singular behavior – our approach can provide the regular basis but not the "extra" functions. Finally, handling the constraints imposed in *hp* methods is possible, but unwieldy, with our methodology, but tetrahedral *hp* bases are available (Ainsworth and Coyle, 2003). We return to some of these issues later.

4.1 Background

The finite element literature contains a huge collection of approximating spaces and degrees of freedom, many of which are surveyed in Chapter 3. Some applications, such as Cahn-Hilliard and other fourth-order problems, can benefit from very smooth finite element bases, while porous media flow requires vector fields discretized by piecewise polynomial functions with only normal components continuous across cell boundaries. Many problems in electromagnetism call for the tangentially continuous vector fields obtained by using Nédélec elements (Nédélec, 1980, 1986). Many elements are carefully designed to satisfy an *inf-sup* condition (Brezzi and Fortin, 1991; Girault and Raviart, 1986), originally developed to explain stability of discretizations of incompressible flow problems. Additionally, some problems call for low-order discretizations, while others are effectively solved with high-order polynomials.

While the automatic generation of computer programs for finite element methods requires one to confront the panoply of finite element families found in the literature, it also provides a pathway for wider employment of Raviart–Thomas, Nédélec, and other difficult-to-program elements. Ideally, one would like to describe the diverse finite element spaces at an abstract level, whence a computer code discerns how to evaluate and differentiate their basis functions. Such goals are in large part accomplished by the FIAT and SyFi projects, whose implementations are described in the chapters 13 and 15.

Projects like FIAT and SyFi may ultimately remain mysterious to the end user of a finite element system, as interactions with finite element bases are typically mediated through tools that construct the global finite element operators. The end user will typically be satisfied if two conditions are met. First, a finite element system should support the common elements used in the application area of interest. Second, it should provide flexibility with respect to order of approximation.

It is entirely possible to satisfy many users by *a priori* enumerating a list of finite elements and implement only those. At certain times, this would even seem ideal. For example, after the rash of research that led to elements such as the Raviart–Thomas-Nédélec and Brezzi–Douglas–Marini families, development of new families slowed considerably. Then, more recent work lead forth by Arnold, Falk, and Winther in the context of exterior calculus has not only led to improved understanding of existing element families, but has also brought a new wave of elements with improved properties, see Arnold et al. (2006a) for an overview. A generative system for finite element bases can far more readily assimilate these and future developments. Automation also provides generality with respect to the order of approximation that standard libraries might not otherwise provide. Finally, the end-user might even easily define his own new element and test its numerical properties before analyzing it mathematically.

In the present chapter, we describe the mathematical formulation underlying such projects as FIAT, SyFi and Exterior (Logg and Mardal, 2009). This formulation starts from definitions of finite elements as given classically by Ciarlet (2002). It then uses basic linear algebra to construct the appropriate basis for an abstract finite element in terms of polynomials that are easy to implement and well-behaved in floating point arithmetic. We focus on constructing nodal bases for a single, fixed reference element. As we will see in the Chapters 11 and 15, form compilers such as FFC and SFC will work in terms of this single reference element.

Other approaches exist in the literature, such as the hierarchical bases studied by Szabó et al. (1991) and extended to $H(\mathrm{curl})$ and $H(\mathrm{div})$ spaces in work such as Ainsworth and Coyle (2003). These approaches can provide greater flexibility for refining the mesh and polynomial degree in finite element methods, but they also require more care during assembly and are typically constructed on a case-by-case basis for each element family. When they are available, they may be cheaper to construct than using the technique studied here, but this present technique is easier to apply to an "arbitrary" finite element and so is considered in the context of automatic software.

4.2 Preliminaries

Both FIAT and SyFi work with a slightly modified version of the abstract definition of a finite element introduced by Ciarlet.

Definition 4.1 (Finite element (Ciarlet, 2002)) *A finite element is defined by a triple $(T, \mathcal{V}, \mathcal{L})$, where*

- *the domain T is a bounded, closed subset of $\mathbb{R}^d$ (for $d = 1, 2, 3, \ldots$) with nonempty interior and piecewise smooth boundary;*
- *the space $\mathcal{V} = \mathcal{V}(T)$ is a finite dimensional function space on T of dimension n;*
- *the set of degrees of freedom (nodes) $\mathcal{L} = \{\ell_1, \ell_2, \ldots, \ell_n\}$ is a basis for the dual space $\mathcal{V}'$; that is, the space of bounded linear functionals on $\mathcal{V}$.*

In this definition, the term "finite element" refers not only to a particular cell in a mesh, but also to the associate function space and degrees of freedom. Typically, the domain T is some simple polygonal or polyhedral shape and the function space $\mathcal{V}$ consists of polynomials.

Given a finite element, a concrete basis, often called the nodal basis, for this element can be computed by using the following definition.

Definition 4.2 *The* nodal basis *for a finite element* $(T, \mathcal{V}, \mathcal{L})$ *is the set of functions* $\{\phi_i\}_{i=1}^n$ *such that for all* $1 \leqslant i, j \leqslant n$,

$$l_i(\phi_j) = \delta_{ij}, \tag{4.1}$$

where δ_{ij} *denotes the Kronecker delta function.*

The main issue at hand in this chapter is the *construction* of this nodal basis. For any given finite element, one may construct the nodal basis explicitly with elementary algebra. However, this becomes tedious as we consider many different families of elements and want arbitrary order instances of each family. Hence, we present a new paradigm here that undergirds computer programs for automating the construction of nodal bases.

In addition to the construction of the nodal base we need to keep in mind that finite elements are patched together to form a piecewise polynomial field over a mesh. The fitness (or stability) of a particular finite element method for a particular problem relies on the continuity requirements of the problem. The degrees of freedom of a particular element are often chosen such that these continuity requirements are fulfilled.

Hence, in addition to computing the nodal basis, the framework developed here simplifies software for the following tasks:

1. Evaluate the basis functions and their derivatives at points.

2. Associate the basis functions (or degrees of freedom) with topological facets of T such as its vertices, edges and its placement on the edges.

3. Associate each basis function with additional metadata that describes the mapping that should be used for the evaluation of the basis functions and their derivatives.

4. Provide rules for evaluating the degrees of freedom applied to arbitrary functions (needed for Dirichlet boundary conditions).

The first of these is relatively simple in the framework of symbolic computation (SyFi), but they require more care if an implementation uses numerical arithmetic (FIAT). The middle two encode the necessary information for a client code to transform the reference basis and assemble global degrees of freedom when the finite element is either less or more than C^0 continuous. The final task may take the form of a point at which data is evaluated or differentiated or more generally as the form of a sum over points and weights, much like a quadrature rule.

4.3 Mathematical framework

4.3.1 Change of basis

The fundamental idea in constructing a nodal basis is from elementary linear algebra: one constructs the desired (nodal) basis as a linear combination of another available basis. We will start with some basis $\{\psi_i\}_{i=1}^n$ that spans $\mathcal{V}$. From this, we construct each nodal basis function ϕ_j as

$$\phi_j = \sum_{k=1}^{n} \alpha_{jk} \psi_k, \tag{4.2}$$

The task is to compute the matrix α. Each fixed ϕ_j must satisfy

$$\ell_i(\phi_j) = \delta_{ij}, \tag{4.3}$$

and using the above expansion for ϕ_j, we obtain

$$\delta_{ij} = \sum_{k=1}^{n} \ell_i(\alpha_{jk}\psi_k) = \sum_{k=1}^{n} \alpha_{jk}\ell_i(\psi_k). \tag{4.4}$$

So, for a fixed j, we have a system of n equations

$$\sum_{k=1}^{n} B_{ik}\alpha_{jk} = \delta_{ij}, \tag{4.5}$$

where

$$B_{ik} = \ell_i(\psi_k) \tag{4.6}$$

is a kind of generalized Vandermonde matrix. Of course, (4.5) can be written as

$$B\alpha^\top = I, \tag{4.7}$$

and we obtain

$$\alpha = B^{-T}. \tag{4.8}$$

In practice, this supposes that one has an implementation of the original basis for which the actions of the degrees of freedom may be readily computed. The degrees of freedom typically involves point evaluation, differentiation, integration, and so on.

4.3.2 Polynomial spaces

In Definition 4.1 we defined the finite element in terms of a finite dimensional function space $\mathcal{V}$. Although it is not strictly necessary, the functions used in finite elements are typically polynomials. While our general strategy will in principle accommodate non-polynomial bases, we only deal with polynomials in this chapter. The most common space is $[\mathcal{P}_q]^d$, the space of polynomials of degree q in $\mathbb{R}^d$. There are many different ways to represent $[\mathcal{P}_q]^d$. We will discuss the power and Bernstein bases, and orthogonal bases such as Dubiner, Jacobi, and Legendre. Each of these bases has explicit representations or recurrence relations making them easy to evaluate and differentiate. In contrast, most finite element bases are determined by solving the linear system in Definition 4.2. In addition to $[\mathcal{P}_q]^d$ we will also for some elements need $\mathcal{H}_q^d$, the space of homogeneous polynomials of degree q in d variables.

Typically, the techniques developed here are used on simplices, where polynomials do not have a nice tensor-product structure. SyFi does, however, have support for rectangular domains, while FIAT does not.

Power basis. On a line segment, the monomial or power basis $\{x^i\}_{i=0}^{q}$ spans $\mathcal{P}_q$, so that any $\psi \in \mathcal{P}_q$ can be written as

$$\psi = a_0 + a_1 x + \ldots a_q x^q = \sum_{i=0}^{q} a_i x^i. \tag{4.9}$$

In 2D on triangles, $[\mathcal{P}_q]^2$ is spanned by functions on the form $\{x^i y^j\}_{i,j=0}^{i+j\leqslant q}$, with a similar definition in three dimensions.

This basis is quite easy to evaluate, differentiate, and integrate. But the basis is very ill-conditioned in numerical calculations. For instance, the condition number of the mass matrix using the power basis in $\mathcal{P}_{10}$ gives a condition number of $5 \cdot 10^{14}$, while corresponding condition numbers are $4 \cdot 10^6$

and $2 \cdot 10^3$ for the Bernstein and Lagrange polynomials, respectively.

Legendre basis. A popular polynomial basis for polygons that are either intervals, rectangles or boxes are the Legendre polynomials. This polynomial basis is also usable to represent polynomials of high degree. The basis is defined on the interval $[-1, 1]$, as

$$\psi_i(x) = \frac{1}{2^i i!} \frac{d^i}{dx^i}(x^2 - 1)^i, \quad i = 0, 1, \ldots, \tag{4.10}$$

A nice feature with these polynomials is that they are orthogonal with respect to the L^2 inner product; that is,

$$\int_{-1}^{1} \psi_i(x)\psi_j(x)\,\mathrm{d}x = \begin{cases} \frac{2}{2i+1}, & i = j, \\ 0, & i \neq j, \end{cases} \tag{4.11}$$

The Legendre polynomials can be extended to rectangular domains in any dimensions by tensor–products. For instance, in 2D the basis reads,

$$\psi_{ij}(x, y) = \psi_i(x)\psi_j(y), \quad i, j \leqslant q. \tag{4.12}$$

Recurrence relations for these polynomials can be found in Karniadakis and Sherwin (2005).

Jacobi basis. The Jacobi polynomials $P_i^{\alpha,\beta}(x)$ generalize the Legendre polynomials, giving orthogonality with respect to a weighted inner product. In particular, $\int_{-1}^{1}(1-x)^\alpha(1+x)^\beta P_i^{\alpha,\beta} P_j^{\alpha,\beta}\,\mathrm{d}x = 0$ unless $i = j$. The polynomials are given by

$$\begin{aligned} P_0^{\alpha,\beta} &= 1 \\ P_1^{\alpha,\beta} &= \frac{1}{2}\left(\alpha - \beta + (\alpha + \beta + 2)\, x\right), \end{aligned} \tag{4.13}$$

with a three-term recurrence for $i \geqslant 1$:

$$P_{i+1}^{\alpha,\beta}(x) = (a_i^{\alpha,\beta} x + b_i^{\alpha,\beta}) P_i^{\alpha,\beta}(x) - c_i^{\alpha,\beta} P_{i-1}^{\alpha,\beta}(x). \tag{4.14}$$

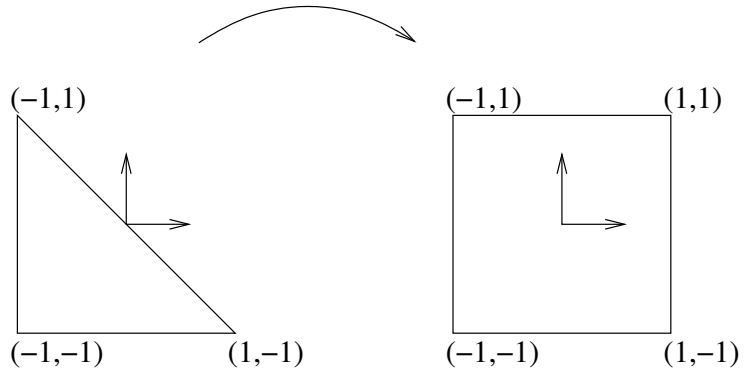
General Jacobi polynomials are used in 1d and tensor-product domains far less frequently than Legendre polynomials, but they play an important role in constructing orthogonal bases on the simplex, to which we now turn.

Dubiner basis. Orthogonal polynomials in simplicial domains are also known, although they lack some of the rotational symmetry of the Legendre polynomials. The Dubiner basis, frequently used in simplicial spectral elements (Dubiner, 1991), is known under many names in the literature. It is an L^2-orthogonal basis that can be constructed by mapping particular tensor products of Jacobi polynomials on a square by a singular coordinate change to a fixed triangle. Let $P_n^{\alpha,\beta}$ denote the n^{th} Jacobi polynomial with weights α, β. Then, define the new coordinates

$$\begin{aligned} \eta_1 &= 2\left(\frac{1+x}{1-y}\right) - 1 \\ \eta_2 &= y, \end{aligned} \tag{4.15}$$

which map the triangle with vertices $(-1,-1), (-1,1), (1,-1)$ to the square $[-1,1]^2$ as shown in Figure 4.1. This is the natural domain for defining the Dubiner polynomials, but they may easily be

Figure 4.1: Reference triangular and square domains with collapsed coordinate transformation.



mapped to other domains like the triangle with vertices $(0,0),(0,1),(1,0)$ by an affine mapping. Then, one defines

$$\psi_{ij}(x,y) = P_i^{0,0}(\eta_1)\left(\frac{1-\eta_2}{2}\right)^i P_j^{2i+1,0}(\eta_2). \tag{4.16}$$

Though it is not obvious from the definition, $\psi_{ij}(x,y)$ is a polynomial in x and y of degree $i+j$. Moreover, for $(i,j) \neq (p,q)$, ψ_{ij} is L^2-orthogonal to ψ_{pq}.

While this basis is more complicated than the power basis, it is very well-conditioned for numerical calculations even with high degree polynomials. The polynomials can also be ordered hierarchically so that $\{\psi_i\}_{i=1}^n$ forms a basis for $\mathcal{P}_{n-1}$ for each $n > 1$. As a possible disadvantage, the basis lacks rotational symmetry that can be found in other bases.

Bernstein basis. The Bernstein basis is another well-conditioned basis that can be used in generating finite element bases. In 1D, the basis functions in $\mathcal{P}_q$ take the form,

$$\psi_i^q = \binom{q}{i} x^i (1-x)^{q-i}, \quad i = 0,\ldots,q, \tag{4.17}$$

and then $\mathcal{P}_q$ is spanned by $\{\psi_i^q\}_{i=0}^q$.

Notice that the Bernstein basis consists of powers of x and $1-x$, which are the barycentric coordinates for $[0,1]$, an observation that makes it easy to extend the basis to simplices in higher dimensions. Let b_1, b_2, and b_3 be the barycentric coordinates for the reference triangle; that is, $b_1 = 1-x-y$, $b_2 = x$, and $b_3 = y$. Then the basis is of the form,

$$\psi_{ijk}^q = \frac{q!}{i!j!k!} b_1^i b_2^j b_3^k, \quad \text{for } i+j+k = q, \tag{4.18}$$

and a basis for $\mathcal{P}_q$ is simply.

$$\{\psi_{ijk}^q\}_{i,j,k\geqslant 0}^{i+j+k=q}. \tag{4.19}$$

The Bernstein polynomials on the tetrahedron and even higher dimensional simplices are completely analogous.

These polynomials, though less common in the finite element community, are well-known in graphics and splines. They have rotational symmetry and are nonnegative and so give positive mass matrices, though they are not hierarchical. Recently, Kirby (2011, 2010b) has analyzed finite element

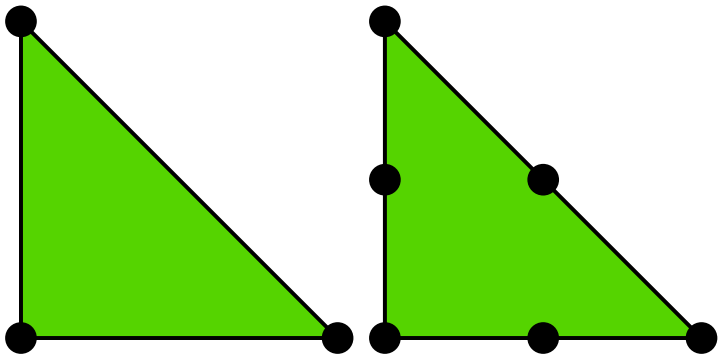


Figure 4.2: Lagrange elements of order one and two.

operators based on Bernstein polynomials. In these papers, particular properties of the Bernstein polynomials are exploited to develop algorithms for matrix-free application of finite element operators with complexity comparable to spectral elements.

Homogeneous polynomials. Another set of polynomials which sometimes is useful is the set of homogeneous polynomials $\mathcal{H}_q$. These are polynomials where all terms have the same degree. $\mathcal{H}_q$ is in 2D spanned by polynomials on the form:

$$\{x^i y^j\}_{i+j=q} \tag{4.20}$$

with a similar definition in dD.

Vector or tensor-valued polynomials. It is straightforward to generalize the scalar-valued polynomials discussed earlier to vector or tensor-valued polynomials. Let $\{e_i\}$ be canonical basis in $\mathbb{R}^d$. Then a basis for the vector-valued polynomials is

$$\phi_{ij} = \phi_j \mathbf{e}_i, \tag{4.21}$$

with a similar definition extending the basis to tensors.

4.4 Examples of elements

We include some standard finite elements to illustrate the concepts. We refer the reader to Chapter 3 for a more thorough review of elements and their properties.

Example 4.1 The Lagrange Element
The Lagrange element shown in Figure 4.2 is the most common element. The degrees of freedom are represented by black dots, which represent point evaluation. The first order element is shown in the leftmost triangle, its degrees of freedom consist of a point evaluation in each of the vertices. That is, the degrees of freedom $\ell_i : \mathcal{V} \to \mathbb{R}$ *are*

$$\ell_i(v) = \int_T v\, \delta_{x_i}\, \mathrm{d}x = v(x_i), \tag{4.22}$$

where x_i *are the vertices (0,0), (1,0), (0,1) and* δ *is the Dirac delta function. The corresponding basis functions are* $1 - x - y$, x, *and* y. *The second order element is shown in right triangle. It has six degrees of freedom, three at the vertices and three at the edges, all are point evaluations. The Lagrange element produces piecewise continuous polynomials and they are therefore well suited for approximation in* H^1. *The Lagrange element of order* q *spans* $\mathcal{P}_q$ *on simplices in any dimension.*

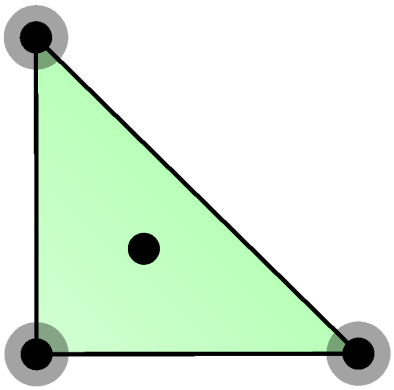


Figure 4.3: Hermite elements of order 3.

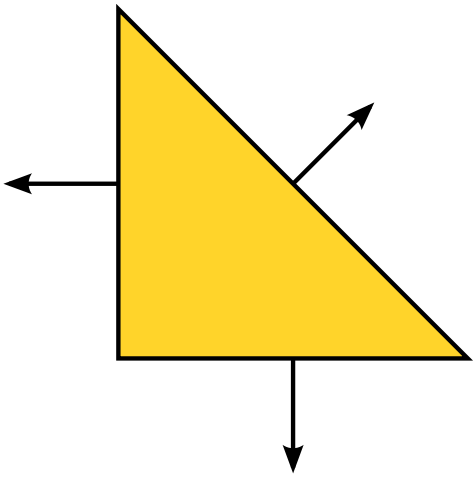


Figure 4.4: Triangular Raviart–Thomas elements of order one.

Example 4.2 The Hermite Element
In Figure 4.3 we show the Hermite element on the reference triangle in 2D. The black dots mean point evaluation, while the white circles mean evaluation of derivatives in both x and y direction. That is, the degrees of freedom $\ell_{i_k} : \mathcal{V} \to \mathbb{R}$ *associated with the vertex* x_i *are,*

$$\ell_{i_1}(v) = \int_T v\,\delta_{x_i}\,\mathrm{d}x = v(x_i), \tag{4.23}$$

$$\ell_{i_2}(v) = \int_T \frac{\partial v}{\partial x}\,\delta_{x_i}\,\mathrm{d}x = \frac{\partial}{\partial x}v(x_i), \tag{4.24}$$

$$\ell_{i_3}(v) = \int_T \frac{\partial v}{\partial y}\,\delta_{x_i}\,\mathrm{d}x = \frac{\partial}{\partial y}v(x_i). \tag{4.25}$$

In addition, there is one internal point evaluation, which in total gives ten degrees of freedom, which is the same number of degrees of freedom as in $\mathcal{P}_3$*. One feature of the Hermite element is that it has continuous derivatives at the vertices (it will however not necessarily result in a* H^2*-conforming approximation).*

Example 4.3 The Raviart–Thomas Element
In Figure 4.4 we illustrate the lowest order Raviart–Thomas element. In contrast to the previous elements, this

element has a vector-valued function space. The arrows represent normal vectors; that is, the degrees of freedom $\ell_i : \mathcal{V} \to \mathbb{R}$ *are*

$$\ell_i(v) = \int_T v \cdot n_i \, \mathrm{d}x, \tag{4.26}$$

where n_i *is the outward normal vector on edge* i*. The Raviart–Thomas element is a vector space with three degrees of freedom. Hence, the standard basis* $[\mathcal{P}_q]^d$ *is not a suitable starting point and we use* $\mathcal{V} = [\mathcal{P}_0]^2 \oplus x\mathcal{H}_0$ *instead. The Raviart–Thomas element is typically used for approximations in* $H(\mathrm{div})$*. We remark that this element may also be defined in terms of point evaluations of normal components.*

4.4.1 Bases for other polynomial spaces

The basis presented above are suitable for constructing many finite elements, but as we have just seen, they do not work in all cases. The Raviart–Thomas function space in 2D is spanned by

$$[\mathcal{P}_n]^2 \oplus \begin{pmatrix} x \\ y \end{pmatrix} \mathcal{H}_n^2. \tag{4.27}$$

Hence, this element requires a basis for vectors of polynomials $(\mathcal{P}_n^2)^2$ enriched with $\begin{pmatrix} x \\ y \end{pmatrix} \mathcal{H}_n^2$. On the other hand, the Brezzi–Douglas–Fortin–Marini on triangle is defined as

$$\left\{ u \in [\mathcal{P}_n(T)]^2 : u \cdot n \in \mathcal{P}_{n-1}(E_i), \quad E_i \in \mathcal{E}(T) \right\}, \tag{4.28}$$

where $\mathcal{E}(T)$ denotes the facets of T.

Hence, this element requires that some functions are removed from $[\mathcal{P}_n(T)]^2$. The removal is expressed by the constraint $u \cdot n \in \mathcal{P}_{n-1}(E_i)$.

Obtaining a basis for this space is somewhat more subtle. FIAT and SyFi have developed different but mathematically equivalent solutions. In SyFi, since it uses a symbolic representation, the polynomial may be easily expressed in the power basis and the coefficients corresponding to n'th order polynomials normal to the edges are set to zero. In a similar fashion, FIAT utilizes the orthogonality of the Legendre polynomials to express the constraints the edges. That is, on the edge E_i the following constraints apply:

$$\ell_i^C(u) = \int_{E_i} (u \cdot n) \mu_n^i = 0, \tag{4.29}$$

where μ_n^i is the n'th order Legendre polynomial on the edge E_i.

In general, assume that we have m constraints and $n - m$ degrees of freedom. Let

$$V_{ij}^1 = \ell_i(\phi_j), \quad 1 \le i \le n - m, \; 1 \le j \le n, \tag{4.30}$$

$$V_{ij}^2 = \ell_i^C(\phi_j), \quad n - m < i \le n, \; 1 \le j \le n. \tag{4.31}$$

and ,

$$V = \begin{pmatrix} V^1 \\ V^2 \end{pmatrix}. \tag{4.32}$$

Consider now the matrix equation

$$V\alpha^\top = I^{n,n-m}, \tag{4.33}$$

where $I^{n,n-m}$ denotes the $n \times n - m$ identity matrix. As before, the columns of α still contain the expansion coefficients of the nodal basis functions ψ_i in terms of $\{\phi_j\}$. Moreover, $V_2\alpha = 0$, which

implies that the nodal basis functions fulfill the constraint.

Other examples than the Brezzi–Douglas–Fortin–Marini element that are defined in terms of constrained polynomials are the Nédélec (Nédélec, 1980), Arnold-Winther (Arnold and Winther, 2002), Mardal-Tai-Winther (Mardal et al., 2002), Tai-Winther (Tai and Winther, 2006), and Bell (Ciarlet, 2002) element families.

4.5 Operations on the polynomial spaces

Here, we show how various important operations may be cast in terms of linear algebra operations, supposing that the operations may be performed on the original basis $\{\psi_i\}_{i=1}^n$.

4.5.1 Evaluation

In order to evaluate the nodal basis $\{\phi_i\}_{i=1}^n$ at a given point $x \in T$, one simply computes the vector

$$\Psi_i = \psi_i(x) \tag{4.34}$$

followed by the product

$$\phi_i(x) \equiv \Phi_i = \sum_j \alpha_{ij} \Psi_j. \tag{4.35}$$

Generally, the nodal basis functions are required at an array of points $\{x_j\}_{j=1}^m \subset T$. For performance reasons, performing matrix-matrix products may be advantageous. So, define $\Psi_{ij} = \Psi_i(x_j)$ and $\Phi_{ij} = \Phi_i(x_j)$. Then all of the nodal basis functions may be evaluated by the product

$$\Phi_{ij} = \sum_k \alpha_{ik} \Psi_{kj}. \tag{4.36}$$

4.5.2 Differentiation

Differentiation is more complicated and presents more options. Let $\alpha = (\alpha_1, \alpha_2, \ldots \alpha_d)$ be a multi-index so that

$$D^\alpha \equiv \frac{\partial^{|\alpha|}}{\partial x_1^{\alpha_1} \partial x_2^{\alpha_2} \ldots \partial x_d^{\alpha_d}}, \tag{4.37}$$

where $|\alpha| = \sum_{i=1}^d \alpha_i$ and we want to compute the array

$$\Phi_i^\alpha = D^\alpha \phi_i(x) \tag{4.38}$$

for some $x \in T$.

One obvious option is to differentiate the original basis functions $\{\psi_i\}$ to produce an array

$$\Psi_i^\alpha = D^\alpha \psi_i(x), \tag{4.39}$$

whence

$$\Phi_i^\alpha = \sum_j \alpha_{ij} \Psi_{ji}^\alpha. \tag{4.40}$$

This presupposes that one may conveniently compute all derivatives of the $\{\psi_i\}$. This is typically true in symbolic computation or when using the power basis. For the Bernstein, Jacobi, and Legendre polynomials recurrence relations are available, see Karniadakis and Sherwin (2005); Kirby (2010b). The Dubiner basis, as typically formulated, contains a coordinate singularity that prevents automatic

differentiation from working at the top vertex. Recent work by Kirby (2010a) has reformulated recurrence relations to allow for this possibility.

If one prefers (or is required by the particular starting basis), one may also compute matrices that encode first derivatives acting on the $\{\phi_i\}$ and construct higher derivatives than these. For each coordinate direction x_k, a matrix D^k is constructed so that

$$\frac{\partial \phi_i}{\partial x_i} = D^k_{ij}\phi_j. \tag{4.41}$$

How to do this depends on which bases are chosen. For particular details on the Dubiner basis, see Dubiner (1991).

4.5.3 Integration

Integration of basis functions over the reference domain, including products of basis functions and/or their derivatives, may be performed numerically, symbolically, or exactly with some known formula. In general, quadrature is easily performed. Quadrature rules for a variety of reference elements may be obtained from for example (Dunavant, 1985; Keegan et al., 2008; Šolín et al., 2004).

4.5.4 Association with facets

As we saw in the definition of for instance the Brezzi–Douglas–Marini element, it is necessary to have polynomials that can be associated with the facets of a polygonal domain. The Bernstein polynomials are expressed via barycentric coordinates and are therefore naturally associated with the facets. The Legendre and Jacobi polynomials are also easy to associated to 1D facets in barycentric coordinates.

4.5.5 Linear functionals

Linear functionals are usually cast in terms of linear combinations of integration, pointwise evaluation and differentiation.

4.5.6 The mapping of the reference element

A common practice, employed throughout the FEniCS software and in many other finite element codes, is to map the nodal basis functions from the reference cell to each cell in a mesh. Sometimes, this is as simple as an affine change of coordinates; in other cases it is more complicated. For completeness, we briefly describe the basics of creating the global finite elements in terms of a mapped reference element. Let therefore T be a global polygon in the mesh and $\hat{T}$ be the corresponding reference polygon. Between the coordinates $x \in T$ and $\hat{x} \in \hat{T}$ we use the mapping

$$x = F_T(\hat{x}) = A_T(\hat{x}) + x_0, \tag{4.42}$$

The Jacobian of this mapping is:

$$J(\hat{x}) = \frac{\partial x}{\partial \hat{x}} = \frac{\partial A_T(\hat{x})}{\partial \hat{x}}. \tag{4.43}$$

Currently, FEniCS only supports affine maps between T and $\hat{T}$, which means that $x = F_T(\hat{x}) = A_T\hat{x} + x_0$ and $J = A_T$. For isoparametric elements, a basis function is defined in terms of the corresponding basis function on the reference element as

$$\phi(x) = \hat{\phi}(\hat{x}). \tag{4.44}$$

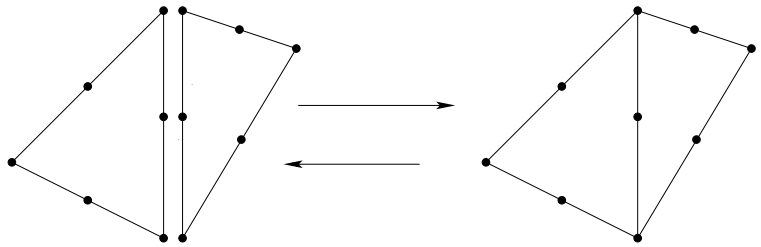


Figure 4.5: Patching together a pair of quadratic local function spaces on a pair of cells to form a global continuous piecewise quadratic function space.

The integral can then be performed on the reference polygon,

$$\int_T \phi(x)\,\mathrm{d}x = \int_{\hat{T}} \hat{\phi}(\hat{x})\,detJ\,\mathrm{d}\hat{x}, \tag{4.45}$$

and the spatial derivatives are defined by the derivatives on the reference element and the geometry mapping by using the chain rule,

$$\frac{\partial \phi}{\partial x_i} = \sum_j \frac{\partial \hat{\phi}}{\partial \hat{x}_j}\frac{\partial \hat{x}_j}{\partial x_i}. \tag{4.46}$$

The above mapping of basis functions is common for approximations in H^1. For approximations in $H(\mathrm{div})$ or $H(\mathrm{curl})$ it is necessary to use the Piola mapping, where the mapping for the basis functions differs from the geometry mapping. That is, for $H(\mathrm{div})$ elements, the Piola mapping reads

$$\phi(x) = \frac{1}{|detJ|} J\hat{\phi}(\hat{x}), \tag{4.47}$$

When using the numbering of mesh entities used by UFC, see Chapter 16, it is advantageous to use $\frac{1}{detJ}$ instead of $\frac{1}{|detJ|}$ since the sign of the determinant relates to the sign of the normal vector, see Rognes et al. (2009) for more details on the Piola mapping and its implementation in FFC. Some elements like the Rannacher-Turek element (Turek, 1999; Rannacher and Turek, 1992) has far better properties when defined globally compared to its analogous definition in terms of a reference element.

4.5.7 Local to global mapping of degrees of freedom

As shown in Figure 4.5, finite elements are patched together with a continuity depending on the degrees of freedom. To obtain the desired patching, the elements should provide identifiers that determine whether the degrees of freedom of some neighboring elements should be shared or not. One alternative is to relate each degree of freedom on the reference cell to a point in the reference cell. The geometry mapping then gives a global point in the mesh, by (4.42), that identifies the degree of freedom; that is, the degrees of freedom in different elements are shared if they correspond to the same global point in the mesh. Alternatively, each degree of freedom may be related to a local mesh entity, like a vertex, edge or face, on the reference element. After mapping the element, the degree of freedom will then be related to the corresponding mesh entity in the global mesh. This alternative requires that the corresponding mesh entities are numbered.

5 Finite element variational forms

By Robert C. Kirby and Anders Logg

Much of the FEniCS software is devoted to the formulation of variational forms (UFL), the discretization of variational forms (FIAT, FFC, SyFi) and the assembly of the corresponding discrete operators (UFC, DOLFIN). This chapter summarizes the notation for variational forms used throughout FEniCS.

5.1 Background

In Chapter 2, we introduced the following canonical variational problem: find $u \in V$ such that

$$a(u, v) = L(v) \quad \forall\, v \in \hat{V}, \tag{5.1}$$

where V is a given trial space and $\hat{V}$ is a given test space. The bilinear form

$$a : V \times \hat{V} \to \mathbb{R} \tag{5.2}$$

maps a pair of trial and test functions to a real number and is linear in both arguments. Similarly, the linear form $L : \hat{V} \to \mathbb{R}$ maps a given test function to a real number. We also considered the discretization of nonlinear variational problems: find $u \in V$ such that

$$F(u; v) = 0 \quad \forall\, v \in \hat{V}. \tag{5.3}$$

Here, $F : V \times \hat{V} \to \mathbb{R}$ again maps a pair of functions to a real number. The semilinear form F is nonlinear in the function u but linear in the test function v. Alternatively, we may consider the mapping

$$L_u \equiv F(u; \cdot) : \hat{V} \to \mathbb{R}, \tag{5.4}$$

and note that L_u is a linear form on $\hat{V}$ for any fixed value of u. In Chapter 2, we also considered the estimation of the error in a given functional $\mathcal{M} : V \to \mathbb{R}$. Here, the possibly nonlinear functional $\mathcal{M}$ maps a given function u to a real number $\mathcal{M}(u)$.

In all these examples, the central concept is that of a form that maps a given tuple of functions to a real number. We shall refer to these as *multilinear forms*. Below, we formalize the concept of a multilinear form, discuss the discretization of multilinear forms, and related concepts such as the *action*, *derivative* and *adjoint* of a multilinear form.

5.2 Multilinear forms

A form is a mapping from the product of a given sequence $\{V_j\}_{j=1}^{\rho}$ of function spaces to a real number:

$$a : V_\rho \times \cdots \times V_2 \times V_1 \to \mathbb{R}. \tag{5.5}$$

If the form a is linear in each of its arguments, we say that the form is *multilinear*. The number of arguments ρ of the form is the *arity* of the form. Note that the spaces are numbered from right to left. As we shall see below in Section 5.3, this is practical when we consider the discretization of multilinear forms.

Forms may often be parametrized over one or more *coefficients*. A typical example is the right-hand side L of the canonical variational problem (5.1), which is a linear form parametrized over a given coefficient f. We shall use the notation $a(f; v) \equiv L_f(v) \equiv L(v)$ and refer to the test function v as an *argument* and to the function f as a *coefficient*. In general, we shall refer to forms which are linear in each argument (but possibly nonlinear in its coefficients) as multilinear forms. Such a multilinear form is a mapping from the product of a sequence of argument spaces and coefficient spaces:

$$\begin{aligned} &a : W_1 \times W_2 \times \cdots \times W_n \times V_\rho \times \cdots \times V_2 \times V_1 \to \mathbb{R}, \\ &a \mapsto a(w_1, w_2, \ldots, w_n; v_\rho, \ldots, v_2, v_1). \end{aligned} \tag{5.6}$$

The argument spaces $\{V_j\}_{j=1}^{\rho}$ and coefficient spaces $\{W_j\}_{j=1}^{n}$ may all be the same space but they typically differ, such as when Dirichlet boundary conditions are imposed on one or more of the spaces, or when the multilinear form arises from the discretization of a mixed problem such as in Section 2.2.2.

In finite element applications, the arity of a form is typically $\rho = 2$, in which case the form is said to be bilinear, or $\rho = 1$, in which case the form is said to be linear. In the special case of $\rho = 0$, we shall refer to the multilinear form as a *functional*. It may sometimes also be of interest to consider forms of higher arity ($\rho > 2$). Below, we give examples of some multilinear forms of different arity.

5.2.1 Examples

Poisson's equation. Consider Poisson's equation with variable conductivity $\kappa = \kappa(x)$,

$$-\operatorname{div}(\kappa \operatorname{grad} u) = f. \tag{5.7}$$

Assuming Dirichlet boundary conditions on the boundary $\partial\Omega$, the corresponding canonical variational problem is defined in terms of a pair of multilinear forms, $a(\kappa; u, v) = \int_\Omega \kappa \operatorname{grad} u \cdot \operatorname{grad} v \,\mathrm{d}x$ and $L(v) = \int_\Omega f v \,\mathrm{d}x$. Here, a is a bilinear form ($\rho = 2$) and L is a linear form ($\rho = 1$). Both forms have one coefficient ($n = 1$) and the coefficients are κ and f respectively:

$$\begin{aligned} a &= a(\kappa; u, v), \\ L &= L(f; v). \end{aligned} \tag{5.8}$$

We usually drop the coefficients from the notation and use the short-hand notation $a = a(u, v)$ and $L = L(v)$.

The incompressible Navier–Stokes equations. The incompressible Navier–Stokes equations for the velocity u and pressure p of an incompressible fluid read:

$$\begin{aligned} \rho(\dot{u} + \operatorname{grad} u \cdot u) - \operatorname{div} \sigma(u, p) &= f, \\ \operatorname{div} u &= 0, \end{aligned} \tag{5.9}$$

where the stress tensor σ is given by $\sigma(u, p) = 2\mu\epsilon(u) - pI$, ϵ is the symmetric gradient; that is, $\epsilon(u) = \frac{1}{2}(\operatorname{grad} u + (\operatorname{grad} u)^\top)$, ρ is the fluid density and f is a body force.[1] Consider here the form obtained by integrating the nonlinear term $\operatorname{grad} u \cdot u$ against a test function v:

$$a(u; v) = \int_\Omega (\operatorname{grad} u \cdot u) \cdot v \, \mathrm{d}x. \tag{5.10}$$

This is a linear form ($\rho = 1$) with one coefficient ($n = 1$). We may linearize around a fixed velocity $\bar{u}$ to obtain

$$a(u; v) = a(\bar{u}; v) + a'(\bar{u}; v)\delta u + \mathcal{O}(\delta u^2), \tag{5.11}$$

where $u = \bar{u} + \delta u$. The linearized operator a' is here given by

$$a'(\bar{u}; \delta u, v) \equiv a'(v; \bar{u})\delta u = \int_\Omega (\operatorname{grad} \delta u \cdot \bar{u}) \cdot v + (\operatorname{grad} \bar{u} \cdot \delta u) \cdot v \, \mathrm{d}x. \tag{5.12}$$

This is a bilinear form ($\rho = 2$) with one coefficient ($n = 1$). We may also consider the *trilinear* form

$$a(w, u, v) = \int_\Omega (\operatorname{grad} u \cdot w) \cdot v \, \mathrm{d}x. \tag{5.13}$$

This trilinear form may be assembled into a rank three tensor and applied to a given vector of expansion coefficients for w to obtain a rank two tensor (a matrix) corresponding to the bilinear form $a(w; u, v)$. This may be useful in an iterative fixed point method for the solution of the Navier–Stokes equations, in which case w is a given (frozen) value for the convective velocity obtained from a previous iteration. This is rarely done in practice due to the cost of assembling the global rank three tensor. However, the corresponding local rank three tensor may be contracted with the local expansion coefficients for w on each local cell to compute the matrix corresponding to $a(w; u, v)$.

Lift and drag. When solving the Navier–Stokes equations, it may be of interest to compute the lift and drag of some object immersed in the fluid. The lift and drag are given by the z- and x-components of the force generated on the object (for a flow in the x-direction):

$$\begin{aligned} L_{\text{lift}}(u, p;) &= \int_\Gamma (\sigma(u, p) \cdot n) \cdot e_z \, \mathrm{d}s, \\ L_{\text{drag}}(u, p;) &= \int_\Gamma (\sigma(u, p) \cdot n) \cdot e_x \, \mathrm{d}s. \end{aligned} \tag{5.14}$$

Here, Γ is the boundary of the body, n is the outward unit normal of Γ and e_x, e_z are unit vectors in the x- and z-directions respectively. The arity of both forms is $\rho = 0$ and both forms have two coefficients.

5.2.2 Canonical form

FEniCS automatically handles the representation and evaluation of a large class of multilinear forms, but not all. FEniCS is currently limited to forms that may be expressed as a sum of integrals over the cells (the domain), the exterior facets (the boundary) and the interior facets of a given mesh. In

[1] For a discussion of the definition of the operators grad and ∇ for a vector-valued function, see Chapter 17. Note that for a vector-valued function u, we have $\operatorname{grad} u \cdot u = u \cdot \nabla u$.

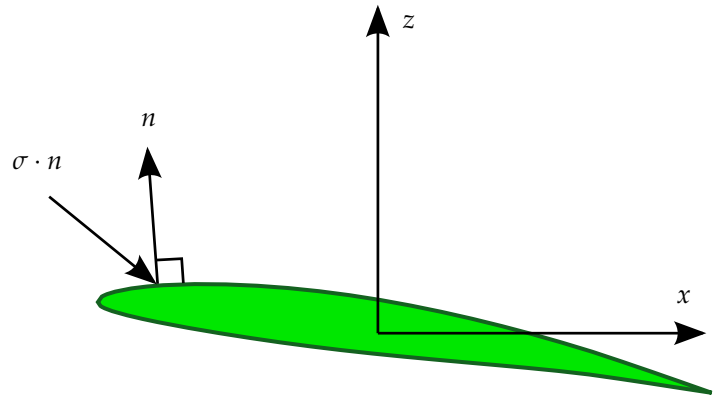


Figure 5.1: The lift and drag of an object, here a NACA 63A409 airfoil, are the integrals of the vertical and horizontal components respectively of the stress $\sigma \cdot n$ over the surface Γ of the object. At each point, the product of the stress tensor σ and the outward unit normal vector n gives the force per unit area acting on the surface.

particular, FEniCS handles forms that may be expressed as the following canonical form:

$$a(w_1, w_2, \ldots, w_n; v_\rho, \ldots, v_2, v_1) = \sum_{k=1}^{n_c} \int_{\Omega_k} I_k^c \,\mathrm{d}x + \sum_{k=1}^{n_f} \int_{\Gamma_k} I_k^f \,\mathrm{d}s + \sum_{k=1}^{n_f^0} \int_{\Gamma_k^0} I_k^{f,0} \,\mathrm{d}S. \tag{5.15}$$

Here, each Ω_k denotes a union of mesh cells covering a subset of the computational domain Ω. Similarly, each Γ_k denotes a subset of the facets on the boundary of the mesh, and Γ_k^0 denotes a subset of the interior facets of the mesh. The latter is of particular interest for the formulation of discontinuous Galerkin methods that typically involve integrals across cell boundaries (interior facets). The contribution from each subset is an integral over the subset of some integrand. Thus, the contribution from the kth subset of cells is an integral over Ω_k of the integrand I_k^c etc.

One may consider extensions of (5.15) that involve point values or integrals over subsets of individual cells (cut cells) or facets. Such extensions are currently not supported by FEniCS but may be added in the future.

5.3 Discretizing multilinear forms

As we saw in Chapter 2, one may obtain the finite element approximation $u_h = \sum_{j=1}^N U_j \phi_j \approx u$ of the canonical variational problem (5.1) by solving a linear system $AU = b$, where

$$\begin{aligned} A_{ij} &= a(\phi_j, \hat{\phi}_i), \quad i, j = 1, 2, \ldots, N, \\ b_i &= L(\hat{\phi}_i), \qquad i = 1, 2, \ldots, N. \end{aligned} \tag{5.16}$$

Here, A and b are the discrete operators corresponding to the bilinear and linear forms a and L for the given bases of the trial and test spaces. Note that the discrete operator is defined as the transpose of the multilinear form applied to the basis functions to account for the fact that in a bilinear form $a(u, v)$, the trial function u is associated with the columns of the matrix A, while the test function v is associated with the rows (the equations) of the matrix A.

In general, we may discretize a multilinear form a of arity ρ to obtain a tensor A of rank ρ. The discrete operator A is defined by

$$A_i = a(w_1, w_2, \ldots, w_n; \phi_{i_\rho}^\rho, \ldots, \phi_{i_2}^2, \phi_{i_1}^1), \tag{5.17}$$

where $i = (i_1, i_2, \ldots, i_\rho)$ is a multi-index of length ρ and $\{\phi_k^j\}_{k=1}^{N_j}$ is a basis for $V_{j,h} \subset V_j$, $j = 1, 2, \ldots, \rho$. The discrete operator is a typically sparse tensor of rank ρ and dimension $N_1 \times N_2 \times \cdots \times N_\rho$.

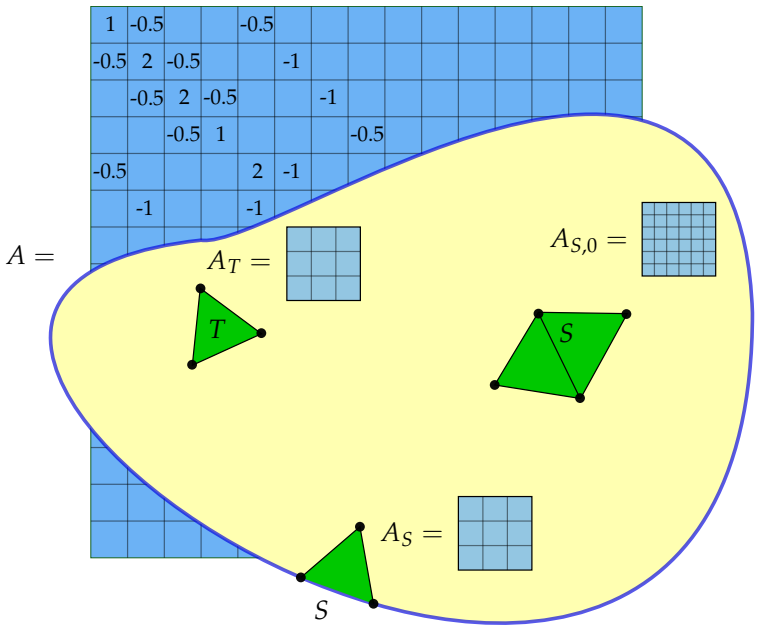


Figure 5.2: The cell tensor A_T, exterior facet tensor A_S, and interior facet tensor $A_{S,0}$ on a mesh are obtained by discretizing the local contribution to a multilinear form on a cell, exterior facet or interior facet, respectively. By assembling the local contributions from all cell and facet tensors, one obtains the global discrete operator A that discretizes the multilinear form.

The discrete operator A may be computed efficiently using an algorithm known as *assembly*, which is the topic of the next chapter. As we shall see then, an important tool is the *cell tensor* obtained as the discretization of the bilinear form on a local cell of the mesh. In particular, consider the discretization of a multilinear form that may be expressed as a sum of local contributions from each cell T of a mesh $\mathcal{T}_h = \{T\}$,

$$a(w_1, w_2, \ldots, w_n; v_\rho, \ldots, v_2, v_1) = \sum_{T \in \mathcal{T}_h} a_T(w_1, w_2, \ldots, w_n; v_\rho, \ldots, v_2, v_1). \tag{5.18}$$

Discretizing a_T using the local finite element basis $\{\phi_k^{T,j}\}_{k=1}^{n_j}$ on T for $j = 1, 2, \ldots, \rho$, we obtain the cell tensor

$$A_{T,i} = a_T(w_1, w_2, \ldots, w_n; \phi_{i_\rho}^{T,\rho}, \ldots, \phi_{i_2}^{T,2}, \phi_{i_1}^{T,1}). \tag{5.19}$$

The cell tensor A_T is a typically dense tensor of rank ρ and dimension $n_1 \times n_2 \times \cdots \times n_\rho$. The discrete operator A may be obtained by appropriately summing the contributions from each cell tensor A_T. We return to this in detail below in Chapter 6.

One may similarly define the exterior and interior facet tensors A_S and $A_{S,0}$ as the contributions from a facet on the boundary or in the interior of the mesh. The exterior facet tensor A_S is defined as in (5.19) by replacing the domain of integration T by a facet S. The dimension of A_S is generally the same as that of A_T. The interior facet tensor $A_{S,0}$ is defined slightly differently by considering the basis on a *macro element* consisting of the two elements sharing the common facet S as depicted in Figure 5.2. For details, we refer to Ølgaard et al. (2008).

5.4 *The action of a multilinear form*

Consider the bilinear form

$$a(u, v) = \int_\Omega \operatorname{grad} u \cdot \operatorname{grad} v \, \mathrm{d}x, \tag{5.20}$$

obtained from the discretization of the left-hand side of Poisson's equation. Here, u and v are a pair of trial and test functions. Alternatively, we may consider v to be a test function and u to be a given

solution to obtain a *linear* form parametrized over the coefficient u:

$$(\mathcal{A}a)(u;v) = \int_\Omega \operatorname{grad} u \cdot \operatorname{grad} v \,\mathrm{d}x. \tag{5.21}$$

We refer to the linear form $\mathcal{A}a$ as the *action* of the bilinear form a. In general, the action of a ρ-linear form with n coefficients is a $(\rho-1)$-linear form with $n+1$ coefficients. In particular, the action of a bilinear form is a linear form, and the action of a linear form is a functional.

The action of a bilinear form plays an important role in the definition of matrix-free methods for solving differential equations. Consider the solution of a variational problem of the canonical form (5.1) by a Krylov subspace method such as GMRES (Generalized Minimal RESidual method) (Saad and Schultz, 1986) or CG (Conjugate Gradient method) (Hestenes and Stiefel, 1952). Krylov methods approximate the solution $U \in \mathbb{R}^N$ of the linear system $AU = b$ by finding an approximation for U in the subspace of $\mathbb{R}^N$ spanned by the vectors $b, Ab, A^2b, \ldots, A^kb$ for some $k \ll N$. These vectors may be computed by repeated application of the discrete operator A defined as above by

$$A_{ij} = a(\phi_j^2, \phi_i^1). \tag{5.22}$$

For any given vector $U \in \mathbb{R}^N$, it follows that

$$(AU)_i = \sum_{j=1}^N A_{ij}U_j = \sum_{j=1}^N a(\phi_j^2, \phi_i^1)U_j = a\left(\sum_{j=1}^N U_j\phi_j^2, \phi_i^1\right) = a(u_h, \phi_i^1) = (\mathcal{A}a)(u_h;\phi_i^1), \tag{5.23}$$

where $u_h = \sum_{j=1}^N U_j\phi_j^2$ is the finite element approximation corresponding to the coefficient vector U. In other words, the application of the matrix A on a given vector U is given by the action of the bilinear form evaluated at the corresponding finite element approximation:

$$(AU)_i = (\mathcal{A}a)(u_h;\phi_i^1). \tag{5.24}$$

The variational problem (5.1) may thus be solved by repeated evaluation (assembly) of a linear form (the action $\mathcal{A}a$ of the bilinear form a) as an alternative to first computing (assembling) the matrix A and then repeatedly computing matrix–vector products with A. Which approach is more efficient depends on how efficiently the action may be computed compared to matrix assembly, as well as on available preconditioners. For a further discussion on the action of multilinear forms, we refer to Bagheri and Scott (2004).

Computing the action of a multilinear form is supported by the UFL form language by calling the `action` function:

Python code

```
a  = inner(grad(u), grad(v))*dx
Aa = action(a)
```

5.5 The derivative of a multilinear form

When discretizing nonlinear variational problems, it may be of interest to compute the derivative of a multilinear form with respect to one or more of its coefficients. Consider the nonlinear variational problem to find $u \in V$ such that

$$a(u;v) = 0 \quad \forall\, v \in \hat{V}. \tag{5.25}$$

To solve this problem by Newton's method, we linearize around a fixed value $\bar{u}$ to obtain

$$0 = a(u; v) \approx a(\bar{u}; v) + a'(\bar{u}; v)\delta u. \tag{5.26}$$

Given an approximate solution $\bar{u}$ of the nonlinear variational problem (5.25), we may then hope to improve the approximation by solving the following linear variational problem: find $\delta u \in V$ such that

$$a'(\bar{u}; \delta u, v) \equiv a'(\bar{u}; v)\delta u = -a(\bar{u}; v) \quad \forall\, v \in \hat{V}. \tag{5.27}$$

Here, a' is a bilinear form with two arguments δu and v, and one coefficient $\bar{u}$. Furthermore, $-a$ is a linear form with one argument v and one coefficient $\bar{u}$.

When there is more than one coefficient, we use the notation $\mathcal{D}_w$ to denote the derivative with respect to a specific coefficient w. In general, the derivative $\mathcal{D}$ of a ρ-linear form with $n > 0$ coefficients is a $(\rho + 1)$-linear form with n coefficients. To solve the variational problem (5.25) using a matrix-free Newton method, we would thus need to repeatedly evaluate the linear form $(\mathcal{A}\mathcal{D}_u a)(\bar{u}_h, \delta u_h; v)$ for a given finite element approximation $\bar{u}_h$ and increment δu_h.

Note that one may equivalently consider the application of Newton's method to the nonlinear discrete system of equations obtained by a direct application of the finite element method to the variational problem (5.25) as discussed in Chapter 2.

Computing the derivative of a multilinear form is supported by the UFL form language by calling the `derivative` function:

Python code

```
a  = inner((1 + u)*grad(u), grad(v))*dx
Da = derivative(a, u)
```

5.6 The adjoint of a bilinear form

The adjoint a^* of a bilinear form a is the form obtained by interchanging the two arguments:

$$a^*(v, w) = a(w, v) \quad \forall\, v \in V^1 \quad \forall\, w \in V^2. \tag{5.28}$$

The adjoint of a bilinear form plays an important role in the error analysis of finite element methods as we saw in Chapter 2 and as will be discussed further in Chapter 29 where we consider the linearized adjoint problem (the dual problem) of the general nonlinear variational problem (5.25). The dual problem takes the form

$$(\mathcal{D}_u a)^*(u; z, v) = \mathcal{D}_u \mathcal{M}(u; v) \quad \forall\, v \in V, \tag{5.29}$$

or simply

$$a'^*(z, v) = \mathcal{M}'(v) \quad \forall\, v \in V, \tag{5.30}$$

where $(\mathcal{D}_u a)^*$ is a bilinear form, $\mathcal{D}_u \mathcal{M}$ is a linear form (the derivative of the functional $\mathcal{M}$), and z is the solution of the dual problem.

Computing the adjoint of a multilinear form is supported by the UFL form language by calling the `adjoint` function:

Python code

```
a = div(u)*q*dx
a_star = adjoint(a)
```

5.7 A note on the order of trial and test functions

It is common in the literature to consider bilinear forms where the trial function u is the first argument, and the test function v is the second argument:

$$a = a(u, v). \tag{5.31}$$

With this notation, one is lead to define the discrete operator A as

$$A_{ij} = a(\phi_j, \phi_i), \tag{5.32}$$

that is, a transpose must be introduced to account for the fact that the order of trial and test functions does not match the order of rows and columns in a matrix. Alternatively, one may change the order of trial and test functions and write $a = a(v, u)$ and avoid taking the transpose in the definition of the discrete operator $A_{ij} = a(\phi_i, \phi_j)$. This is practical in the definition and implementation of software systems such as FEniCS for the general treatment of variational forms.

In this book and throughout the code and documentation of the FEniCS Project, we have adopted the following compromise. Variational forms are expressed using the conventional *order* of trial and test functions; that is,

$$a = a(u, v), \tag{5.33}$$

but using an unconventional *numbering* of trial and test functions. Thus, v is the first argument of the bilinear form and u is the second argument. This ensures that one may express finite element variational problems in the conventional notation, but at the same time allows the implementation to use a more practical numbering scheme.

6 *Finite element assembly*

By Anders Logg, Kent-Andre Mardal and Garth N. Wells

The finite element method may be viewed as a method for forming a discrete linear system $AU = b$ or nonlinear system $b(U) = 0$ corresponding to the discretization of the variational form of a differential equation. A central part of the implementation of finite element methods is therefore the computation of matrices and vectors from variational forms. In this chapter, we describe the standard algorithm for computing the discrete operator (tensor) A. This algorithm is known as *finite element assembly*. We also discuss efficiency aspects of the standard algorithm and extensions to matrix-free methods.

6.1 *Assembly algorithm*

As seen in Chapter 5, the discrete operator of a multilinear form $a : V_\rho \times \cdots \times V_2 \times V_1 \to \mathbb{R}$ of arity ρ is the rank ρ tensor A defined by

$$A_I = a(\phi^\rho_{I_\rho}, \ldots, \phi^2_{I_2}, \phi^1_{I_1}), \tag{6.1}$$

where $I = (I_1, I_2, \ldots, I_\rho)$ is a multi-index of length ρ and $\{\phi^j_k\}_{k=1}^{N_j}$ is a basis for $V_{j,h} \subset V_j$, $j = 1, 2, \ldots, \rho$. The discrete operator is a typically sparse tensor of rank ρ and dimension $N_1 \times N_2 \times \cdots \times N_\rho$.

A straightforward algorithm to compute the tensor A is to iterate over all its entries and compute them one by one as outlined in Algorithm 1. This algorithm has two major drawbacks and is rarely used in practice. First, it does not take into account that most entries of the sparse tensor A may be zero. Second, it does not take into account that each entry is typically a sum of contributions (integrals) from the set of cells that form the support of the basis functions $\phi^1_{I_1}, \phi^2_{I_2}, \ldots, \phi^\rho_{I_\rho}$. As a result, each cell of the mesh must be visited multiple times when computing the local contribution to different entries of the tensor A. For this reason, the tensor A is usually computed by iterating over the cells of the mesh and adding the contribution from each local cell to the global tensor A. To see how the tensor A can be decomposed as a sum of local contributions, we recall the definition of the cell tensor A_T from Chapter 5:

$$A_{T,i} = a_T(\phi^{T,\rho}_{i_\rho}, \ldots, \phi^{T,2}_{i_2}, \phi^{T,1}_{i_1}), \tag{6.2}$$

where $i = (i_1, i_2, \ldots, i_\rho)$ is a multi-index of length ρ, $A_{T,i}$ is the ith entry of the rank ρ tensor A_T, a_T is the local contribution to the multilinear form from a cell $T \in \mathcal{T}_h$ and $\{\phi^{T,j}_k\}_{k=1}^{n_j}$ is the local finite element basis for $V_{j,h}$ on T. We assume here that the multilinear form is expressed as an integral over the domain Ω so that it may be naturally decomposed as a sum of local contributions. If the form contains contributions from facet or boundary integrals, one may similarly decompose the multilinear form into local contributions from facets.

Algorithm 1 Straightforward (naive) "assembly" algorithm.

for $I_1 = 1, 2, \ldots, N_1$
  for $I_2 = 1, 2, \ldots, N_2$
    for ...
      $A_I = a(\phi^\rho_{I_\rho}, \ldots, \phi^2_{I_2}, \phi^1_{I_1})$

To formulate the general assembly algorithm, let $\iota^j_T : [1, n_j] \to [1, N_j]$ denote the local-to-global mapping introduced in Chapter 2 for each discrete function space $V_{j,h}$, $j = 1, 2, \ldots, \rho$, and define for each $T \in \mathcal{T}_h$ the collective local-to-global mapping $\iota_T : \mathcal{I}_T \to \mathcal{I}$ by

$$\iota_T(i) = (\iota^1_T(i_1), \iota^2_T(i_2), \ldots, \iota^\rho_T(i_\rho)) \quad \forall\, i \in \mathcal{I}_T, \tag{6.3}$$

where $\mathcal{I}_T$ is the index set

$$\mathcal{I}_T = \prod_{j=1}^{\rho} [1, n_j] = \{(1, 1, \ldots, 1), (1, 1, \ldots, 2), \ldots, (n_1, n_2, \ldots, n_\rho - 1), (n_1, n_2, \ldots, n_\rho)\}. \tag{6.4}$$

That is, ι_T maps a tuple of local degrees of freedom to a tuple of global degrees of freedom. Furthermore, let $\mathcal{T}_I \subset \mathcal{T}_h$ denote the subset of cells of the mesh on which $\{\phi^j_{I_j}\}_{j=1}^\rho$ are all nonzero. We note that ι_T is invertible if $T \in \mathcal{T}_I$. We may now compute the tensor A by summing local contributions from the cells of the mesh:

$$\begin{aligned} A_I &= \sum_{T \in \mathcal{T}_h} a_T(\phi^\rho_{I_\rho}, \ldots, \phi^2_{I_2}, \phi^1_{I_1}) = \sum_{T \in \mathcal{T}_I} a_T(\phi^\rho_{I_\rho}, \ldots, \phi^2_{I_2}, \phi^1_{I_1}) \\ &= \sum_{T \in \mathcal{T}_I} a_T(\phi^{T,\rho}_{(\iota^\rho_T)^{-1}(I_\rho)}, \ldots, \phi^{T,2}_{(\iota^2_T)^{-1}(I_2)}, \phi^{T,1}_{(\iota^1_T)^{-1}(I_1)}) = \sum_{T \in \mathcal{T}_I} A_{T, \iota_T^{-1}(I)}. \end{aligned} \tag{6.5}$$

This computation may be carried out efficiently by a single iteration over all cells $T \in \mathcal{T}_h$. On each cell T, the cell tensor A_T is computed and then added to the global tensor A as outlined in Algorithm 2 and illustrated in Figure 6.1.

Algorithm 2 Assembly algorithm.

$A = 0$
for $T \in \mathcal{T}_h$
  (1) Compute ι_T
  (2) Compute A_T

  (3) Add A_T to A according to ι_T:
  for $i \in \mathcal{I}_T$
    $A_{\iota_T(i)} \stackrel{+}{=} A_{T,i}$
  end for
end for

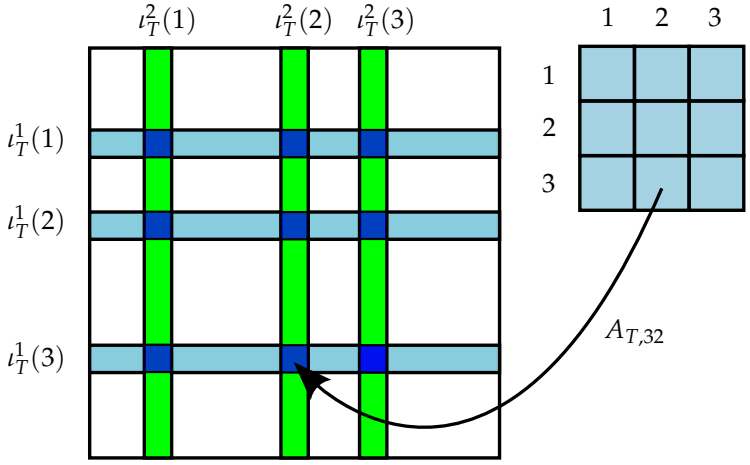


Figure 6.1: Adding the entries of the cell tensor A_T to the global tensor A using the local-to-global mapping ι_T, illustrated here for the assembly of a rank two tensor (matrix) with piecewise linear elements on triangles. On each element T, a 3×3 element matrix A_T is computed and its entries are added to the global matrix. The entries of the first row are added to row $\iota_T^1(1)$ of the global matrix in the columns given by $\iota_T^2(1)$, $\iota_T^2(2)$ and $\iota_T^2(3)$, respectively. The entries of the second row are added to row $\iota_T^1(2)$ of the global matrix etc.

Figure 6.2: Actual implementation (excerpt) of the assembly algorithm (Algorithm 2) in DOLFIN (from `Assembler.cpp` in DOLFIN 1.0).

C++ code

```
for (CellIterator cell(mesh); !cell.end(); ++cell)
{
  ...

  // Get local-to-global dofmap for each dimension
  for (uint i = 0; i < form_rank; ++i)
    dofs[i] = &(dofmaps[i]->cell_dofs(cell->index()));

  // Tabulate cell tensor
  integral->tabulate_tensor(&ufc.A[0],
                            ufc.w(),
                            ufc.cell);

  // Add entries to global tensor
  A.add(&ufc.A[0], dofs);
}
```

6.2 Implementation

In FEniCS, the assembly algorithm (Algorithm 2) is implemented as part of DOLFIN (see Figure 6.2). For the steps (1), (2) and (3) of the assembly algorithm, DOLFIN relies on external code. For steps (1) and (2), DOLFIN calls code generated by a form compiler such as FFC or SyFi. In particular, DOLFIN calls the two functions `tabulate_dofs` and `tabulate_tensor` through the UFC interface for steps (1) and (2), respectively. Step (3) is carried out through the DOLFIN `GenericTensor::add` interface and maps to the corresponding operation in one of a number of linear algebra backends, such as `MatSetValues` for PETSc and `SumIntoGlobalValues` for Trilinos/Epetra.

In typical assembly implementations, the computation of the cell tensor A_T is the most costly operation of the assembly algorithm. For DOLFIN, however, as a result of optimized algorithms for the computation of A_T being generated by form compilers (see Chapters 7 and 8), adding entries of the local tensor A_T to appropriate positions in the global tensor A often constitutes a significant portion of the total assembly time. This operation is costly since the addition of a value to an arbitrary entry of a sparse tensor is not a trivial operation, even when the layout of the sparse matrix has been initialized. In the standard case when A is a sparse matrix (a rank two tensor), the linear algebra backend stores the sparse matrix in *compressed row storage* (CRS) format or some other sparse format. For each given entry, the linear algebra backend must search along a row I to find the position to store the value for a given column J. As a result, the speed of assembly in FEniCS for sparse matrices is currently

limited by the speed of insertion into a sparse linear algebra data structure for many problems. An additional cost is associated with the initialization of a sparse matrix, which involves the computation of a sparsity pattern. For most linear algebra libraries, it is necessary to initialize the layout of a sparse matrix before inserting entries in order to achieve tolerable insertion speed. Computation of the sparsity pattern is a moderately costly operation, but which in the case of nonlinear problems is usually amortized over time.

Algorithm 2 may be easily extended to assembly over the facets of a mesh. Assembly over facets is necessary both for handling variational forms that contain integrals over the boundary of a mesh (the exterior facets), to account for Neumann boundary conditions, and forms that contain integrals over the interior facets of a mesh as part of a discontinuous Galerkin formulation. For this reason, DOLFIN implements three different assembly algorithms. These are assembly over cells, exterior facets and interior facets.

6.3 Symmetric application of boundary conditions

For symmetric problems, it is useful to be able to apply Dirichlet boundary conditions in a fashion that preserves the symmetry of the matrix, since that allows the use of solution algorithms which are limited to symmetric matrices, such as the conjugate gradient method and Cholesky decomposition. The symmetric application of boundary conditions may be handled by modifying the cell tensors A_T before assembling into the global tensor A. Assembly with symmetric application of boundary conditions is implemented in DOLFIN in the class `SystemAssembler`.

To explain the symmetric assembly algorithm, consider the global system $AU = b$ and the corresponding element matrix A_T and element vector b_T. If a global index I is associated with a Dirichlet boundary condition, $U_I = D_I$, then this condition can be enforced by setting $A_{II} = 1$, $A_{IJ} = 0$ for $I \neq J$, and $b_I = D_I$. This approach is applied when calling the DOLFIN function `DirichetBC::apply`. However, to preserve symmetry of the matrix, we can perform a partial Gaussian elimination to obtain $A_{JI} = A_{IJ} = 0$ for $I \neq J$. This is achieved by subtracting the Ith row multiplied by A_{JI} from the Jth equation, locally. These partial Gaussian eliminations are performed on the linear systems at the element level. The local linear systems are then added to the global matrix. As a result, the Dirichlet condition is added multiple times to the global vector, one time for each cell, which is compensated for by the addition of one multiple times to the corresponding diagonal entry of A. This is summarized in Algorithm 3. Alternatively, one may choose to eliminate degrees of freedom corresponding to Dirichlet boundary conditions from the linear system (since these values are known). The values then end up in the right-hand side of the linear system. The described algorithm does not eliminate the degrees of freedom associated with a Dirichlet boundary condition. Instead, these degrees of freedom are retained to preserve the dimension of the linear system so that it always matches the total number of degrees of freedom for the solution (including known Dirichlet values).

6.4 Parallel assembly

The assembly algorithms remain unchanged in a distributed[1] parallel environment if the linear algebra backend supports distributed matrices and insertion for both on- and off-process matrix entries, and if the mesh data structure supports distributed meshes. Both PETSc (Balay et al., 2001, 2004) and Trilinos/Epetra (Heroux et al., 2005) support distributed matrices and vectors. Efficient parallel assembly relies on appropriately partitioned meshes and properly distributed degree-of-freedom maps to minimize inter-process communication. It is not generally possible to produce an effective

[1]By distributed assembly, we refer here to assembly in parallel on a distributed memory parallel architecture, running multiple processes that cannot access the same memory, but must pass data as messages between processes.

Algorithm 3 Symmetric assembly algorithm ($\rho = 2$).

$A = 0$ and $b = 0$
for $T \in \mathcal{T}_h$
  (1) Compute ι_T^A and ι_T^b
  (2) Compute A_T and b_T
  (3) Apply Dirichlet boundary conditions to A_T and b_T
  (4) Perform partial Gaussian elimination on A_T and b_T to preserve symmetry
  (5) Add A_T and b_T to A and b according to ι_T^A and ι_T^b, respectively:
  for $(i, j) \in \mathcal{I}_T^A$
    $A_{\iota_T^{A,1}(i),\iota_T^{A,2}(j)} \stackrel{+}{=} A_{T,ij}$
  end for
  for $i \in \mathcal{I}_T^b$
    $b_{\iota_T^b(i)} \stackrel{+}{=} b_i^\top$
  end for
end for

degree-of-freedom map using only a form compiler, since the degree-of-freedom map should reflect the partitioning of the mesh. Instead, one may use a degree-of-freedom map generated by a form compiler to construct a suitable map at run-time. DOLFIN supports distributed meshes and computes distributed degree of freedom maps for distributed assembly.

Multi-threaded[2] assembly is outwardly simpler than distributed assembly and is attractive given the rapid growth in multi-core architectures. The assembly code can be easily modified, using for example OpenMP, to parallelize the assembly loop over cells. Multi-threaded assembly requires extra care so that multiple threads don't write to the same memory location (when multiple threads attempt to write to the same memory location, this is known as a *race condition*). Multi-threaded assembly has recently been implemented in DOLFIN (from version 1.0) based on coloring the cells of the mesh so that no two neighboring cells (cells that share a common vertex in the case of Lagrange elements) have the same color. One may then iterate over the colors of the mesh, and for each color use OpenMP to parallelize the assembly loop. This ensures that no two cells will read data from the same location (in the mesh), or write data to the same location (in the global tensor).

6.5 Matrix-free methods

A feature of Krylov subspace methods and some other iterative methods for linear systems of the form $AU = b$ is that they rely only on the *action* of the matrix operator A on vectors and do not require direct manipulation of A. This is in contrast with direct linear solvers. Therefore, if the action of A on an arbitrary vector v can be computed, then a Krylov solver can be used to solve the system $AU = b$ without needing to assemble A. This matrix-free approach may be attractive for problem types that are well-suited to Krylov solvers and for which the assembly of A is costly (in terms of CPU time and/or memory). A disadvantage of matrix-free methods is that the preconditioners that are most commonly used to improve the convergence properties and robustness of Krylov solvers do involve manipulations of A; hence these cannot be applied in a matrix-free approach. For the purpose of assembly, a matrix-free approach replaces the assembly of the matrix A with repeated assembly of a vector Av, which is the action of A on the given vector v. A key element in the efficient application

[2]By multi-threaded assembly, we refer here to assembly in parallel on a shared memory parallel architecture, running multiple threads that may access the same memory.

of such methods is the rapid assembly of vectors. The cost of insertion into a dense vector is relatively low, compared to insertion into a sparse matrix. The computation of the cell tensor is therefore the dominant cost. Assembly of the action of a linear or linearized operator is supported in FEniCS.

7 *Quadrature representation of finite element variational forms*

By Kristian B. Ølgaard and Garth N. Wells

This chapter addresses the conventional run-time quadrature approach for the numerical integration of local element tensors associated with finite element variational forms, and in particular automated optimizations that can be performed to reduce the number of floating point operations. An alternative to the run-time quadrature approach is the tensor representation presented in Chapter 8. Both the quadrature and tensor approaches are implemented in FFC (see Chapter 11). In this chapter we discuss four strategies for optimizing the quadrature representation for run-time performance of the generated code and show that optimization strategies lead to a dramatic improvement in run-time performance over a naive implementation. We also examine performance aspects of the quadrature and tensor approaches for different equations, and this will motivate the desirability of being able to choose between the two representations.

7.1 Standard quadrature representation

To illustrate the standard quadrature representation and optimizations implemented in FFC we consider the bilinear form for the weighted Laplace operator $-\nabla \cdot (w\nabla u)$, where u is the unknown and w is a prescribed coefficient. The bilinear form of the variational problem for this equation reads

$$a(u, v) = \int_\Omega w\nabla u \cdot \nabla v \, \mathrm{d}x. \tag{7.1}$$

The quadrature approach can deal with cases in which not all functions come from a finite element space, using 'quadrature functions' that can be evaluated directly at quadrature points. The tensor representation approach only supports cases in which all functions come from a finite element space (using interpolation if necessary). Therefore, to ensure a proper performance comparison between the representations we assume that all functions in a form, including coefficient functions, come from a finite element function space. In the case of (7.1), all functions will come from

$$V_h = \left\{ v \in H^1(\Omega) : \; v|_T \in P_q(T) \, \forall \, T \in \mathcal{T} \right\}, \tag{7.2}$$

where $P_q(T)$ denotes the space of Lagrange polynomials of degree q on the element T of the standard triangulation of Ω, which is denoted by $\mathcal{T}$. If we let $\{\phi_i^T\}$ denote the local finite element basis that span the discrete function space V_h on T, the local element tensor for an element T can be computed as

$$A_{T,i} = \int_T w\nabla\phi_{i_1}^T \cdot \nabla\phi_{i_2}^T \, \mathrm{d}x, \tag{7.3}$$

where $i = (i_1, i_2)$.

The expression for the local element tensor in (7.3) can be expressed in UFL (see Chapter 17), from which FFC generates an intermediate representation of the form (see Chapter 11). Assuming a standard affine mapping $F_T : T_0 \to T$ from a reference element T_0 to a given element $T \in \mathcal{T}$, this intermediate representation reads

$$A_{T,i} = \sum_{q=1}^{N} \sum_{\alpha_3=1}^{n} \Phi_{\alpha_3}(X^q) w_{\alpha_3} \sum_{\beta=1}^{d} \sum_{\alpha_1=1}^{d} \frac{\partial X_{\alpha_1}}{\partial x_\beta} \frac{\partial \Phi_{i_1}(X^q)}{\partial X_{\alpha_1}} \sum_{\alpha_2=1}^{d} \frac{\partial X_{\alpha_2}}{\partial x_\beta} \frac{\partial \Phi_{i_2}(X^q)}{\partial X_{\alpha_2}} \det F_T' W^q, \tag{7.4}$$

where a change of variables from the reference coordinates X to the real coordinates $x = F_T(X)$ has been used. In the above equation, N denotes the number of integration points, d is the dimension of Ω, n is the number of degrees of freedom for the local basis of w, Φ_i denotes basis functions on the reference element, $\det F_T'$ is the determinant of the Jacobian, and W^q is the quadrature weight at integration point X^q. By default, FFC applies a quadrature scheme that will integrate the variational form exactly.

From the representation in (7.4), code for computing entries of the local element tensor is generated by FFC. This code is shown in Figure 7.1. Code generated for the quadrature representation is structured in the following way. First, values of geometric quantities that depend on the current element T, like the components of the inverse of the Jacobian matrix $\partial X_{\alpha_1}/\partial x_\beta$ and $\partial X_{\alpha_2}/\partial x_\beta$, are computed and assigned to the variables like `K_01` in the code (this code is not shown as it is not important for understanding the nature of the quadrature representation). Next, values of basis functions and their derivatives at integration points on the reference element, like $\Phi_{\alpha_3}(X^q)$ and $\partial \Phi_{i_1}(X^q)/\partial X_{\alpha_1}$ are tabulated. Finite element basis functions are computed by FIAT. Basis functions and their derivatives on a reference element are independent of the current element T and are therefore tabulated at compile-time and stored in the tables `Psi_w`, `Psi_vu_D01` and `Psi_vu_D10` in Figure 7.1. After the tabulation of basis functions values, the loop over integration points begins. In the example we are considering linear elements, and only one integration point is necessary for exact integration. The loop over integration points has therefore been omitted. The first task inside a loop over integration points is to compute the values of coefficients at the current integration point. For the considered problem, this involves computing the value of the coefficient w. The code for evaluating `F0` in Figure 7.1 is an exact translation of the representation $\sum_{\alpha_3=1}^{n} \Phi_{\alpha_3}(X^q) w_{\alpha_3}$. The last part of the code in Figure 7.1 is the loop over the basis function indices i_1 and i_2, where the contribution to each entry in the local element tensor, A_T, from the current integration point is added. To generate code using the quadrature representation the FFC command-line option `-r quadrature` should be used.

7.2 Quadrature optimizations

We now address optimizations for improving the run-time performance of the generated code. The underlying philosophy of the optimization strategies implemented in FFC is to manipulate the representation in such a way that the number of operations to compute the local element tensor decreases. Each strategy described in the following sections, with the exception of eliminating operations on zero terms, share some common features which can be categorized as:

Loop invariant code motion In short, this procedure seeks to identify terms that are independent of one or more of the summation indices and to move them outside the loop over those particular indices. For instance, in (7.4) the terms regarding the coefficient w, the quadrature weight W^q and the determinant $\det F_T'$ are all independent of the basis function indices i_1 and i_2 and therefore only need to be computed once for each integration point. A generic discussion of this technique, which is also known as 'loop hoisting', can be found in Alfred et al. (1986).

Figure 7.1: Part of the generated code for the bilinear form associated with the weighted Laplacian using linear elements in two dimensions. The variables like `K_00` are components of the inverse of the Jacobian matrix and `det` is the determinant of the Jacobian. The code to compute these variables is not shown. `A` holds the values of the local element tensor and `w` contains nodal values of the weighting function w.

C++ code

```
virtual void tabulate_tensor(double* A,
                             const double * const * w,
                             const ufc::cell& c) const
{
  ...
  // Quadrature weight.
  static const double W1 = 0.5;

  // Tabulated basis functions at quadrature points.
  static const double Psi_w[1][3] = \
  {{0.33333333333333, 0.33333333333333, 0.33333333333333}};
  static const double Psi_vu_D01[1][3] = \
  {{-1.0, 0.0, 1.0}};
  static const double Psi_vu_D10[1][3] = \
  {{-1.0, 1.0, 0.0}};

  // Compute coefficient value.
  double F0 = 0.0;
  for (unsigned int r = 0; r < 3; r++)
    F0 += Psi_w[0][r]*w[0][r];

  // Loop basis functions.
  for (unsigned int j = 0; j < 3; j++)
  {
    for (unsigned int k = 0; k < 3; k++)
    {
      A[j*3 + k] +=\
      ((K_00*Psi_vu_D10[0][j] + K_10*Psi_vu_D01[0][j])*\
       (K_00*Psi_vu_D10[0][k] + K_10*Psi_vu_D01[0][k]) +\
       (K_01*Psi_vu_D10[0][j] + K_11*Psi_vu_D01[0][j])*\
       (K_01*Psi_vu_D10[0][k] + K_11*Psi_vu_D01[0][k])\
      )*F0*W1*det;
    }
  }
}
```

C++ code

```
// Tabulated basis functions.
static const double Psi_vu[1][2] = {{-1.0, 1.0}};

// Arrays of nonzero columns.
static const unsigned int nzc0[2] = {0, 2};
static const unsigned int nzc1[2] = {0, 1};

// Loop basis functions.
for (unsigned int j = 0; j < 2; j++)
{
 for (unsigned int k = 0; k < 2; k++)
 {
  A[nzc0[j]*3 + nzc0[k]] +=\
   (K_10*Psi_vu[0][j]*K_10*Psi_vu[0][k] +\
    K_11*Psi_vu[0][j]*K_11*Psi_vu[0][k])*F0*W1*det;
  A[nzc0[j]*3 + nzc1[k]] +=\
   (K_11*Psi_vu[0][j]*K_01*Psi_vu[0][k] +\
    K_10*Psi_vu[0][j]*K_00*Psi_vu[0][k])*F0*W1*det;
  A[nzc1[j]*3 + nzc0[k]] +=\
   (K_00*Psi_vu[0][j]*K_10*Psi_vu[0][k] +\
    K_01*Psi_vu[0][j]*K_11*Psi_vu[0][k])*F0*W1*det;
  A[nzc1[j]*3 + nzc1[k]] +=\
   (K_01*Psi_vu[0][j]*K_01*Psi_vu[0][k] +\
    K_00*Psi_vu[0][j]*K_00*Psi_vu[0][k])*F0*W1*det;
 }
}
```

Figure 7.2: Part of the generated code for the weighted Laplacian using linear elements in two dimensions with optimization option `-f eliminate_zeros`. The arrays `nzc0` and `nzc1` contain the nonzero column indices for the mapping of values. Note how eliminating zeros makes it possible to replace the two tables with derivatives of basis functions `Psi_vu_D01` and `Psi_vu_D10` from Figure 7.1 with one table (`Psi_vu`).

Reuse common terms Terms that appear multiple times in an expression can be identified, computed once, stored as temporary values and then reused in all occurrences in the expression. This can have a great impact on the operation count since the expression to compute an entry in A_T is located inside loops over the basis function indices as shown in the code for the standard quadrature representation in Figure 7.1.

To switch on optimization the command-line option `-O` should be used in addition to any of the FFC optimization options presented in the following sections.

7.2.1 *Eliminate operations on zeros*

Some basis functions and derivatives of basis functions may be zero-valued at all integration points for a particular problem. Since these values are tabulated at compile-time, the columns containing nonzero values can be identified. This enables a reduction in the loop dimension for indices concerning these tables. However, a consequence of reducing the tables is that a mapping of indices must be created in order to access values correctly. The mapping results in memory not being accessed contiguously at run-time and can lead to a decrease in run-time performance.

This optimization is switched on by using the command-line option `-f eliminate_zeros`. Code for the weighted Laplace equation generated with this option is shown in Figure 7.2. For brevity, only code different from that in Figure 7.1 has been included.

Although the elimination of zeros has lead to a decrease of the loop dimension for the loops involving the indices `j` and `k` from three to two, the number of operations has increased. The reason is that the mapping causes four entries to be computed at the same time inside the loop, and the code to compute each entry has not been reduced significantly if compared to the code in Figure 7.1. In fact, using this optimization strategy by itself is usually not recommended, but in combination with the strategies outlined in the following sections it can improve run-time performance significantly. This

Figure 7.3: Part of the generated code for the weighted Laplacian using linear elements in two dimensions with optimization option `-f simplify_expressions`.

C++ code

```
// Geometry constants.
double G[3];
G[0] = W1*det*(K_00*K_00 + K_01*K_01);
G[1] = W1*det*(K_00*K_10 + K_01*K_11);
G[2] = W1*det*(K_10*K_10 + K_11*K_11);

// Integration point constants.
double I[3];
I[0] = F0*G[0];
I[1] = F0*G[1];
I[2] = F0*G[2];

// Loop basis functions.
for (unsigned int j = 0; j < 3; j++)
{
 for (unsigned int k = 0; k < 3; k++)
 {
  A[j*3 + k] += \
    (Psi_vu_D10[0][j]*Psi_vu_D10[0][k]*I[0] +\
     Psi_vu_D10[0][j]*Psi_vu_D01[0][k]*I[1] +\
     Psi_vu_D01[0][j]*Psi_vu_D10[0][k]*I[1] +\
     Psi_vu_D01[0][j]*Psi_vu_D01[0][k]*I[2]);
 }
}
```

effect is particularly pronounced when forms contain mixed elements in which many of the values in the basis function tables are zero. Another reason for being careful when applying this strategy is that the increase in the number of terms might prevent FFC compilation due to hardware limitations.

7.2.2 *Simplify expressions*

The expressions to evaluate an entry in the local element tensor can become very complex. Since such expressions are typically located inside loops, a reduction in complexity can reduce the total operation count significantly. The approach can be illustrated by the expression $x(y+z)+2xy$, which after expansion of the first term, grouping common terms and simplification can be reduced to $x(3y+z)$, which involves a reduction from five to three operations. An additional benefit of this strategy is that the expansion of expressions, which take place before the simplification, will typically allow more terms to be precomputed and hoisted outside loops, as explained in the beginning of this section. For the weighted Laplace equation, the terms

$$\sum_{\beta=1}^{d}\sum_{\alpha_1=1}^{d}\frac{\partial X_{\alpha_1}}{\partial x_\beta}\frac{\partial \Phi_{i_1}(X^q)}{\partial X_{\alpha_1}}\sum_{\alpha_2=1}^{d}\frac{\partial X_{\alpha_2}}{\partial x_\beta}\frac{\partial \Phi_{i_2}(X^q)}{\partial X_{\alpha_2}} \tag{7.5}$$

will be expanded into

$$\sum_{\beta=1}^{d}\sum_{\alpha_1=1}^{d}\sum_{\alpha_2=1}^{d}\frac{\partial X_{\alpha_1}}{\partial x_\beta}\frac{\partial X_{\alpha_2}}{\partial x_\beta}\frac{\partial \Phi_{i_1}(X^q)}{\partial X_{\alpha_1}}\frac{\partial \Phi_{i_2}(X^q)}{\partial X_{\alpha_2}}, \tag{7.6}$$

where $\left(\partial X_{\alpha_1}/\partial x_\beta\right)\left(\partial X_{\alpha_2}/\partial x_\beta\right)$ is independent of the indices i_1 and i_2 and can therefore be moved outside these loops.

The FFC command-line option `-f simplify_expressions` should be used to generate code with this optimization enabled. Code generated by this option for the representation in (7.4) is presented in Figure 7.3, where again only code different from that in Figure 7.1 has been included.

C++ code

```
// Geometry constants.
double G[1];
G[0] = W1*det;

// Integration point constants.
double I[1];
I[0] = F0*G[0];

// Loop basis functions.
for (unsigned int j = 0; j < 3; j++)
{
  for (unsigned int k = 0; k < 3; k++)
  {
    A[j*3 + k] +=\
    ((Psi_vu_D01[0][j]*K_10 + Psi_vu_D10[0][j]*K_00)*\
     (Psi_vu_D01[0][k]*K_10 + Psi_vu_D10[0][k]*K_00) +\
     (Psi_vu_D01[0][j]*K_11 + Psi_vu_D10[0][j]*K_01)*\
     (Psi_vu_D01[0][k]*K_11 + Psi_vu_D10[0][k]*K_01)
    )*I[0];
  }
}
```

Figure 7.4: Part of the generated code for the weighted Laplacian using linear elements in two dimensions with optimization option `-f precompute_ip_const`.

Due to expansion of the expression, many terms related to the geometry have been moved outside of the loops over the basis function indices `j` and `k` and stored in the array `G`. Also, note how the expressions to compute the values in `G` have been simplified by moving the variables `det` and `W1` outside the parentheses. Similarly, terms that depend only on the integration point are hoisted and stored in the array `I`. The number of operations has decreased compared to the code in Figure 7.1 for the standard quadrature representation. An improvement in run-time performance can therefore be expected.

The optimization described above is the most expensive of the quadrature optimizations to perform in terms of FFC code generation time and memory consumption as it involves creating new terms when expanding the expressions. The procedure does not scale well for complex expressions, but it is in many cases the most effective approach in terms of reducing the number of operations. This particular optimization strategy, in combination with the elimination of zeros outlined in the previous section, was the first to be implemented in FFC. It has been investigated and compared to the tensor representation in Ølgaard and Wells (2010), to which the reader is referred for further details.

7.2.3 *Precompute integration point constants*

The optimizations described in the previous section are performed at the expense of increased code generation time. In order to reduce the generation time while achieving a reduction in the operation count, another approach can be taken involving hoisting expressions that are constant with respect to integration points without expanding the expression first.

To generate code with this optimization the FFC command-line option `-f precompute_ip_const` should be used. Code generated by this method for the representation in (7.4) can be seen in Figure 7.4.

It is clear from the generated code that this strategy will not lead to a significant reduction in the number of operations for this particular form. However, for more complex forms, with many coefficients, the number of terms that can be hoisted will increase significantly, leading to improved run-time performance.

Figure 7.5: Part of the generated code for the weighted Laplacian using linear elements in two dimensions with optimization option `-f precompute_basis_const`. The array `B` contain precomputed values that depend on indices `j` and `k`.

C++ code

```
for (unsigned int j = 0; j < 3; j++)
{
  for (unsigned int k = 0; k < 3; k++)
  {
    double B[16];
    B[0] = Psi_vu_D01[0][j]*K_10;
    B[1] = Psi_vu_D10[0][j]*K_00;
    B[2] = (B[0] + B[1]);
    B[3] = Psi_vu_D01[0][k]*K_10;
    B[4] = Psi_vu_D10[0][k]*K_00;
    B[5] = (B[3] + B[4]);
    B[6] = B[2]*B[5];
    B[7] = Psi_vu_D01[0][j]*K_11;
    B[8] = Psi_vu_D10[0][j]*K_01;
    B[9] = (B[7] + B[8]);
    B[10] = Psi_vu_D01[0][k]*K_11;
    B[11] = Psi_vu_D10[0][k]*K_01;
    B[12] = (B[10] + B[11]);
    B[13] = B[12]*B[9];
    B[14] = (B[13] + B[6]);
    B[15] = B[14]*I[0];
    A[j*3 + k] += B[15];
  }
}
```

7.2.4 *Precompute basis constants*

This optimization strategy is an extension of the strategy described in the previous section. In addition to hoisting terms related to the geometry and the integration points, values that depends on the basis indices are precomputed inside the loops. This will result in a reduction in operations for cases in which some terms appear frequently inside the loop such that a given value can be reused once computed.

To generate code with this optimization, the FFC command-line option `-f precompute_basis_const` should be used. Code generated by this method for the representation in (7.4) can be seen in Figure 7.5, where only code that differs from that in Figure 7.4 has been included. In this particular case, no additional reduction in operations has been achieved, if compared to the previous method, since no terms can be reused inside the loop over the indices `j` and `k`.

7.2.5 *Future optimizations*

Preliminary investigations suggest that the performance of the quadrature representation can be improved by applying two additional optimizations. Looking at the code in Figure 7.5, we see that about half of the temporary values in the array `B` only depend on the loop index `j`, and they can therefore be hoisted, as we have done for other terms in previous sections. Another approach is to unroll the loops with respect to `j` and `k` in the generated code. This will lead to a dramatic increase in the number of values that can be reused, and the approach can be readily combined with all of the other optimization strategies. However, the total number of temporary values will also increase. Therefore, this optimization strategy might not be feasible for all forms.

FFC implements a few efficient quadrature schemes for integrating polynomials of degree less than or equal to six on simplices. For polynomials of degree higher than six, it calls FIAT (see Chapter 13) to compute the quadrature scheme. FIAT supplies schemes that are based on the Gauss–Legendre–Jacobi rule mapped onto simplices (see Karniadakis and Sherwin (2005) for details of such schemes). This

means that for integrating a seventh-order polynomial, FFC will use four quadrature points in each spatial direction that is, $4^3 = 64$ points per cell in three dimensions. A further optimization of the quadrature representation can thus be achieved by implementing more efficient quadrature schemes for higher order polynomials on simplices since a reduction in the number of integration points will yield improved run-time performance. FFC does, however, provide an option for a user to specify the quadrature degree of a variational form thereby permitting inexact quadrature. To set the quadrature degree equal to one, the command-line option `-f quadrature_degree=1` should be used.

7.3 Performance comparisons

In this section we investigate the impact of the optimization strategies outlined in the previous section on the run-time performance. The point is not to present a rigorous analysis of the optimizations, but to provide indications as to when the different strategies will be most effective. We also compare the run-time performance of quadrature representation to the tensor representation, which is described in Chapter 8, to illustrate the strengths and weaknesses of the two approaches.

7.3.1 Performance of quadrature optimizations

The performance of the quadrature optimizations will be investigated using two forms, namely the bilinear form for the weighted Laplace equation (7.1) and the bilinear form for the hyperelasticity model presented in Chapter 17, equation (17.6). In both cases quadratic Lagrange finite elements will be used.

All tests were performed on an Intel Pentium M CPU at 1.7GHz with 1.0GB of RAM running Ubuntu 11.10 with Linux kernel 2.6.38. We used Python version 2.7.2 and NumPy version 1.5.1 (both pertinent to FFC), and g++ version 4.6.1 to compile the UFC version 2.0.2 compliant C++ code.

The two forms are compiled with the different FFC optimizations, and the number of floating point operations (flops) to compute the local element tensor is determined. We define the number of flops as the sum of all appearances of the operators '+' and '$*$' in the code. The ratio between the number of flops of the current FFC optimization and the standard quadrature representation, 'o/q' is also computed. The generated code is then compiled with g++ using four different optimization options and the time needed to compute the element tensor N times is measured. In the following, we will use `-zeros` as shorthand for the `-f eliminate_zeros` option, `-simplify` as shorthand for the `-f simplify_expressions` option, `-ip` as shorthand for the `-f precompute_ip_const` option and `-basis` as shorthand for the `-f precompute_basis_const` option.

The operation counts for the weighted Laplace equation with different FFC optimizations can be seen in Table 7.1, while Figure 7.6 shows the run-time performance for different compiler options for $N = 1 \times 10^7$. The FFC compiler options can be seen on the x-axis in the figure and the four g++ compiler options are shown with different colors.

The FFC and g++ compile-times were less than two seconds for all optimization options. It is clear from Figure 7.6 that run-time performance is greatly influenced by the g++ optimizations. Compared to the case where no g++ optimizations are used (the `-O0` flag), the run-time for the standard quadrature code improves by a factor of 3.31 when using the `-O2` option, 5.23 when using the `-O2 -funroll-loops` option and 7.75 when using the `-O3` option. The `-O3` option does not appear to improve the run-time noticeably beyond the improvement observed for the `-O2 -funroll-loops` option when the FFC optimization option `-zeros` is used. Using the FFC optimization option `-zeros` alone for this form does not improve run-time performance. In fact, using this option in combination with any of the other optimization options increases the run-time, even when combining with the option `-simplify`, which has a significant lower operation count compared to the standard quadrature

FFC optimization	flops	o/q
`None`	4176	1.00
`-zeros`	6672	1.60
`-simplify`	2712	0.65
`-simplify -zeros`	1920	0.46
`-ip`	3756	0.90
`-ip -zeros`	4290	1.03
`-basis`	3756	0.90
`-basis -zeros`	3690	0.88

Table 7.1: Operation counts for the weighted Laplace equation.

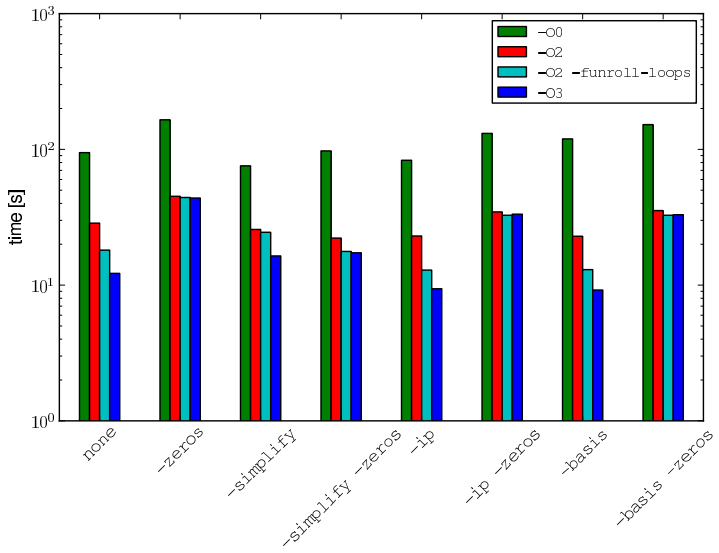


Figure 7.6: Run-Time performance for the weighted Laplace equation for different compiler options. The x-axis shows the FFC compiler options, and the colors denote the g++ compiler options.

FFC optimization	FFC time [s]	o/q	flops	o/q
None	6.8	1.00	70794360	1.000
-zeros	6.9	1.01	31415076	0.444
-simplify	19.6	2.88	3086370	0.044
-simplify -zeros	18.6	2.74	185472	0.003
-ip	12.8	1.88	46742697	0.660
-ip -zeros	15.1	2.22	7673022	0.108
-basis	13.0	1.91	3852297	0.054
-basis -zeros	15.5	2.28	1023624	0.014

Table 7.2: FFC compile-times and operation counts for the hyperelasticity example.

representation. A curious point to note is that without g++ optimization there is a significant difference in run-time for the `-ip` and `-basis` options, even though they involve the same number of flops. When g++ optimizations are switched on, this difference is eliminated completely and the run-time for the two FFC optimizations are identical. This suggests that it is not possible to predict run-time performance from the operation count alone since the type of FFC optimization must be taken into account as well as the intended use of g++ compiler options. The optimal combination of optimizations for this form is FFC option `-ip` or `-basis` combined with g++ option `-O3`, in which case the run-time has improved by a factor of 10.2 compared to standard quadrature code with no g++ optimizations.

The operation counts and FFC compile-time for the bilinear form for hyperelasticity with different FFC optimizations are presented in Table 7.2, while Figure 7.7 shows the run-time performance for different compiler options for $N = 1 \times 10^4$. Comparing the number of flops involved to compute the element tensor to the weighted Laplace example, it is clear that this problem is considerably more complex. The FFC compile-times in Table 7.2 show that the `-simplify` optimization, as anticipated, is the most expensive to perform. The g++ compile-times for all test cases were in the range two to seven seconds for all optimization options. A point to note is that the scope for reducing the flop count is considerably greater for this problem than for the weighted Laplace problem, with a difference in the number of flops spanning several orders of magnitude between the different FFC optimizations. This compares to a difference in flops of roughly a factor two between the non-optimized and the most effective optimization strategy for the weighted Laplace problem. In the case where no g++ optimization is used the run-time performance for the hyperelastic problem can be directly related to the number of floating point operations. When the g++ optimization `-O2` is switched on, this effect becomes less pronounced. Another point to note, in connection with the g++ optimizations, is that switching on additional optimizations beyond `-O2` does not seem to provide any further improvements in run-time. For the hyperelasticity example, the option `-zeros` has a positive effect on the performance, not only when used alone but in particular when combined with the other FFC optimizations. This is in contrast with the weighted Laplace equation. The reason is that the test and trial functions are vector valued rather than scalar valued, which allows more zeros to be eliminated. Finally, it is noted that the `-simplify` option performs particularly well for this example compared to the weighted Laplace problem. The reason is that the nature of the hyperelasticity form results in a relatively complex expression to compute the entries in the local element tensor. However, this expression only consists of a few different variables (components of the inverse of the Jacobian and basis function values) which makes the `-simplify` option very efficient since many terms are common and can be precomputed and hoisted. For the hyperelasticity form, the optimal combination of optimizations is FFC option `-simplify -zeros` and g++ option `-O2 -funroll-loops` which improves the run-time performance of the code by a factor of 335 when compared to the case

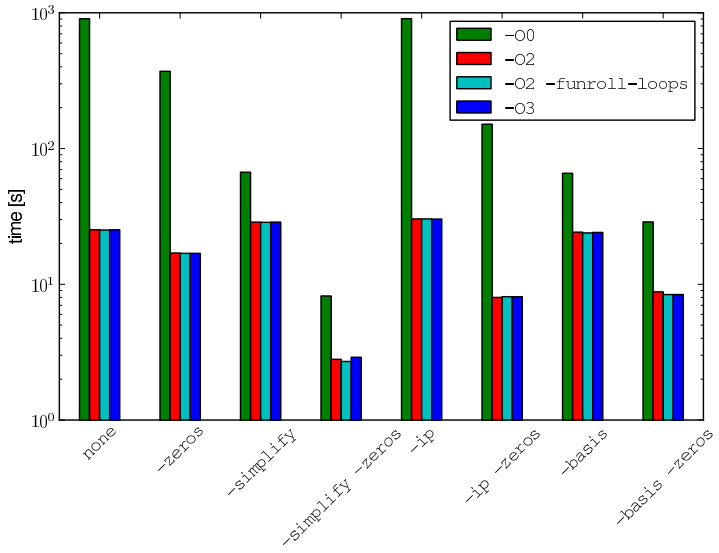


Figure 7.7: Run-Time performance for the hyperelasticity example for different compiler options. The x-axis shows the FFC compiler options, and the colors denote the g++ compiler options.

where no optimization is used by either FFC or g++.

For the considered examples, it is clear that no single optimization strategy is the best for all cases. Furthermore, the generation phase optimizations that one can best use depends on which optimizations are performed by the g++ compiler. It is also very likely that different C++ compilers will give different results for the test cases presented above. The general recommendation for selecting the appropriate optimization for production code will therefore be that the choice should be based on a benchmark program for the specific problem.

7.3.2 *Relative performance of the quadrature and tensor representations*

As demonstrated in the previous section, a given type of optimization may be effective for one class of forms, and be less effective for another class of forms. Similarly, differences can be observed between the quadrature and tensor representations for different equations. A detailed study on this issue was carried out in Ølgaard and Wells (2010). For convenience we reproduce here the main conclusions along with Table 7.3, which has been reproduced from the paper. The results shown in this section pertain to an elasticity-like bilinear form in two dimensions that is premultiplied by a number of scalar coefficients f_i:

$$a(u,v) = \int_\Omega (f_0 f_1, \ldots, f_{n_f}) \, \nabla^s u : \nabla^s v \, \mathrm{d}x, \tag{7.7}$$

where n_f is the number of premultiplying coefficients. The test and trial functions are denoted by $v, u \in V_h$, with

$$V_h = \left\{ v \in [H^1(\Omega)]^2 : \; v|_T \in [P_q(T)]^2 \forall \, T \in \mathcal{T} \right\} \tag{7.8}$$

and the coefficient functions $f_i \in W_h$ with

$$W_h = \left\{ f \in H^1(\Omega) : \; f|_T \in P_p(T) \, \forall \, T \in \mathcal{T} \right\}, \tag{7.9}$$

	$n_f = 1$		$n_f = 2$		$n_f = 3$	
	flops	q/t	flops	q/t	flops	q/t
$p = 1, q = 1$	888	0.34	3060	0.36	10224	0.11
$p = 1, q = 2$	3564	1.42	11400	1.01	35748	0.33
$p = 1, q = 3$	10988	3.23	34904	1.82	100388	0.63
$p = 1, q = 4$	26232	5.77	82548	2.87	254304	0.93
$p = 2, q = 1$	888	1.20	8220	0.31	54684	0.09
$p = 2, q = 2$	7176	1.59	41712	0.49	284232	0.11
$p = 2, q = 3$	22568	2.80	139472	0.71	856736	0.17
$p = 2, q = 4$	54300	4.36	337692	1.01	2058876	0.23
$p = 3, q = 1$	3044	0.36	30236	0.16	379964	0.02
$p = 3, q = 2$	12488	0.92	126368	0.26	1370576	0.03
$p = 3, q = 3$	36664	1.73	391552	0.37	4034704	0.05
$p = 3, q = 4$	92828	2.55	950012	0.49	9566012	0.06
$p = 4, q = 1$	3660	0.68	73236	0.11	1275624	0.01
$p = 4, q = 2$	17652	1.16	296712	0.16	4628460	0.02
$p = 4, q = 3$	57860	1.71	903752	0.22	13716836	0.02
$p = 4, q = 4$	138984	2.46	2133972	0.29	32289984	0.03

Table 7.3: The number of operations and the ratio between number of operations for the two representations for the elasticity-like tensor in two dimensions as a function of different polynomial orders and numbers of functions (taken from Ølgaard and Wells (2010).

where q and p denote the polynomial order of the Lagrange basis functions. The number of coefficients and the polynomial orders are varied and the number of flops needed to compute the local element tensor is recorded for both tensor and quadrature representations. The results were obtained by using the optimization options `-f eliminate_zeros -f simplify_expressions` for the quadrature representation. In Table 7.3 the flops for the tensor representation is presented together with the ratio given by the flops for quadrature representation divided by the flops for tensor representation, denoted by q/t. In terms of flops, a ratio $q/t > 1$ indicates that the tensor representation is more efficient while $q/t < 1$ indicates that the quadrature representation is more efficient. It was found that when comparing the run-time performance of the two representations for this problem that the number of flops is a good indicator of performance. However, as we have shown in the previous section, the quadrature code with the lowest number of flops does not always perform best for a given form. Furthermore, the run-time performance even depends on which g++ options are used. This begs the question of whether or not it is possible to make a sound selection between representations based only on an estimation of flops, as suggested in Ølgaard and Wells (2010).

Nevertheless, some general trends can still be read from the table. Increasing the number of coefficient functions n_f in the form clearly works in favor of quadrature representation. For $n_f = 3$ the quadrature representation can be expected to perform best for all values of q and p. Increasing the polynomial order of the coefficients, p, also works in favor of quadrature representation although the effect is less pronounced compared to the effect of increasing the number of coefficients. The tensor representation appears to perform better when the polynomial order of the test and trial functions, q, is increased although the effect is most pronounced when the number of coefficients is low.

7.4 Automatic selection of representation

We have illustrated how the run-time performance of the generated code for variational forms can be improved by using various optimization options for the FFC and g++ compilers, and by changing

the representation of the form. Choosing the combination of form representation and optimization options that leads to optimal performance will inevitably require a benchmark study of the specific problem. However, very often many variational forms of varying complexity are needed to solve more complex problems. Setting up benchmarks for all of them is cumbersome and time consuming. Additionally, during the model development stage run-time performance is of minor importance compared to rapid prototyping of variational forms as long as the generated code performs reasonably well.

The default behavior of FFC is, therefore, to automatically determine which form representation should be used based on a measure for the cost of using the tensor representation. In short, the cost is simply computed as the maximum value of the sum of the number of coefficients and derivatives present in the monomials representing the form. If this cost is larger than a specified threshold, currently set to three, the quadrature representation is selected. Recall from Table 7.3 that when $n_f = 3$ the flops for quadrature representation was significantly lower for virtually all the test cases. Although this approach may seem *ad hoc*, it will work well for those situations where the difference in run-time performance is significant. It is important to remember that the generated code is only concerned with the evaluation of the local element tensor and that the time needed to insert the values into a sparse matrix and to solve the system of equations will reduce any difference, particularly for simple forms. Therefore, making a correct choice of representation is less important for forms where the difference in run-time performance is small. A future improvement could be to devise a strategy for also letting the system select the optimization strategy for the quadrature representation automatically.

8 Tensor representation of finite element variational forms

By Robert C. Kirby and Anders Logg

In Chapter 6, we saw that an important step in the assembly of matrices and vectors for the discretization of finite element variational problems is the evaluation of the cell (element) tensor A_T defined by

$$A_{T,i} = a_T(\phi_{i_\rho}^{T,\rho}, \ldots, \phi_{i_2}^{T,2}, \phi_{i_1}^{T,1}). \tag{8.1}$$

Here, a_T is the local contribution to a multilinear form $a : V_\rho \times \cdots \times V_2 \times V_1 \to \mathbb{R}$, $i = (i_1, i_2, \ldots, i_\rho)$ is a multi-index of length ρ, and $\{\phi_k^{T,j}\}_{k=1}^{n_j}$ is a basis for the local finite element space of $V_{j,h} \subset V_j$ on a local cell T for $j = 1, 2, \ldots, \rho$. In this chapter, we describe how the cell tensor A_T can be computed efficiently by an approach referred to as *tensor representation*.

8.1 Tensor representation for Poisson's equation

We first describe how one may express the cell tensor for Poisson's equation as a special tensor contraction and explain below how this may be generalized to other variational forms. For Poisson's equation, the cell tensor (matrix) A_T is defined by

$$A_{T,i} = \int_T \nabla\phi_{i_1}^{T,1} \cdot \nabla\phi_{i_2}^{T,2} \,\mathrm{d}x = \int_T \sum_{\beta=1}^{d} \frac{\partial\phi_{i_1}^{T,1}}{\partial x_\beta} \frac{\partial\phi_{i_2}^{T,2}}{\partial x_\beta} \,\mathrm{d}x. \tag{8.2}$$

Let $F_T : \hat{T} \to T$ be an affine map from a reference cell $\hat{T}$ to the current cell T as illustrated in Figure 8.1. Using this affine map, we make a change of variables to obtain

$$A_{T,i} = \int_{\hat{T}} \sum_{\beta=1}^{d} \sum_{\alpha_1=1}^{d} \frac{\partial\hat{x}_{\alpha_1}}{\partial x_\beta} \frac{\partial\hat{\phi}_{i_1}^{1}}{\partial\hat{x}_{\alpha_1}} \sum_{\alpha_2=1}^{d} \frac{\partial\hat{x}_{\alpha_2}}{\partial x_\beta} \frac{\partial\hat{\phi}_{i_2}^{2}}{\partial\hat{x}_{\alpha_2}} \det F_T' \,\mathrm{d}\hat{x}. \tag{8.3}$$

Here, $\hat{\phi}_i^j = \phi_i^{T,j} \circ F_T$ denotes the basis function on the reference cell $\hat{T}$ corresponding to the basis function $\phi_i^{T,j}$ on the current cell T. Since F_T is affine, the partial derivatives $\partial\hat{x}/\partial x$ and the determinant $\det F_T'$ are constant. We thus obtain

$$A_{T,i} = \det F_T' \sum_{\alpha_1=1}^{d} \sum_{\alpha_2=1}^{d} \sum_{\beta=1}^{d} \frac{\partial\hat{x}_{\alpha_1}}{\partial x_\beta} \frac{\partial\hat{x}_{\alpha_2}}{\partial x_\beta} \int_{\hat{T}} \frac{\partial\hat{\phi}_{i_1}^{1}}{\partial\hat{x}_{\alpha_1}} \frac{\partial\hat{\phi}_{i_2}^{2}}{\partial\hat{x}_{\alpha_2}} \,\mathrm{d}\hat{x} = \sum_{\alpha_1=1}^{d} \sum_{\alpha_2=1}^{d} A_{i\alpha}^0 G_T^\alpha, \tag{8.4}$$

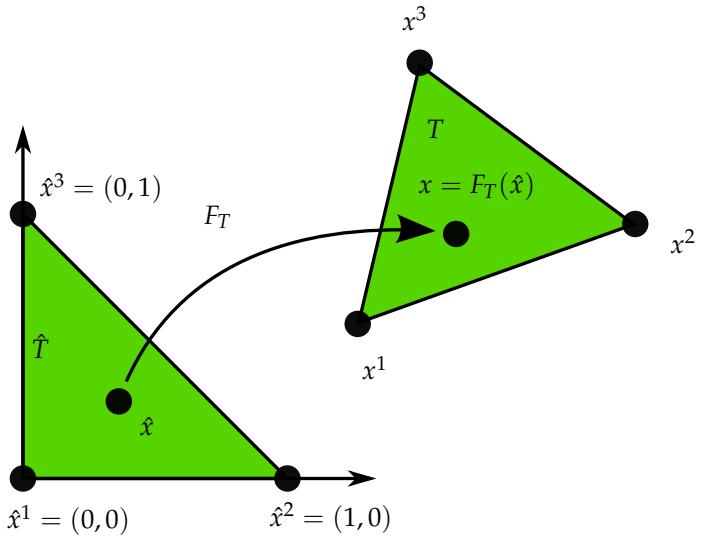


Figure 8.1: The (affine) map F_T from a reference cell $\hat{T}$ to a cell $T \in \mathcal{T}_h$.

where

$$\begin{aligned} A^0_{i\alpha} &= \int_{\hat{T}} \frac{\partial \hat{\phi}^1_{i_1}}{\partial \hat{x}_{\alpha_1}} \frac{\partial \hat{\phi}^2_{i_2}}{\partial \hat{x}_{\alpha_2}} \, \mathrm{d}\hat{x}, \\ G^\alpha_T &= \det F'_T \sum_{\beta=1}^{d} \frac{\partial \hat{x}_{\alpha_1}}{\partial x_\beta} \frac{\partial \hat{x}_{\alpha_2}}{\partial x_\beta}. \end{aligned} \tag{8.5}$$

We refer to the tensor A^0 as the *reference tensor* and to the tensor G_T as the *geometry tensor*. We may thus express the computation of the cell tensor A_T for Poisson's equation as the tensor contraction

$$A_T = A^0 : G_T. \tag{8.6}$$

This tensor contraction may be computed efficiently by precomputing the entries of the reference tensor A^0. This is possible since the reference tensor is constant and does not depend on the cell T or the mesh $\mathcal{T}_h = \{T\}$. On each cell $T \in \mathcal{T}_h$, the cell tensor may thus be computed by first computing the geometry tensor G_T and then contracting it with the precomputed reference tensor. In Chapter 11, we describe the FEniCS Form Compiler (FFC) which precomputes the reference tensor A^0 at compile-time and generates code for computing the tensor contraction.

For Poisson's equation in two space dimensions, the tensor contraction involves contracting the 2×2 geometry tensor G_T with each corresponding block of the $3 \times 3 \times 2 \times 2$ reference tensor A^0 to form the entries of the 3×3 cell tensor A^T. Each of these entries may thus be computed in only four multiply-add pairs (plus the cost of computing the geometry tensor). This brings a considerable speedup compared to evaluation by run-time quadrature, in particular for higher-order elements. In Chapter 9, we discuss how this may be improved further by examining the structure of the reference tensor A^0 to find a reduced-arithmetic computation for the tensor contraction.

$a(u,v)$	$=$	$\langle u, v \rangle$	rank
$A^0_{i\alpha}$	$=$	$\int_{\hat{T}} \hat{\phi}^1_{i_1} \hat{\phi}^2_{i_2} \, \mathrm{d}\hat{x}$	$\lvert i\alpha \rvert = 2$
G^α_T	$=$	$\det F'_T$	$\lvert \alpha \rvert = 0$

Table 8.1: Tensor representation $A_T = A^0 : G_T$ of the cell tensor A_T for the bilinear form associated with a mass matrix.

$a(w;u,v)$	$=$	$\langle w \cdot \nabla u, v \rangle$	rank
$A^0_{i\alpha}$	$=$	$\sum_{\beta=1}^{d} \int_{\hat{T}} \frac{\partial \hat{\phi}^2_{i_2}[\beta]}{\partial \hat{x}_{\alpha_3}} \hat{\phi}^3_{\alpha_1}[\alpha_2] \hat{\phi}^1_{i_1}[\beta] \, \mathrm{d}\hat{x}$	$\lvert i\alpha \rvert = 5$
G^α_T	$=$	$w^T_{\alpha_1} \det F'_T \frac{\partial \hat{x}_{\alpha_3}}{\partial x_{\alpha_2}}$	$\lvert \alpha \rvert = 3$

Table 8.2: Tensor representation $A_T = A^0 : G_T$ of the cell tensor A_T for the bilinear form associated with advection $w \cdot \nabla u$. It is assumed that the velocity field w may be interpolated into a local finite element space with expansion coefficients $w^T_{\alpha_1}$. Note that w is a vector-valued function, the components of which are referenced by $w[\beta]$.

8.2 A representation theorem

In Kirby and Logg (2006), it was proved that the cell tensor for any affinely mapped monomial multilinear form may be expressed as a sum of tensor contractions:

$$A_T = \sum_k A^{0,k} : G_{T,k}. \tag{8.7}$$

By a monomial multilinear form, we here mean a multilinear form that can be expressed as a sum of monomials, where each monomial is a product of coefficients, trial/test functions and their derivatives. This class covers all forms that may be expressed by addition, multiplication and differentiation. Early versions of the form compiler FFC implemented a simple form language that was limited to these three operations. This simple form language is now replaced by the new and more expressive UFL form language. As a result, the tensor representation can only be a applied to a subset of the forms that can be expressed in UFL. This is automatically checked by FFC; if tensor representation is not applicable, then numerical quadrature is used.

The representation theorem was later extended to Piola-mapped elements in Rognes et al. (2009), and in Ølgaard et al. (2008) it was demonstrated how the tensor representation may be computed for discontinuous Galerkin methods.

The ranks of the reference and geometry tensors are determined by the multilinear form a, in particular by the number of coefficients and derivatives of the form. Since the rank of the cell tensor A_T is equal to the arity ρ of the multilinear form a, the rank of the reference tensor A^0 must be $|i\alpha| = \rho + |\alpha|$, where $|\alpha|$ is the rank of the geometry tensor. For Poisson's equation, we have $|i\alpha| = 4$ and $|\alpha| = 2$. In Tables 8.1 and 8.2, we demonstrate how the tensor representation may be computed for the bilinear forms $a(u,v) = \langle u, v \rangle$ (mass matrix) and $a(w;u,v) = \langle w \cdot \nabla u, v \rangle$ (advection).

8.3 Extensions and limitations

The tensor contraction (8.7) assumes that the map F_T from the reference cell is affine, allowing the transforms $\partial \hat{x} / \partial x$ and the determinant $\det F'_K$ to be pulled out of the integral. If the map is

non-affine (sometimes called a "higher-order" map), one may expand it in the basis functions of the corresponding finite element space and pull the coefficients outside the integral, as done for the advection term from Table 8.2. Alternatively, one may evaluate the cell tensor by quadrature and express the summation over quadrature points as a tensor contraction as explained in Kirby and Logg (2006). As noted above, the tensor contraction readily extends to basis functions mapped by Piola transforms.

One limitation of this approach is it requires each basis function on a cell T to be the image of a single reference element basis function under an affine Piola transformation. While this covers a wide range of commonly used elements, it does not include certain kinds of elements with derivative-based degrees of freedom such as the Hermite and Argyris elements. Let $\mathcal{F}_T$ be the mapping of the reference element function space to the function space over the cell T, such as the affine map or Piola transform. Then the physical element basis functions can be expressed as linear combinations of the transformed reference element basis functions:

$$\phi_i^T = \sum_{j=1}^{n} M_{T,ij} \mathcal{F}_T \left(\hat{\phi}_j \right) . \tag{8.8}$$

The structure of this matrix M_T depends on the kinds of degrees freedom, and the values typically vary for each cell T based on the cell geometry. Frequently, the matrix M_T is sparse. Given M_T, the tensor-contraction framework may be extended to handle these more general elements. As before, one may compute the reference tensor A^0 by mapping the reference element basis functions. But in addition, the tensor contraction $A^0 : G_T$ must be corrected by acting on it with the matrix M_T. This is currently not implemented in the form compiler FFC and thus FEniCS does not support Hermite and Argyris elements.

For many simple variational forms, such as those for Poisson's equation, the mass matrix and the advection term discussed above, the tensor contraction (8.7) leads to significant speedups over numerical quadrature, sometimes as much as several orders of magnitude. However, as the complexity of a form increases, the relative efficiency of quadrature also increases. In simple terms, the complexity of a form can be measured as the number of derivatives and the number of coefficients appearing in a form. For each derivative and coefficient, the rank of the reference tensor A^0 increases by one. Thus, for Poisson's equation, the rank is $2 + 2 = 4$ since the form has two derivatives and for the mass matrix, the rank is $2 + 0$ since there are neither derivatives nor coefficients. For the advection term, the rank is $2 + 2 + 1 = 5$ since the form has one derivative, one coefficient, and also an inner product $w \cdot \nabla$. Since the size of the reference tensor A^0 grows exponentially with its rank, the tensor contraction may become very costly for forms of high complexity. In these cases, quadrature is more efficient. Quadrature may sometimes also be the only available option as the tensor contraction is not directly applicable to forms that are not expressed as simple sums of products of coefficients, trial/test functions and their derivatives. For this reason, it is important to be able to choose between both approaches; tensor representation may sometimes be the most efficient approach whereas in other cases quadrature is more efficient or even the only possible alternative. Such trade-offs are discussed in Chapter 7 and Chapter 12.

9 *Discrete optimization of finite element matrix evaluation*

By Robert C. Kirby, Matthew Gregg Knepley, Anders Logg, L. Ridgway Scott and Andy R. Terrel

The tensor contraction structure for the computation of the element tensor A_T obtained in Chapter 8, enables not only the construction of a compiler for variational forms, but an *optimizing* compiler. For typical variational forms, the reference tensor A^0 has significant structure that allows the element tensor A_T to be computed on an arbitrary cell T at a lower computational cost. Reducing the number of operations by making use of this structure, leads naturally to several problems in discrete mathematics. This chapter introduces some of the optimizations that are possible, and discusses compile-time combinatorial optimization problems that form the core of the FErari project (Kirby et al., 2006; Kirby and Scott, 2007; Kirby and Logg, 2008), which is the subject of Chapter 12.

We consider two basic kinds of optimizations in this chapter. First, we consider relations between pairs of rows in the reference tensor. This naturally leads to a graph that models proximity among these pairs. If two rows are "close" together, then one may reuse results computed with the first row to compute a desired quantity with the second. The proximity of two such rows is computed using a Hamming distance and linearity relations. This approach gives rise to a weighted graph that is (almost) a metric space, so we designate such optimizations as "topological". Second, we consider relations between more than two rows of the reference tensor. Such relations typically rely on sets of rows, considered as vectors in Euclidean space. Because we are using planes and hyperplanes to reduce the amount of computation, we describe these optimizations as "geometric". For comparison, we briefly discuss optimizations using more traditional optimized dense linear algebra packages.

9.1 Optimization framework

The tensor paradigm developed in Chapter 8 arrives at the representation

$$A_{T,i} = \sum_{\alpha \in \mathcal{A}} A^0_{i\alpha} G^\alpha_T \quad \forall\, i \in \mathcal{I}, \tag{9.1}$$

or simply

$$A_T = A^0 : G_T, \tag{9.2}$$

where $\mathcal{I}$ is the set of admissible multi-indices for the element tensor A_T and $\mathcal{A}$ is the set of admissible multi-indices for the geometry tensor G_T. The reference tensor A^0 can be computed at compile-time, and may then be contracted with a G_T to obtain the element tensor A_T for each cell T in the finite element mesh at run-time. The case of computing local finite element stiffness matrices of size $n_T \times n_T$ corresponds to $\mathcal{I}$ consisting of $|\mathcal{I}| = n_T^2$ multi-indices of length two, where n_T is the dimension of the

local finite element space on T.

It is convenient to recast (9.2) in terms of a matrix–vector product:

$$A^0 : G_T \leftrightarrow \tilde{A}^0 \tilde{g}_T. \tag{9.3}$$

Here, the matrix $\tilde{A}^0$ lies in $\mathbb{R}^{|\mathcal{I}|\times|\mathcal{A}|}$, and the vector $\tilde{g}_T$ lies in $\mathbb{R}^{|\mathcal{A}|}$. The resulting matrix–vector product can then be reshaped into the element tensor A_T. As this computation must occur for each cell T in a finite element mesh, it makes sense to try to make this operation as efficient as possible.

In the following, we will drop the subscripts and superscripts of (9.3) and consider the problem of computing a general matrix–vector product

$$y = Ax, \tag{9.4}$$

where $A = \tilde{A}^0$ is a constant matrix known *a priori*, and $x = \tilde{g}_T$ is an arbitrary vector. We will study structure of A that allows for a reduction in the number of arithmetic operations required to form these products. With this structure, we are able to produce a routine that computes the action of the system in less operations than would be performed using general sparse or dense linear algebra routines.

Before proceeding with the mathematical formulation, we give an example of a matrix A that we would like to optimize. In (9.5), we display the reference tensor A^0 for computing a standard stiffness matrix discretizing a two–dimensional Laplacian with quadratic Lagrange elements on triangles. The rank four tensor is depicted here as a 6×6 matrix of 2×2 matrices. Full analysis would use a corresponding flattened 36×4 matrix A.

$$A^0 = \left(\begin{array}{cc|cc|cc|cc|cc|cc} 3 & 0 & 0 & -1 & 1 & 1 & -4 & -4 & 0 & 4 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 3 & 1 & 1 & 0 & 0 & 4 & 0 & -4 & -4 \\ \hline 1 & 0 & 0 & 1 & 3 & 3 & -4 & 0 & 0 & 0 & 0 & -4 \\ 1 & 0 & 0 & 1 & 3 & 3 & -4 & 0 & 0 & 0 & 0 & -4 \\ \hline -4 & 0 & 0 & 0 & -4 & -4 & 8 & 4 & 0 & -4 & 0 & 4 \\ -4 & 0 & 0 & 0 & 0 & 0 & 4 & 8 & -4 & -8 & 4 & 0 \\ \hline 0 & 0 & 0 & 4 & 0 & 0 & 0 & -4 & 8 & 4 & -8 & -4 \\ 4 & 0 & 0 & 0 & 0 & 0 & -4 & -8 & 4 & 8 & -4 & 0 \\ \hline 0 & 0 & 0 & -4 & 0 & 0 & 0 & 4 & -8 & -4 & 8 & 4 \\ 0 & 0 & 0 & -4 & -4 & -4 & 4 & 0 & -4 & 0 & 4 & 8 \end{array}\right) \tag{9.5}$$

9.2 Topological optimization

It is possible to apply the matrix A, corresponding to the reference tensor A^0 depicted in (9.5), to an arbitrary vector x in fewer operations than the 144 multiply–add pairs required by a standard matrix–vector multiplication. This requires offline analysis of A and special-purpose code generation that applies the particular A to a generic x. For $A \in \mathbb{R}^{M\times N}$, let $\{a^i\}_{i=1}^M \subset \mathbb{R}^N$ denote the rows of A. The vector $y = Ax$ may then be computed by M dot products of the form $y_i = a^i x$. Below, we investigate relationships among the rows of A to find an optimized computation of the matrix–vector product.

For the purpose of illustration, we consider the following subset of (9.5), which would only cost 40

multiply–add pairs but contains all the relations we use to optimize the larger version:

$$A = \begin{pmatrix} a^1 \leftrightarrow A^0_{1,3} \\ a^2 \leftrightarrow A^0_{1,4} \\ a^3 \leftrightarrow A^0_{2,3} \\ a^4 \leftrightarrow A^0_{3,3} \\ a^5 \leftrightarrow A^0_{4,6} \\ a^6 \leftrightarrow A^0_{4,4} \\ a^7 \leftrightarrow A^0_{4,5} \\ a^8 \leftrightarrow A^0_{5,6} \\ a^9 \leftrightarrow A^0_{6,1} \\ a^{10} \leftrightarrow A^0_{6,6} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 & 0 \\ -4 & -4 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 3 & 3 & 3 & 3 \\ 0 & 4 & 4 & 0 \\ 8 & 4 & 4 & 8 \\ 0 & -4 & -4 & -8 \\ -8 & -4 & -4 & 0 \\ 0 & 0 & 0 & 0 \\ 8 & 4 & 4 & 8 \end{pmatrix}. \tag{9.6}$$

Inspection of (9.6) shows that a^9 is zero; therefore, it does not need to be multiplied by the entries of x. In particular, if z entries of a^i are zero, then the dot product $a^i x$ requires $N - z$ multiply–add pairs rather than N.

If $a^i = a^j$ for some $i \neq j$, as seen in the sixth and tenth rows of A, then it follows that $y_i = y_j$, and only one dot product needs to be performed instead of two. A similar case is where $\alpha a^i = a^j$ for some number α, as in the first and second rows of A. This means that after y_i has been computed, $y_j = \alpha y_i$ may be computed with a single multiplication.

In addition to equality and collinearity discussed above, one may also consider other relations between the rows of A. Further inspection of A in (9.6) reveals rows that have some entries in common but are neither equal nor collinear. Such rows have a small *Hamming distance*; that is, the number of entries in which the two rows differ is small. This occurs frequently, as seen in, for example, rows five and six . We can write $a^j = a^i + (a^j - a^i)$, where $a^j - a^i$ has $d_H \leqslant N$ nonzero entries and where d_H is the Hamming distance between a^i and a^j. Once y_i has been computed, one may thus compute y_j as

$$y_j = y_i + \left(a^j - a^i\right) x, \tag{9.7}$$

which requires only d_H additional multiply–add pairs. If d_H is small compared to N, the savings are considerable.

In Wolf and Heath (2009), these binary relations are extended to include the partial collinearity of two vectors. For example, the sixth and seventh rows have parts that are collinear, namely $a^6_{2:4} = -a^7_{2:4}$. This allows y_j to be computed via:

$$y_j = \alpha(y_i - y_{i,\text{nonmatching}}) + a^j_{\text{nonmatching}}\, x, \tag{9.8}$$

where the subscript indicates non-matching portions of the vectors padded with zeroes. Such relationships reduce the computation of y_j to the subtraction of the non-matching contributions, a scaling of the result computed with y^i, and then an additional multiplication with the non-matching entries in a^j.

All of these examples of structure relate to either a single row of A or a pair of rows of A. Such *binary* relations between pairs of rows are amenable to the formulation of graph-theoretic structures, as is developed in Section 9.3. Higher-order relations also occur between the rows of A. For example, the first and third rows may be added and scaled to make the fourth row. In this case, once $a^1 x$ and $a^3 x$ are known, the results may be used to compute $a^4 x$ using one addition and one multiplication, compared to four multiplications and three additions for direct evaluation of the dot product $a^4 x$.

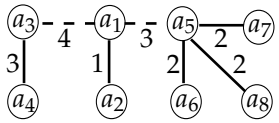


Figure 9.1: Minimum spanning tree (forest) for the vectors in (9.6). The dashed edges represent edges that do not reduce the number of operations (relative to $N - z$) and thus disconnect the graph.

9.3 A graph problem

If we restrict consideration to binary relations between the rows of A, we are led naturally to a weighted, undirected graph whose vertices are the rows a^i of A. An edge between a^i and a^j with weight d indicates that if $a^i x$ is known, then that result may be used to compute $a^j x$ with d multiply–add pairs. In practice, such edges also need to be labeled with information indicating the kind of relationship such as equality, collinearity or a low Hamming distance.

To find the optimal computation through the graph, we use Prim's algorithm (Prim, 1957) for computing a minimum-spanning tree. A minimum spanning tree is a tree that connects all the vertices of the graph and has minimum total edge weight. In Kirby et al. (2006), it is demonstrated that, under a given set of relationships between rows, a minimum spanning tree in fact encodes an algorithm that optimally reduces the number of arithmetic operations required. This discussion assumes that the initial graph is connected. In principle, every a^i is no more than a distance of N away from any a^j. In practice, however, only edges with $d < N - z$ are included in a graph since N is the cost of computing y_i without reference to y_j. This often makes the graph unconnected and thus one must construct a minimum spanning *forest* instead of a tree (a set of disjoint trees that together touch all the vertices of the graph). An example of a minimum spanning tree using the binary relations is shown in Figure 9.1.

Such a forest may then be used to determine an efficient algorithm for evaluating Ax as follows. Start with some a^i and compute $y^i = a^i x$ directly in at most N multiply–add pairs. The number of multiply–add pairs may be less than N if one or more entries of a^i are zero. Then, if a^j is a nearest neighbor of a^i in the forest, use the relationship between a^j and a^i to compute $y^j = a^j x$. After this, take a nearest neighbor of a^j, and continue until all the entries of y have been computed.

Additional improvements may be obtained by recognizing that the input tensor $G_T \leftrightarrow x$ is symmetric for certain operators like the Laplacian. In two spatial dimensions, G_T for the Laplacian is 2×2 with only 3 unique entries, and in three spatial dimensions it is 3×3 with only 6 unique entries. This fact may be used to construct a modified reference tensor A^0 with fewer columns. For other operators, it might have symmetry along some but not all of the axes.

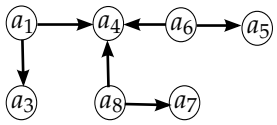
Heath and Wolf proposed a slight variation on this algorithm. Rather than picking an arbitrary starting row a^i, they enrich the graph with an extra vertex labeled *IP* for "inner product." Each a^i is a distance $N - z$ from IP, where z is the number of vanishing entries in a^i. The IP vertex is always selected as the root of the minimum spanning tree. It allows for a more robust treatment of unary relations such as sparsity, and detection of partial collinearity relations.

9.4 Geometric optimization

When relations between more than two rows are considered, the optimization problem may no longer be phrased in terms of a graph, but requires some other structure. In these cases, proving that one has found an optimal solution is typically difficult, and it is suspected that the associated combinatorial problems are *NP*-hard.

As a first attempt, one can work purely from linear dependencies among the data as follows. Let $B = \{b^i\}_i \subseteq \{a^i\}_{i=1}^N$ be a maximal set of nonzero rows of A, such that no two rows are collinear. Then

Figure 9.2: Generating graph for the vectors in (9.6).



enumerate all triples which are linearly dependent,

$$S = \left\{ \left\{ b^i, b^j, b^k \right\} \subseteq B : \exists\, \alpha_1, \alpha_2, \alpha_3 \neq 0 : \alpha_1 b^i + \alpha_2 b^j + \alpha_3 b^k = 0 \right\}. \tag{9.9}$$

The idea is now to identify some subset C of B that may be used to recursively construct the rest of the rows in B using the relationships in S.

Given some $C \subset B$, we may define the *closure* of C, denoted by $\bar{C}$, as follows. First of all, if $b \in C$, then $b \in \bar{C}$. Second, if $b \in B$ and there exist $c, d \in \bar{C}$ such that $\{b, c, d\} \in S$, then $b \in \bar{C}$ as well. If $\bar{C} = B$, we say that C is a *generator* for B or that C *generates* B.

The recursive definition suggests a greedy process for constructing the closure of any set C. Each vector in B is put in a priority queue with an initial value of the cost to compute independent of other vectors. While $C \neq B$, a vector from $B \backslash C$ with the minimum cost to compute is added to C and the priorities of B are updated according to S. This process constructs a directed, acyclic graph that indicates the linear dependence being used. Each $b \in C$ will have no out-neighbors, while each $b \in \bar{C} \backslash C$ will point to two other members of $\bar{C}$. This graph is called a *generating graph*. Using (9.6), we have the following sets B, S, and C, with the generating graph shown in Figure 9.2:

$$\begin{array}{rcl} B & = & \{a_1, a_3, a_4, a_5, a_6, a_7, a_8\} \\ S & = & \{(a_1, a_3, a_4), (a_4, a_5, a_6), (a_4, a_7, a_8)\} \\ C & = & \{a_3, a_4, a_5, a_7\} \end{array} \tag{9.10}$$

If C generates B, then the generating graph indicates an optimized (but perhaps not optimal) process for computing $\{y^i = b^i x\}_i$. Take a topological ordering of the vectors b^i according to this graph. Then, for each b^i in the topological ordering, if b^i has no out-neighbors, then $b^i x$ is computed explicitly. Otherwise, b^i will point to two other vectors b^j and b^k for which the dot products with x will already be known. Since the generating graph has been built from the set of linearly dependent triples S, there must exist some β_1, β_2 such that $b^i = \beta_1 b^j + \beta_2 b^k$. We may thus compute y^i by

$$y^i = b^i x = \beta_1 b^j x + \beta_2 b^k x, \tag{9.11}$$

which requires only two multiply–add pairs instead of N.

To make best use of the linear dependence information, one would like to find a generator C that has as few members as possible. We say that a generator C is *minimal* for B if no $C' \subset C$ also generates B. A stronger requirement is for a generator to be *minimum*. A generator C is minimum if no other generator C' has lower cardinality. More complete details and heuristics for constructing minimal generators are considered in Kirby and Scott (2007); it is not currently known whether such heuristics construct minimum generators or how hard the problem of finding minimum generators is.

Given a minimal generator C for B, one may consider searching for higher order linear relations among the elements of C, such as sets of four items that have a three-dimensional span. The discussion of generating graphs and their utilization is the same in this case.

In Wolf and Heath (2009), a combination of the binary and higher-order relations between the rows of A in a hypergraph model is studied. While greedy algorithms provide optimal solutions for a graph model, it is demonstrated that the obvious generalizations to hypergraphs can be suboptimal. While the hypergraph problems are most likely very hard, heuristics perform well and provide additional

triangles

degree	M	N	MN	$MAPs$
1	6	3	18	9
2	21	3	63	17
3	55	3	165	46

tetrahedra

degree	M	N	MN	$MAPs$
1	10	6	60	27
2	55	6	330	101
3	210	6	1260	370

Table 9.1: Number of multiply–add pairs for graph-optimized Laplace operator (MAPS) compared to the basic number of multiply–add pairs (MN).

degree	topological	geometric
2	101	105
3	370	327
4	1118	1072

Table 9.2: Comparison of topological and geometric optimizations for the Laplace operator on tetrahedra using polynomial degrees two through four. In each case, the final number of MAPs for the optimized algorithm is reported. The case $q = 1$ is not reported since then both strategies yield the same number of operations.

optimizations beyond the graph models. So, even if a non-optimal solution is found, it still provides an improved reduction in arithmetic requirements.

In Table 9.2, topological and geometric optimization are compared for the Laplacian using quadratic through quartic polynomials on tetrahedra. In the geometric case, the vectors a^i were filtered for unique direction; that is, only one vector for each class of collinear vectors was retained. Then, a generating graph was constructed for the remaining vectors using pairwise linear dependence. The generator for this set was then searched for linear dependence among sets of four vectors, and a generating graph constructed. Perhaps surprisingly, the geometric optimization found flop reductions comparable to or better than graph-based binary relations. These are shown in Table 9.2.

9.5 Optimization by dense linear algebra

As an alternative to optimizations that try to find a reduced arithmetic for computing the element tensor A_T, one may consider computing the element tensor by efficient dense linear algebra. As above, we note that the entries of the element tensor A_T may be computed by the matrix–vector product $\tilde{A}^0 \tilde{g}_T$. Although zeros may appear in $\tilde{A}^0$, this is typically a dense matrix and so the matrix–vector product may be computed efficiently with Level 2 BLAS, in particular using a call to `dgemv`. There exist a number of optimized implementations of BLAS, including hand-optimized vendor implementations, empirically and automatically tuned libraries (Whaley et al., 2001) and formal methods for automatic derivation of algorithms (Bientinesi et al., 2005).

The computation of the element tensor A_T may be optimized further by recognizing that one may

compute the element tensors for a batch of elements $\{T_i\}_i \subset \mathcal{T}$ in one matrix–matrix multiplication:

$$\begin{bmatrix} \tilde{A}^0 \tilde{g}_{T_1} & \tilde{A}^0 \tilde{g}_{T_2} & \cdots \end{bmatrix} = \tilde{A}^0 \begin{bmatrix} \tilde{g}_{T_1} & \tilde{g}_{T_2} & \cdots \end{bmatrix}. \tag{9.12}$$

This matrix–matrix product may be computed efficiently using a single Level 3 BLAS call (`dgemm`) instead of a sequence of Level 2 BLAS calls, and typically leads to better floating-point performance.

9.6 Notes on implementation

A subset of the optimizations discussed in this chapter are available as part of the FErari Python module. FErari (0.2.0) implements optimization based on finding binary relations between the entries of the element tensor. With optimizations turned on, FFC calls FErari at compile-time to generate optimized code. Optimization for FFC can be turned on either by the `-O` parameter when FFC is called from the command-line, or by setting `parameters["form_compiler"]["optimize"] = True` when FFC is called as a just-in-time compiler from the DOLFIN Python interface. Note that the FErari optimizations are only used when FFC generates code based on the tensor representation described in Chapter 8. When FFC generates code based on quadrature, optimization is handled differently, as described in Chapter 7. Improved run-times for several problems are detailed in Kirby and Logg (2008).

Part II

Implementation

10 DOLFIN: A C++/Python finite element library

By Anders Logg, Garth N. Wells and Johan Hake

DOLFIN is a C++/Python library that functions as the main user interface of FEniCS. In this chapter, we review the functionality of DOLFIN. We also discuss the implementation of some key features of DOLFIN in detail. For a general discussion on the design and implementation of DOLFIN, we refer to Logg and Wells (2010).

10.1 Overview

A large part of the functionality of FEniCS is implemented as part of DOLFIN. It provides a problem solving environment for models based on partial differential equations and implements core parts of the functionality of FEniCS, including data structures and algorithms for computational meshes and finite element assembly. To provide a simple and consistent user interface, DOLFIN wraps the functionality of other FEniCS components and external software, and handles the communication between these components.

Figure 10.1 presents an overview of the relationships between the components of FEniCS and external software. The software map presented in the figure shows a user application implemented on top of the DOLFIN user interface, either in C++ or in Python. User applications may also be developed using FEniCS Apps, a collection of solvers implemented on top of FEniCS/DOLFIN. DOLFIN itself functions as both a user interface and a core component of FEniCS. All communication between a user program, other core components of FEniCS and external software is routed through wrapper layers that are implemented as part of the DOLFIN user interface. In particular, variational forms expressed in the UFL form language (Chapter 17) are passed to the form compiler FFC (Chapter 11) or SFC (Chapter 15) to generate UFC code (Chapter 16), which can then be used by DOLFIN to assemble linear systems. In the case of FFC, this code generation depends on the finite element backend FIAT (Chapter 13), the just-in-time compilation utility Instant (Chapter 14) and the optional optimizing backend FErari (Chapter 12). Finally, the plotting capabilities provided by DOLFIN are implemented by Viper. Some of this communication is exposed to users of the DOLFIN C++ interface, which requires a user to explicitly generate UFC code from a UFL form file by calling a form compiler on the command-line.

DOLFIN also relies on external software for important functionality such as the linear algebra libraries PETSc, Trilinos, uBLAS and MTL4, and the mesh partitioning libraries ParMETIS and SCOTCH (Pellegrini).

10.2 User interfaces

DOLFIN provides two user interfaces. One interface is implemented as a traditional C++ library, and another interface is implemented as a standard Python module. The two interfaces are near-identical,

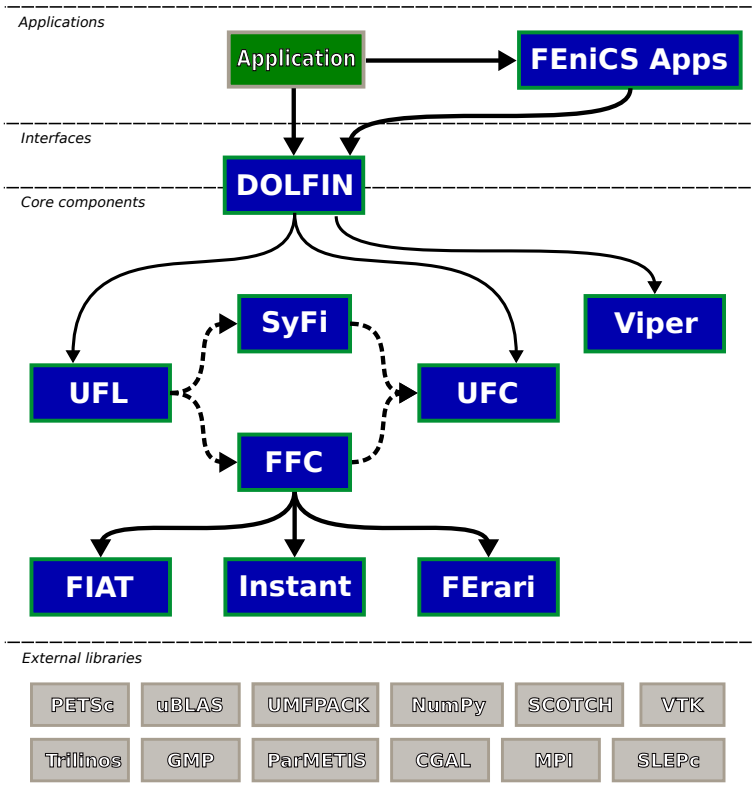


Figure 10.1: DOLFIN functions as the main user interface of FEniCS and handles the communication between the various components of FEniCS and external software. Solid lines indicate dependencies and dashed lines indicate data flow.

but in some cases particular language features of either C++ or Python require variations in the interfaces. In particular, the Python interface adds an additional level of automation by employing run-time (just-in-time) code generation. Below, we comment on the design and implementation of the two user interfaces of DOLFIN.

10.2.1 C++ interface

The DOLFIN C++ interface is designed as a standard object-oriented C++ library. It provides classes such as `Matrix`, `Vector`, `Mesh`, `FiniteElement`, `FunctionSpace` and `Function`, which model important concepts for finite element computing (see Figure 10.2). It also provides a small number of free functions (a function that is not a member function of a class), most notably `assemble` and `solve`, which can be used in conjunction with DOLFIN class objects to implement finite element solvers. The interface is designed to be as simple as possible, and without compromising on generality. When external software is wrapped, a simple and consistent user interface is provided to allow the rapid development of solvers without needing to deal with differences in the interfaces of external libraries. However, DOLFIN has been designed to interact flexibly with external software. In particular, in cases where DOLFIN provides wrappers for external libraries, such as the `Matrix` and `Vector` classes which wrap data structures from linear algebra libraries like PETSc and Trilinos, advanced users may, if necessary, access the underlying data structures in order to use native functionality from the wrapped external libraries.

To solve partial differential equations using the DOLFIN C++ interface, users must express finite element variational problems in the UFL form language. This is accomplished by entering the forms into separate `.ufl` files and compiling those files using a form compiler to generate UFC-compliant C++ code. The generated code may then be included in a DOLFIN C++ program. We return to this

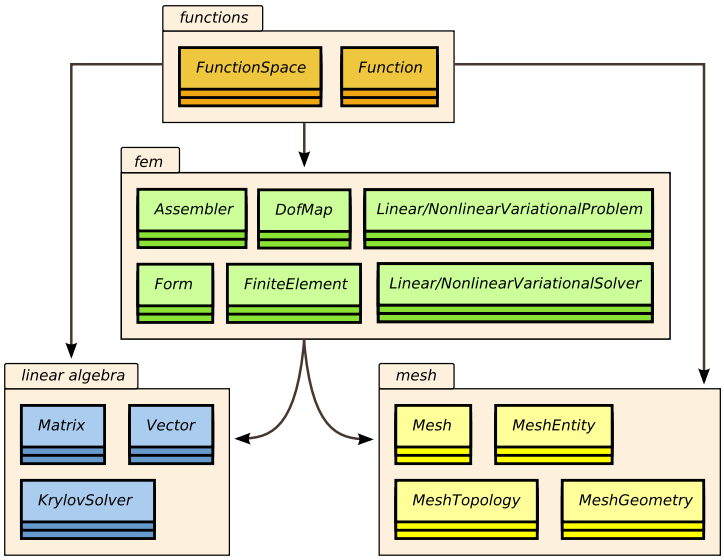


Figure 10.2: Schematic overview of some of the most important components and classes of DOLFIN. Arrows indicate dependencies.

issue in Section 10.3.

To use DOLFIN from C++, users need to include one or more header files from the DOLFIN C++ library. In the simplest case, one includes the header file `dolfin.h`, which in turn includes all other DOLFIN header files:

C++ code

```
#include <dolfin.h>

using namespace dolfin;

int main()
{

  return 0;
}
```

10.2.2 Python interface

Over the last decade, Python has emerged as an attractive choice for the rapid development of simulation codes for scientific computing. Python brings the benefits of a high-level scripting language, the strength of an object-oriented language and a wealth of libraries for numerical computation.

The bulk of the DOLFIN Python interface is automatically generated from the C++ interface using SWIG (Beazley, 1996; SWIG). Since the functionality of both the C++ and Python interfaces are implemented as part of the DOLFIN C++ library, DOLFIN is equally efficient via the C++ and Python interfaces for most operations.

The DOLFIN Python interface offers some functionality that is not available from the C++ interface. In particular, the UFL form language is seamlessly integrated into the Python interface and code generation is automatically handled at run-time. To use DOLFIN from Python, users need to import functionality from the DOLFIN Python module. In the simplest case, one includes all functionality from the Python module named `dolfin`:

Python code

```
from dolfin import *
```

10.3 Functionality

DOLFIN is organized as a collection of libraries (modules), with each covering a certain area of functionality. We review here these areas and explain the purpose and usage of the most commonly used classes and functions. The review is bottom-up; that is, we start by describing the core low-level functionality of DOLFIN (linear algebra and meshes) and then move upwards to describe higher level functionality. For further details, we refer to the DOLFIN Programmer's Reference on the FEniCS Project web page and to Logg and Wells (2010).

10.3.1 Linear algebra

DOLFIN provides a range of linear algebra objects and functionality, including vectors, dense and sparse matrices, direct and iterative linear solvers and eigenvalues solvers, and does so via a simple and consistent interface. For the bulk of underlying functionality, DOLFIN relies on third-party libraries such as PETSc and Trilinos. DOLFIN defines the abstract base classes `GenericTensor`, `GenericMatrix` and `GenericVector`, and these are used extensively throughout the library. Implementations of these generic interfaces for a number of backends are provided in DOLFIN, thereby achieving a common interface for different backends. Users can also wrap other linear algebra backends by implementing the generic interfaces.

Matrices and vectors. The simplest way to create matrices and vectors is via the classes `Matrix` and `Vector`. In general, `Matrix` and `Vector` represent distributed linear algebra objects that may be stored across (MPI) processes when running in parallel. Consistent with the most common usage in a finite element library, a `Vector` uses dense storage and a `Matrix` uses sparse storage. A `Vector` can be created as follows:

C++ code

```
Vector x;
```

Python code

```
x = Vector()
```

and a matrix can be created by:

C++ code

```
Matrix A;
```

Python code

```
A = Matrix()
```

In most applications, a user may need to create a matrix or a vector, but most operations on the linear algebra objects, including resizing, will take place inside the library and a user will not have to operate on the objects directly.

The following code illustrates how to create a vector of size 100:

C++ code

```
Vector x(100);
```

Python code

```
x = Vector(100)
```

A number of backends support distributed linear algebra for parallel computation, in which case the vector `x` will have global size 100, and DOLFIN will partition the vector across processes in (near) equal-sized portions.

Creating a `Matrix` of a given size is more involved as the matrix is sparse and in general needs to be initialized (data structures allocated) based on the structure of the sparse matrix (its sparsity pattern). Initialization of sparse matrices is handled by DOLFIN when required.

While DOLFIN supports distributed linear algebra objects for parallel computation, it is rare that a user is exposed to details at the level of parallel data layouts. The distribution of objects across processes is handled automatically by the library.

Solving linear systems. The simplest approach to solving the linear system $Ax = b$ is to use

C++ code

```
solve(A, x, b);
```

Python code

```
solve(A, x, b)
```

DOLFIN will use a default method to solve the system of equations. Optional arguments may be given to specify which algorithm to use when solving the linear system and, in the case of an iterative method, which preconditioner to use:

C++ code

```
solve(A, x, b, "lu");
solve(A, x, b, "gmres", "ilu");
```

Python code

```
solve(A, x, b, "lu");
solve(A, x, b, "gmres", "ilu")
```

Which methods and preconditioners that are available depends on which linear algebra backend DOLFIN has been configured with. To list the available solver methods and preconditioners, the following commands may be used:

C++ code

```
list_lu_solver_methods();
list_krylov_solver_methods();
list_krylov_solver_preconditioners();
```

Python code

```
list_lu_solver_methods()
list_krylov_solver_methods()
list_krylov_solver_preconditioners()
```

Using the function `solve` is straightforward, but it offers little control over details of the solution process. For many applications, it is desirable to exercise a degree of control over the solution process and reuse solver objects throughout a simulation.

The linear system $Ax = b$ can be solved using LU decomposition (a direct method) as follows:

C++ code

```
LUSolver solver(A);
solver.solve(x, b);
```

Python code

```
solver = LUSolver(A)
solver.solve(x, b)
```

Alternatively, the operator A associated with the linear solver can be set post-construction:

C++ code

```
LUSolver solver;
solver.set_operator(A);
solver.solve(x, b);
```

Python code

```
solver = LUSolver()
solver.set_operator(A)
solver.solve(x, b)
```

This can be useful when passing a linear solver via a function interface and setting the operator inside a function.

In some cases, the system $Ax = b$ may be solved a number of times for a given matrix A and different vectors b, or for different A but with the same nonzero structure. If the nonzero structure of A does not change, then some efficiency gains for repeated solves can be achieved by informing the LU solver of this fact:

C++ code

```
solver.parameters["same_nonzero_pattern"] = true;
```

Python code

```
solver.parameters["same_nonzero_pattern"] = True
```

In the case that A does not change, the solution time for subsequent solves can be reduced dramatically by re-using the LU factorization of A. Re-use of the factorization is controlled by the parameter `"reuse_factorization"`.

It is possible for some backends to prescribe the specific LU solver to be used. This depends on the backend, which solvers that have been configured by DOLFIN and how third-party linear algebra backends have been configured.

The system of equations $Ax = b$ can be solved using a preconditioned Krylov solver by:

C++ code

```
KrylovSolver solver(A);
solver.solve(x, b);
```

Python code

```
solver = KrylovSolver(A)
solver.solve(x, b)
```

The above will use a default preconditioner and solver, and default parameters. If a `KrylovSolver` is constructed without a matrix operator A, the operator can be set post-construction:

C++ code

```
KrylovSolver solver;
solver.set_operator(A);
solver.solve(x, b);
```

Python code

```
solver = KrylovSolver()
solver.set_operator(A)
solver.solve(x, b)
```

In some cases, it may be useful to use a preconditioner matrix P that differs from A:

C++ code

```
KrylovSolver solver;
solver.set_operators(A, P);
solver.solve(x, b);
```

Python code

```
solver = KrylovSolver()
solver.set_operators(A, P)
solver.solve(x, b)
```

Various parameters for Krylov solvers can be set. Some common parameters are:

Python code

```
solver = KrylovSolver()
solver.parameters["relative_tolerance"]      = 1.0e-6
solver.parameters["absolute_tolerance"]      = 1.0e-15
solver.parameters["divergence_limit"]        = 1.0e4
solver.parameters["maximum_iterations"]      = 10000
solver.parameters["error_on_nonconvergence"] = True
solver.parameters["nonzero_initial_guess"]   = False
```

The parameters may be set similarly from C++. Printing a summary of the convergence of a `KrylovSolver` and printing details of the convergence history can be controlled via parameters:

C++ code

```
KrylovSolver solver;
solver.parameters["report"] = true;
solver.parameters["monitor_convergence"] = true;
```

Python code

```
solver = KrylovSolver()
solver.parameters["report"] = True
solver.parameters["monitor_convergence"] = True
```

The specific Krylov solver and preconditioner to be used can be set at construction of a solver object. The simplest approach is to set the Krylov method and the preconditioner via string descriptions. For example:

C++ code

```
KrylovSolver solver("gmres", "ilu");
```

Python code

```
solver = KrylovSolver("gmres", "ilu")
```

The above specifies the Generalized Minimum Residual (GMRES) method as a solver, and incomplete LU (ILU) preconditioning.

When backends such as PETSc and Trilinos are configured, a wide range of Krylov methods and preconditioners can be applied, and a large number of solver and preconditioner parameters can be set. In addition to what is described here, DOLFIN provides more advanced interfaces which permit finer control of the solution process. It is also possible for users to provide their own preconditioners.

Solving eigenvalue problems. DOLFIN uses the library SLEPc, which builds on PETSc, to solve eigenvalue problems. The SLEPc interface works only with PETSc-based linear algebra objects. Therefore, it is necessary to use PETSc-based objects, or to set the default linear algebra backend to PETSc and downcast objects (as explained in the next section). The following code illustrates the solution of the eigenvalue problem $Ax = \lambda x$:

C++ code

```
// Create matrix
PETScMatrix A;

// Code omitted for setting the entries of A

// Create eigensolver
SLEPcEigenSolver eigensolver(A);

// Compute all eigenvalues of A
eigensolver.solve();

// Get first eigenpair
double lambda_real, lambda_complex;
PETScVector x_real, x_complex;
eigensolver.get_eigenpair(lambda_real, lambda_complex, x_real, x_complex, 0);
```

Python code

```
# Create matrix
A = PETScMatrix()

# Code omitted for setting the entries of A

# Create eigensolver
eigensolver = SLEPcEigenSolver(A)

# Compute all eigenvalues of A
eigensolver.solve()

# Get first eigenpair
lambda_r, lambda_c, x_real, x_complex = eigensolver.get_eigenpair(0)
```

The real and complex components of the eigenvalue are returned in `lambda_real` and `lambda_complex`, respectively, and the real and complex components of the eigenvector are returned in `x_real` and `x_complex`, respectively.

To create a solver for the generalized eigenvalue problem $Ax = \lambda Mx$, the eigensolver can be constructed using A and M:

C++ code

```
PETScMatrix A;
PETScMatrix M;

// Code omitted for setting the entries of A and M

SLEPcEigenSolver eigensolver(A, M);
```

Python code

```
A = PETScMatrix()
M = PETScMatrix()

# Code omitted for setting the entries of A and M

eigensolver = SLEPcEigenSolver(A, M)
```

There are many options that a user can set via the parameter system to control the eigenvalue problem solution process. To print a list of available parameters, call `info(eigensolver.parameters, true)` and `info(eigensolver.parameters, True)` from C++ and Python, respectively.

Selecting a linear algebra backend. The `Matrix`, `Vector`, `LUSolver` and `KrylovSolver` objects are all based on a specific linear algebra backend. The default backend depends on which backends are enabled when DOLFIN is configured. The backend can be set via the global parameter `"linear_algebra_backend"`. To use PETSc as the linear algebra backend:

C++ code

```
parameters["linear_algebra_backend"] = "PETSc";
```

Python code

```
parameters["linear_algebra_backend"] = "PETSc"
```

This parameter should be set before creating linear algebra objects. To use Epetra from the Trilinos collection, the parameter `"linear_algebra_backend"` should be set to `"Epetra"`. For uBLAS, the parameter should be set to `"uBLAS"` and for MTL4, the parameter should be set to `"MTL4"`.

Users can explicitly create linear algebra objects that use a particular backend. Generally, such objects are prefixed with the name of the backend. For example, a PETSc-based vector and LU solver are created by:

C++ code

```
PETScVector x;
PETScLUSolver solver;
```

Python code

```
x = PETScVector()
solver = PETScLUSolver()
```

Solving nonlinear systems. DOLFIN provides a Newton solver in the form of the class `NewtonSolver` for solving nonlinear systems of equations of the form

$$F(x) = 0, \tag{10.1}$$

where $x \in \mathbb{R}^n$ and $F : \mathbb{R}^n \to \mathbb{R}^n$. To solve such a problem using the DOLFIN Newton solver, a user needs to provide a subclass of `NonlinearProblem`. The purpose of a `NonlinearProblem` object is to evaluate F and the Jacobian of F, which will be denoted by $J : \mathbb{R}^n \to \mathbb{R}^n \times \mathbb{R}^n$. An outline of a user-provided `MyNonlinearProblem` class for solving a nonlinear differential equation is shown below.

C++ code

```
class MyNonlinearProblem : public NonlinearProblem
{
public:

  // Constructor
  MyNonlinearProblem(const Form& L, const Form& a,
                     const BoundaryCondition& bc) : L(L), a(a), bc(bc) {}

  // User-defined residual vector F
  void F(GenericVector& b, const GenericVector& x)
  {
    assemble(b, L);
    bc.apply(b, x);
  }

  // User-defined Jacobian matrix J
  void J(GenericMatrix& A, const GenericVector& x)
  {
    assemble(A, a);
    bc.apply(A);
  }

private:

  const Form& L;
  const Form& a;
  const BoundaryCondition& bc;

};
```

A `MyNonlinearProblem` object is constructed using a linear form `L`, that when assembled corresponds to F, and a bilinear form `a`, that when assembled corresponds to J. The classes `Form` and `BoundaryCondition` used in the example are discussed in more detail later. The same `MyNonlinearProblem` class can be defined in Python:

Python code

```
class MyNonlinearProblem(NonlinearProblem):
    def __init__(self, L, a, bc):
        NonlinearProblem.__init__(self)
        self.L = L
        self.a = a
        self.bc = bc
    def F(self, b, x):
        assemble(self.L, tensor=b)
        self.bc.apply(b, x)
    def J(self, A, x):
        assemble(self.a, tensor=A)
        self.bc.apply(A)
```

Once a nonlinear problem class has been defined, a `NewtonSolver` object can be created and the Newton solver can be used to compute the solution vector x to the nonlinear problem:

C++ code

```
MyNonlinearProblem problem(L, a, bc);
NewtonSolver newton_solver;

newton_solver.solve(problem, u.vector());
```

Python code

```
problem = MyNonlinearProblem(L, a, bc)
newton_solver = NewtonSolver()

newton_solver.solve(problem, u.vector())
```

A number of parameters can be set for a `NewtonSolver`. Some parameters that determine the behavior of the Newton solver are:

Python code

```
newton_solver = NewtonSolver()
newton_solver.parameters["maximum_iterations"] = 20
newton_solver.parameters["relative_tolerance"] = 1.0e-6
newton_solver.parameters["absolute_tolerance"] = 1.0e-10
newton_solver.parameters["error_on_nonconvergence"] = False
```

The parameters may be set similarly from C++. When testing for convergence, usually a norm of the residual F is checked. Sometimes it is useful instead to check a norm of the iterative correction dx. This is controlled by the parameter `"convergence_criterion"`, which can be set to `"residual"`, for checking the size of the residual F, or `"incremental"`, for checking the size of the increment dx.

For more advanced usage, a `NewtonSolver` can be constructed with arguments that specify the linear solver and preconditioner to be used in the solution process.

10.3.2 Meshes

A central part of DOLFIN is its mesh library and the `Mesh` class. The mesh library provides data structures and algorithms for computational meshes, including the computation of mesh connectivity (incidence relations), mesh refinement, mesh partitioning and mesh intersection.

The mesh library is implemented in C++ and has been optimized to minimize storage requirements and to enable efficient access to mesh data. In particular, a DOLFIN mesh is stored in a small number of contiguous arrays, on top of which a light-weight object-oriented layer provides a *view* to the underlying data. For a detailed discussion on the design and implementation of the mesh library, we refer to Logg (2009).

Creating a mesh. DOLFIN provides functionality for creating simple meshes, such as meshes of unit squares and unit cubes, spheres, rectangles and boxes. The following code demonstrates how to create a 16×16 triangular mesh of the unit square (consisting of $2 \times 16 \times 16 = 512$ triangles) and a $16 \times 16 \times 16$ tetrahedral mesh of the unit cube (consisting of $6 \times 16 \times 16 \times 16 = 24,576$ tetrahedra).

C++ code

```
UnitSquare unit_square(16, 16);
UnitCube unit_cube(16, 16, 16);
```

Python code

```
unit_square = UnitSquare(16, 16)
unit_cube = UnitCube(16, 16, 16)
```

Simplicial meshes (meshes consisting of intervals, triangles or tetrahedra) may be constructed explicitly by specifying the cells and vertices of the mesh. An interface for creating simplicial meshes is provided by the class `MeshEditor`. The following code demonstrates how to create a mesh consisting of two triangles covering the unit square:

C++ code

```
Mesh mesh;
MeshEditor editor;
editor.open(mesh, 2, 2);
editor.init_vertices(4);
editor.init_cells(2);
editor.add_vertex(0, 0.0, 0.0);
editor.add_vertex(1, 1.0, 0.0);
editor.add_vertex(2, 1.0, 1.0);
editor.add_vertex(3, 0.0, 1.0);
editor.add_cell(0, 0, 1, 2);
editor.add_cell(1, 0, 2, 3);
editor.close();
```

Python code

```
mesh = Mesh();
editor = MeshEditor();
editor.open(mesh, 2, 2)
editor.init_vertices(4)
editor.init_cells(2)
editor.add_vertex(0, 0.0, 0.0)
editor.add_vertex(1, 1.0, 0.0)
editor.add_vertex(2, 1.0, 1.0)
editor.add_vertex(3, 0.0, 1.0)
editor.add_cell(0, 0, 1, 2)
editor.add_cell(1, 0, 2, 3)
editor.close()
```

Reading a mesh from file. Although the built-in classes `UnitSquare` and `UnitCube` are useful for testing, a typical application will need to read from file a mesh that has been generated by an external mesh generator. To read a mesh from file, simply supply the filename to the constructor of the `Mesh` class:

C++ code

```
Mesh mesh("mesh.xml");
```

Python code

```
mesh = Mesh("mesh.xml")
```

Meshes must be stored in the DOLFIN XML format. The following example illustrates the XML format for a 2×2 mesh of the unit square:

XML code

```
<?xml version="1.0" encoding="UTF-8"?>

<dolfin xmlns:dolfin="http://fenicsproject.org">
```

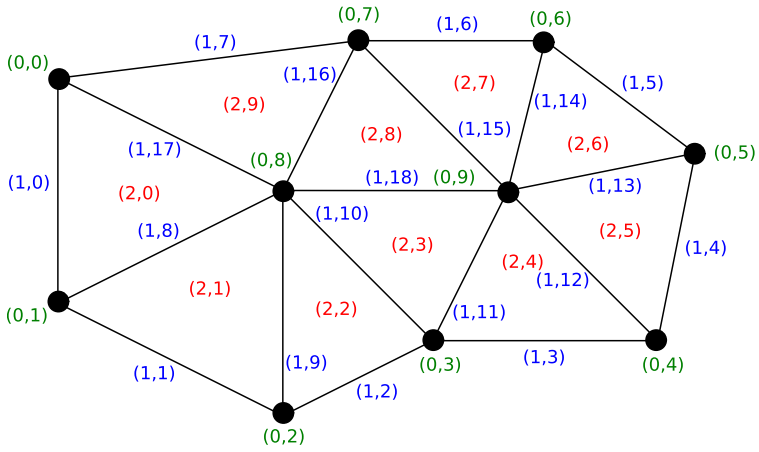


Figure 10.3: Each entity of a mesh is identified by a pair (d, i) which specifies the topological dimension d and a unique index i for the entity within the set of entities of dimension d.

```
  <mesh celltype="triangle" dim="2">
    <vertices size="9">
      <vertex index="0" x="0" y="0"/>
      <vertex index="1" x="0.5" y="0"/>
      <vertex index="2" x="1" y="0"/>
      <vertex index="3" x="0" y="0.5"/>
      <vertex index="4" x="0.5" y="0.5"/>
      <vertex index="5" x="1" y="0.5"/>
      <vertex index="6" x="0" y="1"/>
      <vertex index="7" x="0.5" y="1"/>
      <vertex index="8" x="1" y="1"/>
    </vertices>
    <cells size="8">
      <triangle index="0" v0="0" v1="1" v2="4"/>
      <triangle index="1" v0="0" v1="3" v2="4"/>
      <triangle index="2" v0="1" v1="2" v2="5"/>
      <triangle index="3" v0="1" v1="4" v2="5"/>
      <triangle index="4" v0="3" v1="4" v2="7"/>
      <triangle index="5" v0="3" v1="6" v2="7"/>
      <triangle index="6" v0="4" v1="5" v2="8"/>
      <triangle index="7" v0="4" v1="7" v2="8"/>
    </cells>
  </mesh>
</dolfin>
```

Meshes stored in other data formats may be converted to the DOLFIN XML format using the command `dolfin-convert`, as explained in more detail below.

Mesh entities. Conceptually, a *mesh* (modeled by the class `Mesh`), consists of a collection of *mesh entities*. A *mesh entity* is a pair (d, i), where d is the topological dimension of the mesh entity and i is a unique index of the mesh entity. Mesh entities are numbered within each topological dimension from 0 to $n_d - 1$, where n_d is the number of mesh entities of topological dimension d.

For convenience, mesh entities of topological dimension 0 are referred to as *vertices*, entities of dimension 1 as *edges*, entities of dimension 2 as *faces*. Entities of *codimension* 1 are referred to as *facets* and entities of codimension 0 as *cells*. These concepts are summarized in Figure 10.3 and Table 10.1. We note that a triangular mesh consists of vertices, edges and cells, and that the edges may alternatively be referred to as facets and the cells as faces. We further note that a tetrahedral mesh consists of vertices, edges, faces and cells, and that the faces may alternatively be referred to as facets. These concepts are implemented by the classes `MeshEntity`, `Vertex`, `Edge`, `Face`, `Facet` and

Entity	Dimension	Codimension
Vertex	0	D
Edge	1	$D-1$
Face	2	$D-2$
Facet	$D-1$	1
Cell	D	0

Table 10.1: Mesh entities and their dimensions/codimensions. The codimension of an entity is $D-d$ where D is the maximal dimension and d is the dimension.

`Cell`. These classes do not store any data. Instead, they are light-weight objects that provide views of the underlying mesh data. A `MeshEntity` may be created from a `Mesh`, a topological dimension and an index. The following code demonstrates how to create various entities on a mesh:

C++ code

```
MeshEntity entity(mesh, 0, 33); // vertex number 33
Vertex vertex(mesh, 33);        // vertex number 33
Cell cell(mesh, 25);            // cell number 25
```

Python code

```
entity = MeshEntity(mesh, 0, 33) # vertex number 33
vertex = Vertex(mesh, 33)        # vertex number 33
cell = Cell(mesh, 25)            # cell number 25
```

Mesh topology and geometry. The topology of a mesh is stored separately from its geometry. The topology of a mesh is a description of the relations between the various entities of the mesh, while the geometry describes how those entities are embedded in $\mathbb{R}^d$.

Users are rarely confronted with the `MeshTopology` and `MeshGeometry` classes directly since most algorithms on meshes can be expressed in terms of *mesh iterators*. However, users may sometimes need to access the dimension of a `Mesh`, which involves accessing either the `MeshTopology` or `MeshGeometry`, which are stored as part of the `Mesh`, as illustrated in the following code examples:

C++ code

```
uint gdim = mesh.topology().dim();
uint tdim = mesh.geometry().dim();
```

Python code

```
gdim = mesh.topology().dim()
tdim = mesh.geometry().dim()
```

It should be noted that the topological and geometric dimensions may differ. This is the case in particular for the boundary of a mesh, which is typically a mesh of topological dimension D embedded in $\mathbb{R}^{D+1}$. That is, the geometry dimension is $D+1$.

Mesh connectivity. The topology of a `Mesh` is represented by the *connectivity* (incidence relations) of the mesh, which is a complete description of which entities of the mesh are connected to which entities. Such connectivity is stored in DOLFIN by the `MeshConnectivity` class. One such data set is stored as

	0	1	2	3
0	–	×	–	×
1	×	×	–	–
2	–	–	–	–
3	×	×	–	×

Table 10.2: DOLFIN computes the connectivity $d \to d'$ of a mesh for any pair $d, d' = 0, 1, \ldots, D$. The table indicates which connectivity pairs (indicated by $\times$) have been computed in order to compute the connectivity $1 \to 1$ (edge–edge connectivity) for a tetrahedral mesh.

part of the class `MeshTopology` for each pair of topological dimensions $d \to d'$ for $d, d' = 0, 1, \ldots, D$, where D is the topological dimension.

When a `Mesh` is created, a minimal `MeshTopology` is created. Only the connectivity from cells (dimension D) to vertices (dimension 0) is stored (`MeshConnectivity` $D \to 0$). When a certain connectivity is requested, such as for example the connectivity $1 \to 1$ (connectivity from edges to edges), DOLFIN automatically computes any other connectivities required for computing the requested connectivity. This is illustrated in Table 10.2, where we indicate which connectivities are required to compute the $1 \to 1$ connectivity. The following code demonstrates how to initialize various kinds of mesh connectivity for a tetrahedral mesh ($D = 3$):

C++ code

```
mesh.init(2);    // Compute faces
mesh.init(0, 0); // Compute vertex neighbors for each vertex
mesh.init(1, 1); // Compute edge neighbors for each edge
```

Python code

```
mesh.init(2)      # Compute faces
mesh.init(0, 0)   # Compute vertex neighbors for each vertex
mesh.init(1, 1)   # Compute edge neighbors for each edge
```

Mesh iterators. Algorithms operating on a mesh can often be expressed in terms of *iterators*. The mesh library provides the general iterator `MeshEntityIterator` for iteration over mesh entities, as well as the specialized mesh iterators `VertexIterator`, `EdgeIterator`, `FaceIterator`, `FacetIterator` and `CellIterator`.

The following code illustrates how to iterate over all incident (connected) vertices of all vertices of all cells of a given mesh. Two vertices are considered as neighbors if they both belong to the same cell. For simplex meshes, this is equivalent to an edge connecting the two vertices.

C++ code

```
for (CellIterator c(mesh); !c.end(); ++c)
  for (VertexIterator v0(*c); !v0.end(); ++v0)
    for (VertexIterator v1(*v0); !v1.end(); ++v1)
      cout << *v1 << endl;
```

Python code

```
for c in cells(mesh):
    for v0 in vertices(c):
        for v1 in vertices(v0):
            print v1
```

This may alternatively be implemented using the general iterator `MeshEntityIterator` as follows:

C++ code

```
uint D = mesh.topology().dim();
for (MeshEntityIterator c(mesh, D); !c.end(); ++c)
  for (MeshEntityIterator v0(*c, 0); !v0.end(); ++v0)
    for (MeshEntityIterator v1(*v0, 0); !v1.end(); ++v1)
      cout << *v1 << endl;
```

Python code

```
D = mesh.topology().dim()
for c in entities(mesh, D):
    for v0 in entities(c, 0):
        for v1 in entities(v0, 0):
            print v1
```

Mesh functions. A useful class for storing data associated with a `Mesh` is the `MeshFunction` class. This makes it simple to store, for example, material parameters, subdomain indicators, refinement markers on the `Cells` of a `Mesh` or boundary markers on the `Facets` of a `Mesh`. A `MeshFunction` is a discrete function that takes a value on each mesh entity of a given topological dimension d. The number of values stored in a `MeshFunction` is equal to the number of entities n_d of dimension d. A `MeshFunction` is templated over the value type and may thus be used to store values of any type. For convenience, named `MeshFunctions` are provided by the classes `VertexFunction`, `EdgeFunction`, `FaceFunction`, `FacetFunction` and `CellFunction`. The following code illustrates how to create a pair of `MeshFunctions`, one for storing subdomain indicators on `Cells` and one for storing boundary markers on `Facets`:

C++ code

```
CellFunction<uint> sub_domains(mesh);
sub_domains.set_all(0);
for (CellIterator cell(mesh); !cell.end(); ++cell)
{
  Point p = cell.midpoint();
  if (p.x() > 0.5)
    sub_domains[cell] = 1;
}

FacetFunction<uint> boundary_markers(mesh);
boundary_markers.set_all(0);
for (FacetIterator facet(mesh); !facet.end(); ++facet)
{
  Point p = facet.midpoint();
  if (near(p.y(), 0.0) || near(p.y(), 1.0))
    boundary_markers[facet] = 1;
}
```

Python code

```
sub_domains = CellFunction("uint", mesh)
sub_domains.set_all(0)
for cell in cells(mesh):
    p = cell.midpoint()
    if p.x() > 0.5:
        sub_domains[cell] = 1
```

```
boundary_markers = FacetFunction("uint", mesh)
boundary_markers.set_all(0)
for facet in facets(mesh):
    p = facet.midpoint()
    if near(p.y(), 0.0) or near(p.y(), 1.0):
        boundary_markers[facet] = 1
```

Mesh data. The `MeshData` class provides a simple way to associate data with a `Mesh`. It allows arbitrary `MeshFunctions` (and other quantities) to be associated with a `Mesh`. The following code illustrates how to attach and retrieve a `MeshFunction` named `"sub_domains"` to/from a `Mesh`:

C++ code

```
MeshFunction<uint>* sub_domains = mesh.data().create_mesh_function("sub_domains");
sub_domains = mesh.data().mesh_function("sub_domains");
```

Python code

```
sub_domains = mesh.data().create_mesh_function("sub_domains")
sub_domains = mesh.data().mesh_function("sub_domains")
```

To list data associated with a given `Mesh`, issue the command `info(mesh.data(), true)` in C++ or `info(mesh.data(), True)` in Python.

Mesh refinement. A `Mesh` may be refined, by either uniform or local refinement, by calling the `refine` function, as illustrated in the code examples below.

C++ code

```
// Uniform refinement
mesh = refine(mesh);

// Local refinement
CellFunction<bool> cell_markers(mesh);
cell_markers.set_all(false);
Point origin(0.0, 0.0, 0.0);
for (CellIterator cell(mesh); !cell.end(); ++cell)
{
  Point p = cell.midpoint();
  if (p.distance(origin) < 0.1)
    cell_markers[cell] = true;
}
mesh = refine(mesh, cell_markers);
```

Python code

```
# Uniform refinement
mesh = refine(mesh)

# Local refinement
cell_markers = CellFunction("bool", mesh)
cell_markers.set_all(False)
origin = Point(0.0, 0.0, 0.0)
for cell in cells(mesh):
    p = cell.midpoint()
    if p.distance(origin) < 0.1:
        cell_markers[cell] = True
mesh = refine(mesh, cell_markers)
```

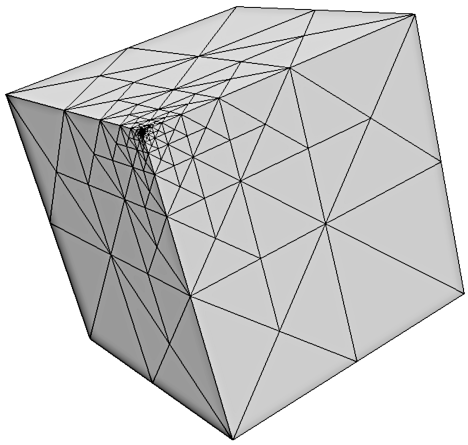


Figure 10.4: A locally refined mesh obtained by repeated marking of the cells close to one of the corners of the unit cube.

Currently, local refinement defaults to recursive refinement by edge bisection (Rivara, 1984, 1992). An example of a locally refined mesh obtained by a repeated marking of the cells close to one of the corners of the unit cube is shown in Figure 10.4.

Parallel meshes. When running a program in parallel on a distributed memory architecture (using MPI by invoking the program with the `mpirun` wrapper), DOLFIN automatically partitions and distributes meshes. Each process then stores a portion of the global mesh as a standard `Mesh` object. In addition, it stores auxiliary data needed for correctly computing local-to-global maps on each process and for communicating data to neighboring regions. Parallel computing with DOLFIN is discussed in Section 10.4.

10.3.3 Finite elements

The concept of a finite element as discussed in Chapters 2 and 3 (the Ciarlet definition) is implemented by the DOLFIN `FiniteElement` class. This class is implemented differently in the C++ and Python interfaces.

The C++ implementation of the `FiniteElement` class relies on code generated by a form compiler such as FFC or SFC, which are discussed in Chapters 11 and 15, respectively. The class `FiniteElement` is essentially a wrapper class for the UFC class `ufc::finite_element`. A C++ `FiniteElement` provides all the functionality of a `ufc::finite_element`. Users of the DOLFIN C++ interface will typically not use the `FiniteElement` class directly, but it is an important building block for the `FunctionSpace` class, which is discussed below. However, users developing advanced algorithms that require run-time evaluation of finite element basis function will need to familiarize themselves with the `FiniteElement` interface. For details, we refer to the DOLFIN Programmer's Reference.

The Python interface also provides a `FiniteElement` class. The Python `FiniteElement` class is imported directly from the UFL Python module (see Chapter 17). As such, it is just a label for a particular finite element that can be used to define variational problems. Variational problems are more conveniently defined in terms of the DOLFIN `FunctionSpace` class, so users of the Python interface are rarely confronted with the `FiniteElement` class. However, advanced users who wish to develop algorithms in Python that require functionality defined in the UFC interface, such as run-time evaluation of basis functions, can access such functionality by explicitly generating code from within the Python interface. This can be accomplished by a call to the DOLFIN `jit` function (just-in-time compilation), which takes as input a UFL `FiniteElement` and returns a pair containing a `ufc::finite_element` and a `ufc::dofmap`. The returned objects are created by first generating the

Name	Symbol
Argyris	*ARG*
Arnold–Winther	*AW*
Brezzi–Douglas–Marini	BDM
Crouzeix–Raviart	CR
Discontinuous Lagrange	DG
Hermite	*HER*
Lagrange	CG
Mardal–Tai–Winther	*MTW*
Morley	*MOR*
Nédélec 1st kind $H(\text{curl})$	N1curl
Nédélec 2nd kind $H(\text{curl})$	N2curl
Raviart–Thomas	RT

Table 10.3: List of finite elements supported by DOLFIN 1.0. Elements in grey italics are partly supported in FEniCS but not throughout the entire toolchain.

corresponding C++ code, then compiling and wrapping that C++ code into a Python module. The returned objects are therefore directly usable from within Python.

The degrees of freedom of a `FiniteElement` can be plotted directly from the Python interface by a call to `plot(element)`. This will draw a picture of the shape of the finite element, along with a graphical representation of its degrees of freedom in accordance with the notation described in Chapter 3.

Table 10.3 lists the finite elements currently supported by DOLFIN (and the toolchain FIAT–UFL–FFC/SFC–UFC). A `FiniteElement` may be specified (from Python) using either its full name or its short symbol, as illustrated in the code example below:

UFL code

```
element = FiniteElement("Lagrange", tetrahedron, 5)
element = FiniteElement("CG", tetrahedron, 5)

element = FiniteElement("Brezzi-Douglas-Marini", triangle, 3)
element = FiniteElement("BDM", triangle, 3)

element = FiniteElement("Nedelec 1st kind H(curl)", tetrahedron, 2)
element = FiniteElement("N1curl", tetrahedron, 2)
```

10.3.4 Function spaces

The DOLFIN `FunctionSpace` class represents a finite element function space V_h, as defined in Chapter 2. The data of a `FunctionSpace` is represented in terms of a triplet consisting of a `Mesh`, a `DofMap` and a `FiniteElement`:

$$\texttt{FunctionSpace} = (\texttt{Mesh}, \texttt{DofMap}, \texttt{FiniteElement}).$$

The `Mesh` defines the computational domain and its discretization. The `DofMap` defines how the degrees of freedom of the function space are distributed. In particular, the `DofMap` provides the function `tabulate_dofs` which maps the local degrees of freedom on any given cell of the `Mesh` to global degrees of freedom. The `DofMap` plays a role in defining the global regularity of the finite element function space. The `FiniteElement` defines the local function space on any given cell of the `Mesh`.

Note that if two or more `FunctionSpaces` are created on the same `Mesh`, that `Mesh` is shared between the two `FunctionSpaces`.

Creating function spaces. As for the `FiniteElement` class, `FunctionSpaces` are handled differently in the C++ and Python interfaces. In C++, the instantiation of a `FunctionSpace` relies on generated code. As an example, we consider here the creation of a `FunctionSpace` representing continuous piecewise linear Lagrange polynomials on triangles. First, the corresponding finite element must be defined in the UFL form language. We do this by entering the following code into a file named `Lagrange.ufl`:

UFL code

```
element = FiniteElement("Lagrange", triangle, 1)
```

We may then generate C++ code using a form compiler such as FFC:

Bash code

```
ffc -l dolfin Lagrange.ufl
```

This generates a file named `Lagrange.h` that we may include in our C++ program to instantiate a `FunctionSpace` on a given `Mesh`:

C++ code

```
#include <dolfin.h>
#include "Lagrange.h"

using namespace dolfin;

int main()
{
  UnitSquare mesh(8, 8);
  Lagrange::FunctionSpace V(mesh);

  ...
  return 0;
}
```

In typical applications, a `FunctionSpace` is not generated through a separate `.ufl` file, but is instead generated as part of the code generation for a variational problem.

From the Python interface, one may create a `FunctionSpace` directly, as illustrated by the following code which creates the same function space as the above example (piecewise linear Lagrange polynomials on triangles):

Python code

```
mesh = UnitSquare(8, 8)
V = FunctionSpace(mesh, "Lagrange", 1)
```

Mixed spaces. Mixed function spaces may be created from arbitrary combinations of function spaces. As an example, we consider here the creation of the *Taylor–Hood* function space for the discretization of the Stokes or incompressible Navier–Stokes equations. This mixed function space is the tensor product of a vector-valued continuous piecewise quadratic function space for the velocity field and a scalar continuous piecewise linear function space for the pressure field. This may be easily defined in either a UFL form file (for code generation and subsequent inclusion in a C++ program) or directly in a Python script as illustrated in the following code examples:

UFL code

```
V = VectorElement("Lagrange", triangle, 2)
Q = FiniteElement("Lagrange", triangle, 1)
W = V*Q
```

Python code

```
V = VectorFunctionSpace(mesh, "Lagrange", 2)
Q = FunctionSpace(mesh, "Lagrange", 1)
W = V*Q
```

DOLFIN allows the generation of arbitrarily nested mixed function spaces. A mixed function space can be used as a building block in the construction of a larger mixed space. When a mixed function space is created from more than two function spaces (nested on the same level), then one must use the `MixedElement` constructor (in UFL/C++) or the `MixedFunctionSpace` constructor (in Python). This is because Python will interpret the expression `V*Q*P` as `(V*Q)*P`, which will create a mixed function space consisting of two subspaces: the mixed space `V*Q` and the space `P`. If that is not the intention, one must instead define the mixed function space using `MixedElement([V, Q, P])` in UFL/C++ or `MixedFunctionSpace([V, Q, P])` in Python.

Subspaces. For a mixed function space, one may access its subspaces. These subspaces differ, in general, from the function spaces that were used to create the mixed space in their degree of freedom maps (`DofMap` objects). Subspaces are particularly useful for applying boundary conditions to components of a mixed element. We return to this issue below.

10.3.5 Functions

The `Function` class represents a finite element function u_h in a finite element space V_h as defined in Chapter 2:

$$u_h(x) = \sum_{j=1}^{N} U_j \phi_j(x), \tag{10.2}$$

where $U \in \mathbb{R}^N$ is the vector of degrees of freedom for the function u_h and $\{\phi_j\}_{j=1}^N$ is a basis for V_h. A `Function` is represented in terms of a `FunctionSpace` and a `GenericVector`:

`Function` = (`FunctionSpace`, `GenericVector`).

The `FunctionSpace` defines the function space V_h and the `GenericVector` holds the vector U of degrees of freedom; see Figure 10.5. When running in parallel on a distributed memory architecture, the `FunctionSpace` and the `GenericVector` are distributed across the processes.

Creating functions. `function!creation`

To create a `Function` on a `FunctionSpace`, one simply calls the constructor of the `Function` class with the `FunctionSpace` as the argument, as illustrated in the following code examples:

C++ code

```
Function u(V);
```

Python code

```
u = Function(V)
```

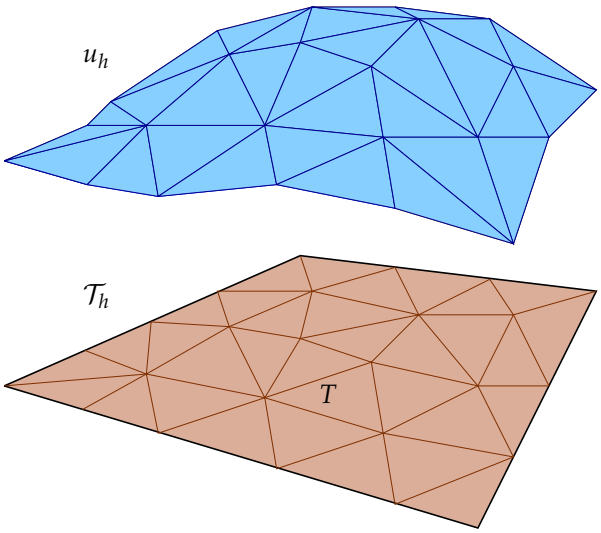


Figure 10.5: A piecewise linear finite element function u_h on a mesh consisting of triangular elements. The vector of degrees of freedom U is given by the values of u_h at the mesh vertices.

If two or more `Functions` are created on the same `FunctionSpace`, the `FunctionSpace` is shared between the `Functions`.

A `Function` is typically used to hold the computed solution to a partial differential equation. One may then obtain the degrees of freedom U by solving a system of equations, as illustrated in the following code examples:

C++ code

```
Function u(V);
solve(A, u.vector(), b);
```

Python code

```
u = Function(V)
solve(A, u.vector(), b)
```

The process of assembling and solving a linear system is handled automatically by the classes `Linear/NonlinearVariationalSolver`, which will be discussed in more detail below.

Function evaluation. A `Function` may be evaluated at arbitrary points inside the computational domain[1]. The value of a `Function` is computed by first locating the cell of the mesh containing the given point, and then evaluating the linear combination of basis functions on that cell. Finding the cell exploits an efficient search tree algorithm that is implemented as part of CGAL.

The following code examples illustrate function evaluation in the C++ and Python interfaces for scalar- and vector-valued functions:

C++ code

```
# Evaluation of scalar function
double scalar = u(0.1, 0.2, 0.3);

# Evaluation of vector-valued function
Array<double> vector(3);
u(vector, 0.1, 0.2, 0.3);
```

[1]One may also evaluate a `Function` outside of the computational domain by setting the global parameter value `"allow_extrapolation"` to true. This may sometimes be necessary when evaluating a `Function` on the boundary of a domain since round-off errors may result in points slightly outside of the domain.

Python code

```
# Evaluation of scalar function
scalar = u(0.1, 0.2, 0.3)

# Evaluation of vector-valued function
vector = u(0.1, 0.2, 0.3)
```

When running in parallel with a distributed mesh, functions can only be evaluated at points located in the portion of the mesh that is stored by the local process.

Subfunctions. For `Functions` constructed on a mixed `FunctionSpace`, subfunctions (components) of the `Function` can be accessed, for example to plot the solution components of a mixed system. Subfunctions may be accessed as either *shallow* or *deep copies*. By default, subfunctions are accessed as shallow copies, which means that the subfunctions share data with their parent functions. They provide *views* to the data of the parent function. Sometimes, it may also be desirable to access subfunctions as deep copies. A deep copied subfunction does not share its data (namely, the vector holding the degrees of freedom) with the parent `Function`. Both shallow and deep copies of `Function` objects are themselves `Function` objects and may (with some exceptions) be used as regular `Function` objects.

Creating shallow and deep copies of subfunctions is done differently in C++ and Python, as illustrated by the following code examples:

C++ code

```
Function w(W);

// Create shallow copies
Function& u = w[0];
Function& p = w[1];

// Create deep copies
Function uu = w[0];
Function pp = w[1];
```

Python code

```
w = Function(W)

# Create shallow copies
u, p = w.split()

# Create deep copies
uu, pp = w.split(deepcopy=True)
```

Note that component access, such as `w[0]`, from the Python interface does not create a new `Function` object as in the C++ interface. Instead, it creates a UFL expression that denotes a component of the original `Function`.

10.3.6 Expressions

The `Expression` class is closely related to the `Function` class in that it represents a function that can be evaluated on a finite element space. However, where a `Function` must be defined in terms of a vector of degrees of freedom, an `Expression` may be freely defined in terms of, for example, coordinate values, other geometric entities, or a table lookup.

An `Expression` may be defined in both C++ and Python by subclassing the `Expression` class and overloading the `eval` function, as illustrated in the following code examples which define the function $f(x, y) = \sin x \cos y$ as an `Expression`:

C++ code

```
class MyExpression : public Expression
{
  void eval(Array<double>& values, const Array<double>& x) const
  {
    values[0] = sin(x[0])*cos(x[1]);
  }
};

MyExpression f;
```

Python code

```
class MyExpression(Expression):
    def eval(self, values, x):
        values[0] = sin(x[0])*cos(x[1])

f = MyExpression()
```

For vector-valued (or tensor-valued) `Expressions`, one must also specify the value shape of the `Expression`. The following code examples demonstrate how to implement the vector-valued function $g(x, y) = (\sin x, \cos y)$. The value shape is defined slightly differently in C++ and Python.

C++ code

```
class MyExpression : public Expression
{
  void eval(Array<double>& values, const Array<double>& x) const
  {
    values[0] = sin(x[0]);
    values[1] = cos(x[1]);
  }

  uint value_rank() const
  {
    return 1;
  }

  uint value_dimension(uint i) const
  {
    return 2;
  }
};

MyExpression g;
```

Python code

```
class MyExpression(Expression):

    def eval(self, values, x):
        values[0] = sin(x[0])
        values[1] = cos(x[1])

    def value_shape(self):
        return (2,)
```

```
g = MyExpression()
```

The above *functor* construct for the definition of expressions is powerful and allows a user to define complex expressions, the evaluation of which may involve arbitrary operations as part of the `eval` function. For simple expressions like $f(x, y) = \sin x \cos y$ and $g(x, y) = (\sin x, \cos y)$, users of the Python interface may, alternatively, use a simpler syntax:

Python code

```
f = Expression("sin(x[0])*cos(x[1])")
g = Expression(("sin(x[0])", "cos(x[1])"))
```

The above code will automatically generate subclasses of the DOLFIN C++ `Expression` class that overload the `eval` function. This has the advantage of being more efficient, since the callback to the `eval` function takes place in C++ rather than in Python.

A feature that can be used to implement a time-dependent `Expression` in the Python interface is to use a variable name in an `Expression` string. For example, one may use the variable `t` to denote time:

Python code

```
h = Expression("t*sin(x[0])*cos(x[1])", t=0.0)
while t < T:
    h.t = t
    ...
    t += dt
```

The `t` variable has here been used to create a time-dependent `Expression`. Arbitrary variable names may be used as long as they do not conflict with the names of built-in functions, such as `sin` or `exp`.

In addition to the above examples, the Python interface allows the direct definition of (more complex) subclasses of the C++ `Expression` class by supplying C++ code for their definition. For more information, we refer to the DOLFIN Programmer's Reference.

10.3.7 Variational forms

DOLFIN relies on the FEniCS toolchain FIAT–UFL–FFC/SFC–UFC for the evaluation of finite element variational forms. Variational forms expressed in the UFL form language (Chapter 17) are compiled using one of the form compilers FFC or SFC (Chapters 11 and 15), and the generated UFC code (Chapter 16) is used by DOLFIN to evaluate (assemble) variational forms.

The UFL form language allows a wide range of variational forms to be expressed in a language close to the mathematical notation, as exemplified by the following expressions defining (in part) the bilinear and linear forms for the discretization of a linear elastic problem:

UFL code

```
a = inner(sigma(u), epsilon(v))*dx
L = dot(f, v)*dx
```

This should be compared to the corresponding mathematical notation:

$$a(u, v) = \int_\Omega \sigma(u) : \epsilon(v) \, \mathrm{d}x, \tag{10.3}$$

$$L(v) = \int_\Omega f \cdot v \, \mathrm{d}x. \tag{10.4}$$

Here, $\epsilon(v) = (\operatorname{grad} v + (\operatorname{grad} v)^T)/2$ denotes the symmetric gradient and $\sigma(v) = 2\mu\,\epsilon(v) + \lambda \operatorname{tr} \epsilon(v) I$ is the stress tensor. For a detailed presentation of the UFL form language, we refer to Chapter 17.

The code generation process must be handled explicitly by users of the C++ interface by calling a form compiler on the command-line. To solve the linear elastic problem above for a specific choice of parameter values (the Lamé constants μ and λ), a user may enter the following code in a file named `Elasticity.ufl`[2]:

UFL code

```
V = VectorElement("Lagrange", tetrahedron, 1)

u = TrialFunction(V)
v = TestFunction(V)
f = Coefficient(V)

E  = 10.0
nu = 0.3

mu    = E / (2.0*(1.0 + nu))
lmbda = E*nu / ((1.0 + nu)*(1.0 - 2.0*nu))

def sigma(v):
    return 2.0*mu*sym(grad(v)) + lmbda*tr(sym(grad(v)))*Identity(v.cell().d)

a = inner(sigma(u), sym(grad(v)))*dx
L = dot(f, v)*dx
```

This code may be compiled using a UFL/UFC compliant form compiler to generate UFC C++ code. For example, using FFC:

Bash code

```
ffc -l dolfin Elasticity.ufl
```

This generates a C++ header file (including implementation) named `Elasticity.h` which may be included in a C++ program and used to instantiate the two forms `a` and `L`:

C++ code

```
#include <dolfin.h>
#include "Elasticity.h"

using namespace dolfin;

int main()
{
  UnitSquare mesh(8, 8);
  Elasticity::FunctionSpace V(mesh);
  Elasticity::BilinearForm a(V, V);
  Elasticity::LinearForm L(V);
  MyExpression f; // code for the definition of MyExpression omitted
  L.f = f;

  return 0;
}
```

The instantiation of the forms involves the instantiation of the `FunctionSpace` on which the forms are defined. Any coefficients appearing in the definition of the forms (here the right-hand side `f`) must be attached after the creation of the forms.

Python users may rely on automated code generation, and define variational forms directly as part of a Python script:

[2]Note that 'lambda' has been deliberately misspelled since it is a reserved keyword in Python.

Python code

```
from dolfin import *

mesh = UnitSquare(8, 8)
V = VectorElement(mesh, "Lagrange", 1)

u = TrialFunction(V)
v = TestFunction(V)
f = MyExpression() # code emitted for the definition of f

E  = 10.0
nu = 0.3

mu    = E / (2.0*(1.0 + nu))
lmbda = E*nu / ((1.0 + nu)*(1.0 - 2.0*nu))

def sigma(v):
    return 2.0*mu*sym(grad(v)) + lmbda*tr(sym(grad(v)))*Identity(v.cell().d)

a = inner(sigma(u), sym(grad(v)))*dx
L = dot(f, v)*dx
```

This script will trigger automatic code generation for the definition of the `FunctionSpace` `V`. Code generation of the two forms `a` and `L` is postponed until the point when the corresponding discrete operators (the matrix and vector) are assembled.

10.3.8 Finite element assembly

A core functionality of DOLFIN is the assembly of finite element variational forms. Given a variational form (`a`), DOLFIN assembles the corresponding discrete operator (`A`). The assembly of the discrete operator follows the general algorithm described in Chapter 6. The following code illustrates how to assemble a scalar (`m`), a vector (`b`) and a matrix (`A`) from a functional (`M`), a linear form (`L`) and a bilinear form (`a`), respectively:

C++ code

```
Vector b;
Matrix A;

double m = assemble(M);
assemble(b, L);
assemble(A, a);
```

Python code

```
m = assemble(M)
b = assemble(L)
A = assemble(a)
```

The assembly of variational forms from the Python interface automatically triggers code generation, compilation and linking at run-time. The generated code is automatically instantiated and sent to the DOLFIN C++ compiler. As a result, finite element assembly from the Python interface is equally efficient as assembly from the C++ interface, with only a small overhead for handling the automatic code generation. The generated code is cached for later reuse, hence repeated assembly of the same form or running the same program twice does not re-trigger code generation. Instead, the previously generated code is automatically loaded from cache.

DOLFIN provides a common assembly algorithm for the assembly of tensors of any rank (scalars, vectors, matrices, ...) for any form. This is possible since the assembly algorithm relies on the `GenericTensor` interface, portions of the assembly algorithm that depend on the variational form and its particular discretization are generated prior to assembly, and the mesh interface is dimension-independent. The assembly algorithm accepts a number of optional arguments that control whether the sparsity of the assembled tensor should be reset before assembly and whether the tensor should be zeroed before assembly. Arguments may also be supplied to specify subdomains of the `Mesh` if the form is defined over particular subdomains (using `dx(0)`, `dx(1)` etc.).

In addition to the `assemble` function, DOLFIN provides the `assemble_system` function which assembles a pair of forms consisting of a bilinear and a linear form and applies essential boundary conditions during the assembly process. The application of boundary conditions as part of the call to `assemble_system` preserves symmetry of the matrix being assembled (see Chapter 6).

The assembly algorithms have been parallelized for both distributed memory architectures (clusters) using MPI and shared memory architectures (multi-core) using OpenMP. This is discussed in more detail in Section 10.4.

10.3.9 Boundary conditions

DOLFIN handles the application of both Neumann (natural) and Dirichlet (essential) boundary conditions.[3] Natural boundary conditions are usually applied via the variational statement of a problem, whereas essential boundary conditions are usually applied to the discrete system of equations.

Natural boundary conditions. Natural boundary conditions typically appear as boundary terms as the result of integrating by parts a partial differential equation multiplied by a test function. As a simple example, we consider the linear elastic variational problem. The partial differential equation governing the displacement of an elastic body may be expressed as

$$\begin{aligned} -\operatorname{div}\sigma(u) &= f \quad \text{in } \Omega, \\ \sigma \cdot n &= g \quad \text{on } \Gamma_{\mathrm{N}} \subset \partial\Omega, \\ u &= u_0 \quad \text{on } \Gamma_{\mathrm{D}} \subset \partial\Omega, \end{aligned} \tag{10.5}$$

where u is the unknown displacement field to be computed, $\sigma(u)$ is the stress tensor, f is a given body force, g is a given traction on a portion Γ_{N} of the boundary, and u_0 is a given displacement on a portion Γ_{D} of the boundary. Multiplying by a test function v and integrating by parts, we obtain

$$\int_\Omega \sigma(u) : \epsilon(v)\,\mathrm{d}x - \int_{\partial\Omega} (\sigma \cdot n) \cdot v\,\mathrm{d}s = \int_\Omega f \cdot v\,\mathrm{d}x, \tag{10.6}$$

where we have used the symmetry of $\sigma(u)$ to replace $\operatorname{grad} v$ by the symmetric gradient $\epsilon(v)$. Since the displacement u is known on the Dirichlet boundary Γ_{D}, we let $v = 0$ on Γ_{D}. Furthermore, we replace $\sigma \cdot n$ by the given traction g on the remaining (Neumann) portion of the boundary Γ_{N} to obtain

$$\int_\Omega \sigma(u) : \epsilon(v)\,\mathrm{d}x = \int_\Omega f \cdot v\,\mathrm{d}x + \int_{\Gamma_{\mathrm{N}}} g \cdot v\,\mathrm{d}s. \tag{10.7}$$

The following code demonstrates how to implement this variational problem in the UFL form language, either as part of a `.ufl` file or as part of a Python script:

[3] As noted in Chapter 2, Dirichlet boundary conditions may sometimes be *natural* and Neumann boundary conditions may sometimes be *essential*.

UFL code

```
a = inner(sigma(u), sym(grad(v)))*dx
L = dot(f, v)*dx + dot(g, v)*ds
```

To specify that the boundary integral `dot(g, v)*ds` should only be evaluated along the Neumann boundary Γ_N, one must specify which part of the boundary is included in the `ds` integral. If there is only one Neumann boundary, then one may simply write the `ds` integral as an integral over the entire boundary, including the Dirichlet boundary as the test function `v` will be set to zero along the Dirichlet boundary.

In cases where there is more than one Neumann boundary condition, one must instead specify the Neumann boundary in terms of a `FacetFunction`. This `FacetFunction` must specify for each facet of the `Mesh` to which part of the boundary it belongs. For the current example, an appropriate strategy is to mark each facet on the Neumann boundary by `0` and all other facets (including facets internal to the domain) by `1`. This can be accomplished in a number of different ways. One simple way to do this is to use the program MeshBuilder and graphically mark the facets of the `Mesh`. Another option is through the DOLFIN class `SubDomain`. The following code illustrates how to mark all boundary facets to the left of $x = 0.5$ as the first Neumann boundary and all other boundary facets as the second Neumann boundary. Note the use of the `on_boundary` argument supplied by DOLFIN to the `inside` function. This argument informs whether a point is located on the boundary $\partial\Omega$ of Ω, and this allows us to mark only facets that are on the boundary and to the left of $x = 0.5$. Also note the use of `DOLFIN_EPS` which makes sure that we include points that, as a result of finite precision arithmetic, may be located just to the right of $x = 0.5$.

C++ code

```
class NeumannBoundary : public SubDomain
{
  bool inside(const Array<double>& x, bool on_boundary) const
  {
    return x[0] < 0.5 + DOLFIN_EPS && on_boundary;
  }
};

NeumannBoundary neumann_boundary;
FacetFunction<uint> exterior_facet_domains(mesh);
exterior_facet_domains.set_all(1);
neumann_boundary.mark(exterior_facet_domains, 0);
```

Python code

```
class NeumannBoundary(SubDomain):
    def inside(self, x, on_boundary):
        return x[0] < 0.5 + DOLFIN_EPS and on_boundary

neumann_boundary = NeumannBoundary()
exterior_facet_domains = FacetFunction("uint", mesh)
exterior_facet_domains.set_all(1)
neumann_boundary.mark(exterior_facet_domains, 0)
```

When combined with integrals defined using `ds(0)` and `ds(1)`, those integrals will correspond to integration over the domain boundary to the left of $x = 0.5$ and all facets to the right of $x = 0.5$, respectively.

Once the boundaries have been specified as a `FacetFunction`, that object can be used to define the corresponding domains of integration. This is done differently in C++ and Python. From C++, one must assign to the `ds` member variable of the corresponding forms:

C++ code

```
a.ds = exterior_facet_domains;
L.ds = exterior_facet_domains;
```

In addition to `exterior_facet_domains` specified in terms of the `ds` member variable, one may similarly specify `cell_domains` using the `dx` member variable and `interior_facet_domains` using the `dS` variable. Note that different forms may potentially use different definitions of their boundaries. From Python, one may simply connect the boundary definition to the corresponding measure by subscripting:

Python code

```
dss = ds[neumann_boundary]
a = ... + g*v*dss(0) + h*v*dss(1) + ...
```

The correct specification of boundaries is a common error source. For debugging the specification of boundary conditions, it can be helpful to plot the `FacetFunction` that specifies the boundary markers by writing the `FacetFunction` to a VTK file (see the file I/O section) or using the `plot` command. When using the `plot` command, the plot shows the facet values interpolated to the vertices of the `Mesh`. As a result, care must be taken to interpret the plot close to domain boundaries (corners) in this case. The issue is not present in the VTK output.

Essential boundary conditions. The application of essential boundary conditions is handled by the class `DirichletBC`. Using this class, one may specify a Dirichlet boundary condition in terms of a `FunctionSpace`, a `Function` or an `Expression`, and a subdomain. The subdomain may be specified either in terms of a `SubDomain` object or in terms of a `FacetFunction`. A `DirichletBC` specifies that the solution should be equal to the given value on the given subdomain.

The following code examples illustrate how to define the Dirichlet condition $u(x) = u_0(x) = \sin x$ on the Dirichlet boundary Γ_D (assumed here to be the part of the boundary to the right of $x = 0.5$) for the elasticity problem (10.5) using the `SubDomain` class. Alternatively, the subdomain may be specified using a `FacetFunction`.

C++ code

```
class DirichletValue : public Expression
{
  void eval(Array<double>& values, const Array<double>& x) const
  {
    values[0] = sin(x[0]);
  }
};

class DirichletBoundary : public SubDomain
{
  bool inside(const Array<double>& x, bool on_boundary) const
  {
    return x[0] > 0.5 - DOLFIN_EPS && on_boundary;
  }
};

DirichletValue u_0;
DirichletBoundary Gamma_D;

DirichletBC bc(V, u_0, Gamma_D);
```

Python code

```
class DirichletValue(Expression):
    def eval(self, value, x):
        values[0] = sin(x[0])

class DirichletBoundary(SubDomain):
    def inside(self, x, on_boundary):
        return x[0] > 0.5 - DOLFIN_EPS and on_boundary

u_0 = DirichletValue()
Gamma_D = DirichletBoundary()

bc = DirichletBC(V, u_0, Gamma_D)
```

Python users may also use the following compact syntax:

Python code

```
u_0 = Expression("sin(x[0])")
bc = DirichletBC(V, u_0, "x[0] > 0.5 && on_boundary")
```

To speed up the application of Dirichlet boundary conditions, users of the Python interface may also use the function `compile_subdomains`. For details of this, we refer to the DOLFIN Programmer's Reference.

A Dirichlet boundary condition can be applied to a linear system or to a vector of degrees of freedom associated with a `Function`, as illustrated by the following code examples:

C++ code

```
bc.apply(A, b);
bc.apply(u.vector());
```

Python code

```
bc.apply(A, b)
bc.apply(u.vector())
```

The application of a Dirichlet boundary condition to a linear system will identify all degrees of freedom that should be set to the given value and modify the linear system such that its solution respects the boundary condition. This is accomplished by zeroing and inserting 1 on the diagonal of the rows of the matrix corresponding to Dirichlet values, and inserting the Dirichlet value in the corresponding entry of the right-hand side vector. This application of boundary conditions does not preserve symmetry. If symmetry is required, one may alternatively consider using the `assemble_system` function which applies Dirichlet boundary conditions symmetrically as part of the assembly process.

Multiple boundary conditions may be applied to a single system or vector. If two different boundary conditions are applied to the same degree of freedom, the last applied value will overwrite any previously set values.

10.3.10 Variational problems

Variational problems (finite element discretizations of partial differential equations) can be easily solved in DOLFIN using the `solve` function. Both linear and nonlinear problems can be solved. A linear problem must be expressed in the following canonical form: find $u \in V$ such that

$$a(u, v) = L(v) \quad \forall\, v \in \hat{V}. \tag{10.8}$$

A nonlinear problem must be expressed in the following canonical form: find $u \in V$ such that

$$F(u; v) = 0 \quad \forall v \in \hat{V}. \tag{10.9}$$

In the case of a linear variational problem specified in terms of a bilinear form `a` and a linear form `L`, the solution is computed by assembling the matrix `A` and vector `b` of the corresponding linear system, then applying boundary conditions to the system, and finally solving the linear system. In the case of a nonlinear variational problem specified in terms of a linear form `F` and a bilinear form `J` (the derivative or Jacobian of `F`), the solution is computed by Newton's method.

The code examples below demonstrate how to solve a linear variational problem specified in terms of a bilinear form `a`, a linear form `L` and a list of Dirichlet boundary conditions given as `DirichletBC` objects:

C++ code

```
std::vector<const BoundaryCondition*> bcs;
bcs.push_back(&bc0);
bcs.push_back(&bc1);
bcs.push_back(&bc2);

Function u(V);
solve(a == L, u, bcs);
```

Python code

```
bcs = [bc0, bc1, bc2]

u = Function(V)
solve(a == L, u, bcs=bcs)
```

To solve a nonlinear variational problem, one must supply a linear form `F` and, in the case of C++, its derivative `J`, which is a bilinear form. In Python, the derivative is computed automatically but may also be specified manually. In many cases, the derivative can be easily computed using the function `derivative`, either in a `.ufl` form file or as part of a Python script. We here demonstrate how a nonlinear problem may be solved using the Python interface. Nonlinear variational problems may be solved similarly in C++.

Python code

```
u  = Function(V)
v  = TestFunction(V)
F  = inner((1 + u**2)*grad(u), grad(v))*dx - f*v*dx

# Let DOLFIN compute Jacobian
solve(F == 0, u, bcs=bcs)

# Differentiate to get Jacobian
J = derivative(F, u)

# Supply Jacobian manually
solve(F == 0, u, bcs=bcs, J=J)
```

More advanced control over the solution process may be gained by using the classes `LinearVariational{Problem,Solver}` and `NonlinearVariational{Problem,Solver}`. Use of these classes is illustrated by the following code examples:

Python code

```
u = Function(V)
problem = LinearVariationalProblem(a, L, u, bcs=bcs)
solver = LinearVariationalSolver(problem)
solver.parameters["linear_solver"] = "gmres"
solver.parameters["preconditioner"] = "ilu"
solver.solve()
```

Python code

```
u = Function(V)
problem = NonlinearVariationalProblem(F, u, bcs=bcs, J=J)
solver = NonlinearVariationalSolver(problem)
solver.parameters["linear_solver"] = "gmres"
solver.parameters["preconditioner"] = "ilu"
solver.solve()
```

These classes may be used similarly from C++.

The solver classes provide a range of parameters that can be adjusted to control the solution process. For example, to view the list of available parameters for a `LinearVariationalSolver` or `NonlinearVariationalSolver`, issue the following commands:

C++ code

```
info(solver.parameters, true)
```

Python code

```
info(solver.parameters, True)
```

10.3.11 File I/O and visualization

Preprocessing. DOLFIN has capabilities for mesh generation only in the form of the built-in meshes `UnitSquare`, `UnitCube`, etc. External software must be used to generate more complicated meshes. To simplify this process, DOLFIN provides a simple script `dolfin-convert` to convert meshes from other formats to the DOLFIN XML format. Currently supported file formats are listed in Table 10.4. The following code illustrates how to convert a mesh from the Gmsh format (suffix `.msh` or `.gmsh`) to the DOLFIN XML format:

Bash code

```
dolfin-convert mesh.msh mesh.xml
```

Once a mesh has been converted to the DOLFIN XML file format, it can be read into a program, as illustrated by the following code examples:

C++ code

```
Mesh mesh("mesh.xml");
```

Python code

```
mesh = Mesh("mesh.xml")
```

Suffix	File format
`.xml`	DOLFIN XML format
`.ele / .node`	Triangle file format
`.mesh`	Medit format, generated by TetGen with option `-g`
`.msh / .gmsh`	Gmsh version 2.0 format
`.grid`	Diffpack tetrahedral grid format
`.inp`	Abaqus tetrahedral grid format
`.e / .exo`	Sandia Exodus II file format
`.ncdf`	ncdump'ed Exodus II file format
`.vrt/.cell`	Star-CD tetrahedral grid format

Table 10.4: List of file formats supported by the `dolfin-convert` script.

Postprocessing. To visualize a solution (`Function`), a `Mesh` or a `MeshFunction`, the `plot` command[4] can be issued, from either C++ or Python:

C++ code

```
plot(u);
plot(mesh);
plot(mesh_function);
```

Python code

```
plot(u)
plot(mesh)
plot(mesh_function)
```

Example plots generated using the `plot` command are presented in Figures 10.6 and 10.7. From Python, one can also plot expressions and finite elements:

Python code

```
plot(grad(u))
plot(u*u)

element = FiniteElement("BDM", tetrahedron, 3)
plot(element)
```

To enable interaction with a plot window (rotate, zoom) from Python, call the function `interactive`, or add an optional argument `interactive=True` to the `plot` command.

The `plot` command provides rudimentary plotting, and advanced postprocessing is better handled by external software such as ParaView and MayaVi2. This is easily accomplished by storing the solution (a `Function` object) to file in PVD format (ParaView Data, an XML-based format). This can be done in both C++ and Python by writing to a file with the `.pvd` extension, as illustrated in the following code examples:

C++ code

```
File file("solution.pvd");
file << u;
```

[4]The plot command requires a working installation of the `viper` Python module. Plotting finite elements requires access to FFC and the `soya` Python plotting module.

Figure 10.6: Plotting a mesh using the DOLFIN `plot` command, here the mesh `dolfin-1.xml.gz` distributed with DOLFIN.

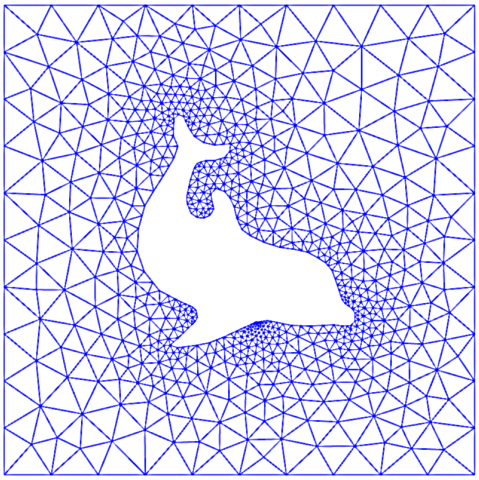
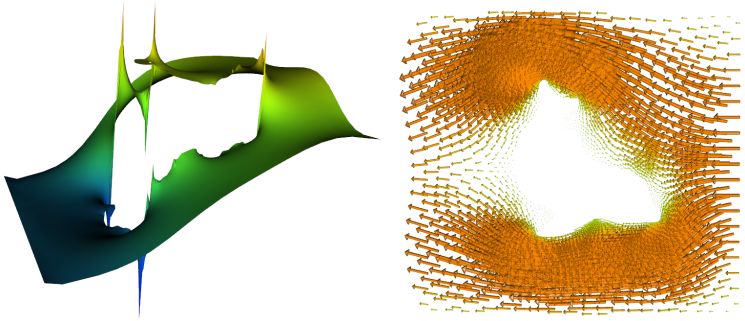


Figure 10.7: Plotting a scalar and a vector-valued function using the DOLFIN `plot` command, here the pressure (left) and velocity (right) from a solution of the Stokes equations on the mesh from Figure 10.6.



Python code

```
file = File("solution.pvd")
file << u
```

The standard PVD format is ASCII based, hence the file size can become very large for large data sets. To use a compressed binary format, a string `"compressed"` can be used when creating a PVD-based `File` object:

C++ code

```
File file("solution.pvd", "compressed");
```

If multiple `Functions` are written to the same file (by repeated use of `<<`), then the data is interpreted as a time series, which may then be animated in ParaView or MayaVi2. Each frame of the time series is stored as a `.vtu` (VTK unstructured data) file, with references to these files stored in the `.pvd` file. When writing time-dependent data, it can be useful to store the time `t` of each snapshot. This is done as illustrated below:

C++ code

```
File file("solution.pvd", "compressed");
file << std::make_pair<const Function*, double>(&u, t);
```

Python code

```
file = File("solution.pvd", "compressed");
file << (u, t)
```

Storing the time is particularly useful when animating simulations that use a varying time step.

The PVD format supports parallel post-processing. When running in parallel, a single `.pvd` file is created and a `.vtu` file is created for the data on each partition. Results computed in parallel can be viewed seamlessly using ParaView.

DOLFIN XML format. DOLFIN XML is the native format of DOLFIN. An advantage of XML is that it is a robust and human-readable format. If the files are compressed, there is also little overhead in terms of file size compared to a binary format.

Many of the classes in DOLFIN can be written to and from DOLFIN XML files using the standard stream operators `<<` and `>>`, as illustrated in the following code examples:

C++ code

```
File vector_file("vector.xml");
vector_file << vector;
vector_file >> vector;

File mesh_file("mesh.xml");
mesh_file << mesh;
mesh_file >> mesh;

File parameters_file("parameters.xml");
parameters_file << parameters;
parameters_file >> parameters;
```

Python code

```
vector_file = File("vector.xml")
vector_file << vector
vector_file >> vector
```

```
mesh_file = File("mesh.xml")
mesh_file << mesh
mesh_file >> mesh

parameters_file = File("parameters.xml")
parameters_file << parameters
parameters_file >> parameters
```

One cannot read/write `Function` and `FunctionSpace` objects since the representation of a `FunctionSpace` (and thereby the representation of a `Function`) relies on generated code.

DOLFIN automatically handles reading and writing of gzipped XML files. Thus, one may save space by storing meshes and other data in gzipped XML files (with suffix `.xml.gz`).

Time series. For time-dependent problems, it may be useful to store a sequence of solutions or meshes in a format that enables fast reading/writing of data. For this purpose, DOLFIN provides the `TimeSeries` class. This enables the storage of a series of `Vectors` (of degrees of freedom) and/or `Meshes`. The following code illustrates how to store a series of `Vectors` and `Meshes` to a `TimeSeries`:

C++ code

```
TimeSeries time_series("simulation_data");

while (t < T)
{
  ...
  time_series.store(u.vector(), t);
  time_series.store(mesh, t);
  t += dt;
}
```

Python code

```
time_series = TimeSeries("simulation_data")

while t < T:
    ...
    time_series.store(u.vector(), t)
    time_series.store(mesh, t)
    t += dt
```

Data in a `TimeSeries` are stored in a binary format with one file for each stored dataset (`Vector` or `Mesh`) and a common index. Data may be retrieved from a `TimeSeries` by calling the `retrieve` member function as illustrated in the code examples below. If a dataset is not stored at the requested time, then the values are interpolated linearly for `Vectors`. For `Meshes`, the closest data point will be used.

C++ code

```
time_series.retrieve(u.vector(), t);
time_series.retrieve(mesh, t);
```

Python code

```
time_series.retrieve(u.vector(), t)
time_series.retrieve(mesh, t)
```

Log level	value
ERROR	40
WARNING	30
INFO	20
PROGRESS	16
DBG / DEBUG	10

Table 10.5: Log levels in DOLFIN.

10.3.12 Logging / diagnostics

DOLFIN provides a simple interface for the uniform handling of log messages, including warnings and errors. All messages are collected to a single stream, which allows the destination and formatting of the output from an entire program, including the DOLFIN library, to be controlled by the user.

Printing messages. Informational messages from DOLFIN are normally printed using the `info` command. This command takes a string argument and an optional list of variables to be formatted, much like the standard C `printf` command. Note that the `info` command automatically appends a newline to the given string. Alternatively, C++ users may use the `dolfin::cout` and `dolfin::endl` objects for C++ style formatting of messages as illustrated below.

C++ code

```
info("Assembling system of size %d x %d.", M, N);
cout << "Assembling system of size " << M << " x " << N << "." << endl;
```

Python code

```
info("Assembling system of size %d x %d." % (M, N))
```

The `info` command and the `dolfin::cout/endl` objects differ from the standard C `printf` command and the C++ `std::cout/endl` objects in that the output is directed into a special stream, the output of which may be redirected to destinations other than standard output. In particular, one may completely disable output from DOLFIN, or select the verbosity of printed messages, as explained below.

Warnings and errors. In addition to the `info` command, DOLFIN provides the commands `warning` and `error` that can be used to issue warnings and errors, respectively. These two commands work in much the same way as the `info` command. However, the `warning` command will prepend the given message with `"*** Warning: "` and the `error` command will raise an exception that can be caught, from both C++ and Python. Both commands will also print the message at a *log level* higher than messages printed using `info`.

Setting the log level. The DOLFIN log level determines which messages routed through the logging system will be printed. Only messages on a level higher than or equal to the current log level are printed. The log level of DOLFIN may be set using the function `set_log_level`. This function expects an integer value that specifies the log level. To simplify the specification of the log level, one may use one of a number of predefined log levels as listed in Table 10.5. The default log level is `INFO`. Log messages may be switched off entirely by calling the command `set_log_active(false)` from C++ and `set_log_active(False)` from Python. For technical reasons, the log level for debugging messages is named `DBG` in C++ and `DEBUG` in Python. This is summarized in Table 10.5.

To print messages at an arbitrary log level, one may specify the log level to the `log` command, as illustrated in the code examples below.

C++ code

```
info("Test message");                      // will be printed
cout << "Test message" << endl;            // will be printed
log(DBG, "Test message");                  // will not be printed
log(15, "Test message");                   // will not be printed

set_log_level(DBG);
info("Test message");                      // will be printed
cout << "Test message" << endl;            // will be printed
log(DBG, "Test message");                  // will be printed
log(15, "Test message");                   // will be printed

set_log_level(WARNING);
info("Test message");                      // will not be printed
cout << "Test message" << endl;            // will not be printed
warning("Test message");                   // will be printed
std::cout << "Test message" << std::endl;  // will be printed!
```

Python code

```
info("Test message")                       # will be printed
log(DEBUG, "Test message")                 # will not be printed
log(15, "Test message")                    # will not be printed

set_log_level(DEBUG)
info("Test message")                       # will be printed
log(DEBUG, "Test message")                 # will be printed
log(15, "Test message")                    # will be printed

set_log_level(WARNING)
info("Test message")                       # will not be printed
warning("Test message")                    # will be printed
print "Test message"                       # will be printed!
```

Printing objects. Many of the standard DOLFIN objects can be printed using the `info` command, as illustrated in the code examples below.

C++ code

```
info(vector);
info(matrix);
info(solver);
info(mesh);
info(mesh_function);
info(function);
info(function_space);
info(parameters);
```

Python code

```
info(vector)
info(matrix)
info(solver)
info(mesh)
info(mesh_function)
info(function)
```

```
info(function_space)
info(parameters)
```

The above commands will print short informal messages. For example, the command `info(mesh)` may result in the following output:

Generated code

```
<Mesh of topological dimension 2 (triangles) with 25 vertices and 32 cells, ordered>
```

In the Python interface, the same short informal message can be printed by calling `print mesh`. To print more detailed data, one may set the verbosity argument of the `info` function to true (defaults to false), which will print a detailed summary of the object.

C++ code

```
info(mesh, true);
```

Python code

```
info(mesh, True)
```

The detailed output for some objects may be very lengthy.

Tasks and progress bars. In addition to basic commands for printing messages, DOLFIN provides a number of commands for organizing the diagnostic output from a simulation program. Two such commands are `begin` and `end`. These commands can be used to nest the output from a program; each call to `begin` increases the indentation level by one unit (two spaces), while each call to `end` decreases the indentation level by one unit.

Another way to provide feedback is via progress bars. DOLFIN provides the `Progress` class for this purpose. Although an effort has been made to minimize the overhead of updating the progress bar, it should be used with care. If only a small amount of work is performed in each iteration of a loop, the relative overhead of using a progress bar may be substantial. The code examples below illustrate the use of the `begin/end` commands and the progress bar.

C++ code

```
begin("Starting nonlinear iteration.");
info("Updating velocity.");
info("Updating pressure.");
info("Computing residual.");
end();

Progress p("Iterating over all cells.", mesh.num_cells());
for (CellIterator cell(mesh); !cell.end(); ++cell)
{
  ...
  p++;
}

Progress q("Time-stepping");
while (t < T)
{
  ...
  t += dt;
  q = t / T;
}
```

Python code

```
begin("Starting nonlinear iteration.")
info("Updating velocity.")
info("Updating pressure.")
info("Computing residual.")
end()

p = Progress("Iterating over all cells.", mesh.num_cells())
for cell in cells(mesh):
  ...
  p += 1

q = Progress("Time-stepping")
while t < T:
  ...
  t += dt
  q.update(t / T)
```

Setting timers. Timing can be accomplished using the `Timer` class. A `Timer` is automatically started when it is created, and automatically stopped when it goes out of scope. Creating a `Timer` at the start of a function is therefore a convenient way to time that function, as illustrated in the code examples below.

C++ code

```
void solve(const Matrix& A, Vector& x, const Vector& b)
{
  Timer timer("Linear solve");
  ...
}
```

Python code

```
def solve(A, b):
  timer = Timer("Linear solve")
  ...
  return x
```

One may explicitly call the `start` and `stop` member functions of a `Timer`. To directly access the value of a timer, the `value` member function can be called. A summary of the values of all timers created during the execution of a program can be printed by calling the `list_timings` function.

10.3.13 Parameters

DOLFIN keeps a global database of parameters that control the behavior of its various components. Parameters are controlled via a uniform type-independent interface that allows the retrieval of parameter values, modification of parameter values, and the addition of new parameters to the database. Different components (classes) of DOLFIN also rely on parameters that are local to each instance of the class. This permits different parameter values to be set for different objects of a class.

Parameter values can be either integer-valued, real-valued (standard double), string-valued or boolean-valued. Parameter names must not contain spaces.

Accessing parameters. Global parameters can be accessed through the global variable `parameters`. The below code illustrates how to print the values of all parameters in the global parameter database, and how to access and change parameter values.

C++ code

```
info(parameters, True);
uint num_threads = parameters["num_threads"];
bool allow_extrapolation = parameters["allow_extrapolation"];
parameters["num_threads"] = 8;
parameters["allow_extrapolation"] = true;
```

Python code

```
info(parameters, True)
num_threads = parameters["num_threads"]
allow_extrapolation = parameters["allow_extrapolation"]
parameters["num_threads"] = 8
parameters["allow_extrapolation"] = True
```

Parameters that are local to specific components of DOLFIN can be controlled by accessing the member variable named `parameters`. The following code illustrates how to set some parameters for a Krylov solver:

C++ code

```
KrylovSolver solver;
solver.parameters["absolute_tolerance"] = 1e-6;
solver.parameters["report"] = true;
solver.parameters("gmres")["restart"] = 50;
solver.parameters("preconditioner")["reuse"] = true;
```

Python code

```
solver = KrylovSolver()
solver.parameters["absolute_tolerance"] = 1e-6
solver.parameters["report"] = True
solver.parameters["gmres"]["restart"] = 50
solver.parameters["preconditioner"]["reuse"] = True
```

The above example accesses the nested parameter databases `"gmres"` and `"preconditioner"`. DOLFIN parameters may be nested to arbitrary depths, which helps with organizing parameters into different categories. Note the subtle difference in accessing nested parameters in the two interfaces. In the C++ interface, nested parameters are accessed by brackets `("...")`, and in the Python interface are they accessed by square brackets `["..."]`. The parameters that are available for a certain component can be viewed by using the `info` function.

Adding parameters. Parameters can be added to an existing parameter database using the `add` member function which takes the name of the new parameter and its default value. It is also simple to create new parameter databases by creating a new instance of the `Parameters` class. The following code demonstrates how to create a new parameter database and adding to it a pair of integer-valued and floating-point valued parameters:

C++ code

```
Parameters parameters("my_parameters");
my_parameters.add("foo", 3);
my_parameters.add("bar", 0.1);
```

Python code

```
my_parameters = Parameters("my_parameters")
my_parameters.add("foo", 3)
my_parameters.add("bar", 0.1)
```

A parameter database resembles the `dict` class in the Python interface. A user can iterate over the `keys`, `values` and `items`:

Python code

```
for key, value in parameters.items():
    print key, value
```

A Python `dict` can also be used to update a Parameter database:

Python code

```
d = dict(num_threads=4, krylov_solver=dict(absolute_tolerance=1e-6))
parameters.update(d)
```

A parameter database can also be created in more compact way in the Python interface:

Python code

```
my_parameters = Parameters("my_parameters", foo=3, bar=0.1,
                           nested=Parameters("nested", baz=True))
```

Parsing command-line parameters. Command-line parameters may be parsed into the global parameter database or into any other parameter database. The following code illustrates how to parse command-line parameters in C++ and Python, and how to pass command-line parameters to the program:

C++ code

```
int main(int argc, char* argv[])
{
  ...
  parameters.parse(argc, argv);
  ...
}
```

Python code

```
parameters.parse()
```

Bash code

```
python myprogram.py --num_threads 8 --allow_extrapolation true
```

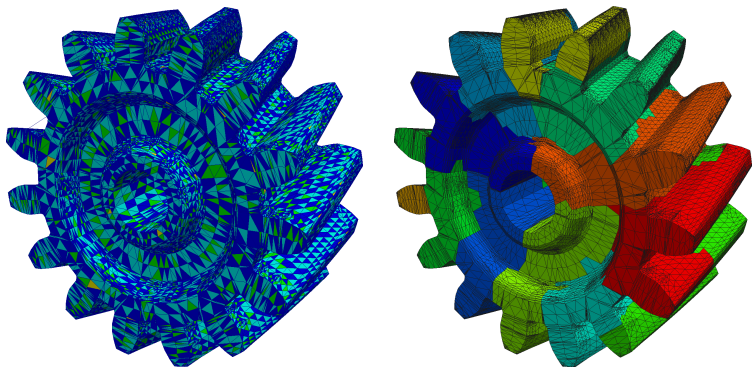
Storing parameters to file. It can be useful to store parameter values to file, for example to document which parameter values were used to run a simulation or to reuse a set of parameter values from a previous run. The following code illustrates how to write and then read back parameter values to/from a DOLFIN XML file:

C++ code

```
File file("parameters.xml");
file << parameters;
file >> parameters;
```

Python code

```
file = File("parameters.xml")
file << parameters
file >> parameters
```



(a) (b)

Figure 10.8: A mesh that is (a) colored based on facet connectivity such that cells that share a common facet have different colors and (b) partitioned into 12 parts, with each partition indicated by a color.

At startup, DOLFIN automatically scans the current directory and the directory `.config/fenics` in the user's home directory (in that order) for a file named `dolfin_parameters.xml`. If found, these parameters are read into DOLFIN's global parameter database.

10.4 Implementation notes

In this section, we comment on specific aspects of the implementation of DOLFIN, including parallel computing, the generation of the Python interface, and just-in-time compilation.

10.4.1 Parallel computing

DOLFIN supports parallel computing on multi-core workstations through to massively parallel supercomputers. It is designed such that users can perform parallel simulations using the same code that is used for serial computations.

Two paradigms for parallel simulation are supported. The first paradigm is multithreading for shared memory machines. The second paradigm is fully distributed parallelization for distributed memory machines. For both paradigms, special preprocessing of a mesh is required. For multithreaded parallelization, a so-called coloring approach is used (see Figure 10.8a), and for distributed parallelization a mesh partitioning approach is used (see Figure 10.8b). Aspects of these two approaches are discussed below. It is also possible to combine the approaches, thereby yielding hybrid approaches to leverage the power of modern clusters of multi-core processors.

Shared memory parallel computing. Multithreaded assembly for finite element matrices and vectors on shared memory machines is supported using OpenMP. It is activated by setting the number of threads to use via the parameter system. For example, the code

C++ code

```
parameters["num_threads"] = 6;
```

instructs DOLFIN to use six threads in the assembly process. During assembly, DOLFIN loops over the cells or cell facets in a mesh, and computes local contributions to the global matrix or vector, which are then added to the global matrix or vector. When using multithreaded assembly, each thread is assigned a collection of cells or facets for which it is responsible. This is transparent to the user.

The use of multithreading requires design care to avoid race conditions, which occur if multiple threads attempt to write to the same memory location at the same time. Race conditions will typically result in unpredictable behavior of a program. To avoid race conditions during assembly, which would

occur if two threads were to add values to a global matrix or vector at almost the same time, DOLFIN uses a graph coloring approach. Before assembly, the mesh on a given process is 'colored' such that each cell is assigned a color (which in practice is an integer) and such that no two neighboring cells have the same color. The sense in which cells are neighbors for a given problem depends on the type of finite element being used. In most cases, cells that share a vertex are considered neighbors, but in other cases cells that share edges or facets may be considered neighbors. During assembly, cells are assembled by color. All cells of the first color are shared among the threads and assembled, and this is followed by the next color. Since cells of the same color are not neighbors, and therefore do not share entries in the global matrix or vector, race conditions will not occur during assembly. The coloring of a mesh is performed in DOLFIN using either the interface to the Boost Graph Library or the interface to Zoltan (which is part of the Trilinos project). Figure 10.8a shows a mesh that has been colored such that no two neighboring cells (in the sense of a shared facet) are of the same color.

Multithreaded support in third-party linear algebra libraries is limited at the present time, but is an area of active development. The LU solver PaStiX, which can be accessed via the PETSc linear algebra backend, supports multithreaded parallelism.

Distributed memory parallel computing. Fully distributed parallel computing is supported using the Message Passing Interface (MPI). To perform parallel simulations, DOLFIN should be compiled with MPI and a parallel linear algebra backend (such as PETSc or Trilinos) enabled. To execute a parallel simulation, a DOLFIN program should be launched using `mpirun` (the name of the program to launch MPI programs may differ on some computers). A C++ program using 16 processes can be executed using:

Bash code

```
mpirun -n 16 ./myprogram
```

and for Python:

Bash code

```
mpirun -n 16 python myprogram.py
```

DOLFIN supports fully distributed parallel meshes, which means that each processor has a copy of only the portion of the mesh for which it is responsible. This approach is scalable since no processor is required to hold a copy of the full mesh. An important step in a parallel simulation is the partitioning of the mesh. DOLFIN can perform mesh partitioning in parallel using the libraries ParMETIS and SCOTCH (Pellegrini). The library to be used for mesh partitioning can be specified via the parameter system, e.g., to use SCOTCH:

C++ code

```
parameters["mesh_partitioner"] = "SCOTCH";
```

or to use ParMETIS:

Python code

```
parameters["mesh_partitioner"] = "ParMETIS"
```

Figure 10.8b shows a mesh that has been partitioned in parallel into 12 domains. One process would take responsibility for each domain.

If a parallel program is launched using MPI and a parallel linear algebra backend is enabled, then linear algebra operations will be performed in parallel. In most applications, this will be transparent to the user. Parallel output for postprocessing is supported through the PVD output format, and is

used in the same way as for serial output. Each process writes an output file, and the single main output file points to the files produced by the different processes.

10.4.2 Implementation and generation of the Python interface

The DOLFIN C++ library is wrapped to Python using the Simplified Wrapper and Interface Generator SWIG (Beazley, 1996; SWIG); see Chapter 19 for more details. The wrapped C++ library is accessible in a Python module named `cpp` residing inside the main `dolfin` module of DOLFIN. This means that the compiled module, with all its functions and classes, can be accessed directly by:

Python code

```
from dolfin import cpp
Function = cpp.Function
assemble = cpp.assemble
```

The classes and functions in the `cpp` module have the same functionality as the corresponding classes and functions in the C++ interface. In addition to the wrapper layer automatically generated by SWIG, the DOLFIN Python interface relies on a number of components implemented directly in Python. Both are imported into the Python module named `dolfin`. In the following sections, the key customizations to the DOLFIN interface that facilitate this integration are presented. The Python interface also integrates well with the NumPy and SciPy toolkits, which is also discussed below.

10.4.3 UFL integration and just-in-time compilation

In the Python interface, the UFL form language has been integrated with the Python wrapped DOLFIN C++ module. When explaining the integration, we use in this section the notation `dolfin::Foo` or `dolfin::bar` to denote a C++ class or function in DOLFIN. The corresponding SWIG-wrapped classes or functions will be referred to as `cpp.Foo` and `cpp.bar`. A class in UFL will be referred to as `ufl.Foo` and a class in UFC as `ufc::foo` (note lower case). The Python classes and functions in the added Python layer on top of the wrapped C++ library, will be referred to as `dolfin.Foo` or `dolfin.bar`. The prefixes of the classes and functions are sometimes skipped for convenience. Most of the code snippets presented in this section are pseudo code. Their purpose is to illustrate the logic of a particular method or function. Parts of the actual code may be intentionally excluded. An interested reader can examine particular classes or functions in the code for a full understanding of the implementation.

Construction of function spaces. In the Python interface, `ufl.FiniteElement` and `dolfin::FunctionSpace` are integrated. The declaration of a `FunctionSpace` is similar to that of a `ufl.FiniteElement`, but instead of a cell type (for example, `triangle`) the `FunctionSpace` constructor takes a `cpp.Mesh` (`dolfin.Mesh`):

Python code

```
mesh = UnitSquare(8, 8)
V = FunctionSpace(mesh, "Lagrange", 1)
```

In the Python constructor of `FunctionSpace`, a `ufl.FiniteElement` is instantiated. The `FiniteElement` is passed to a just-in-time (JIT) compiler, which returns compiled and Python-wrapped `ufc` objects: a `ufc::finite_element` and a `ufc::dofmap`. These two objects, together with the mesh, are used to instantiate a `cpp.FunctionSpace`. The following pseudo code illustrates the instantiation of a `FunctionSpace` from the Python interface:

Python code

```
class FunctionSpace(cpp.FunctionSpace):
    def __init__(self, mesh, family, degree):
        # Figure out the domain from the mesh topology
        if mesh.topology().dim() == 2:
            domain = ufl.triangle
        else:
            domain = ufl.tetrahedron

        # Create the UFL FiniteElement
        self.ufl_element = ufl.FiniteElement(family, domain, degree)

        # JIT compile and instantiate the UFC classes
        ufc_element, ufc_dofmap = jit(self.ufl_element)

        # Instantiate DOLFIN classes and finally the FunctionSpace
        dolfin_element = cpp.FiniteElement(ufc_element)
        dolfin_dofmap = cpp.DofMap(ufc_dofmap, mesh)
        cpp.FunctionSpace.__init__(self, mesh, dolfin_element, dolfin_dofmap)
```

Constructing arguments (trial and test functions). The `ufl.Argument` class (the base class of `ufl.TrialFunction` and `ufl.TestFunction`) is subclassed in the Python interface. Instead of using a `ufl.FiniteElement` to instantiate the classes, a DOLFIN `FunctionSpace` is used:

Python code

```
u = TrialFunction(V)
v = TestFunction(V)
```

The `ufl.Argument` base class is instantiated in the subclassed constructor by extracting the `ufl.FiniteElement` from the passed `FunctionSpace`, which is illustrated by the following pseudo code:

Python code

```
class Argument(ufl.Argument):
    def __init__(self, V, index=None):
        ufl.Argument.__init__(self, V.ufl_element, index)
        self.V = V
```

The `TrialFunction` and `TestFunction` are then defined using the subclassed `Argument` class:

Python code

```
def TrialFunction(V):
    return Argument(V, -1)

def TestFunction(V):
    return Argument(V, -2)
```

Coefficients, functions and expressions. When a UFL form is defined using a `Coefficient`, a user must associate with the form either a discrete finite element `Function` or a user-defined `Expression` before the form is assembled. In the C++ interface of DOLFIN, a user needs to explicitly carry out this association (`L.f = f`). In the Python interface of DOLFIN, the `ufl.Coefficient` class is combined with the DOLFIN `Function` and `Expression` classes, and the association between the coefficient as a symbol in the form expression (`Coefficient`) and its value (`Function` or `Expression`) is automatic. A user can therefore assemble a form defined using instances of these combined classes directly:

Python code

```
class Source(Expression):
    def eval(self, values, x):
        values[0] = sin(x[0])

v = TestFunction(V)
f = Source()
L = f*v*dx
b = assemble(L)
```

The `Function` class in the Python interface inherits from both `ufl.Coefficient` and `cpp.Function`, as illustrated by the following pseudo code:

Python code

```
class Function(ufl.Coefficient, cpp.Function):
    def __init__(self, V):
        ufl.Coefficient.__init__(self, V.ufl_element)
        cpp.Function().__init__(self, V)
```

The actual constructor also includes logic to instantiate a `Function` from other objects. A more elaborate logic is also included to handle access to subfunctions.

A user-defined `Expression` can be created in two different ways: (i) as a pure Python `Expression`; or (ii) as a JIT compiled `Expression`. A pure Python `Expression` is an object instantiated from a subclass of `Expression` in Python. The `Source` class above is an example of this. Pseudo code for the constructor of the `Expression` class is similar to that for the `Function` class:

Python code

```
class Expression(ufl.Coefficient, cpp.Expression):
    def __init__(self, element=None):
        if element is None:
            element = auto_select_element(self.value_shape())
        ufl.Coefficient.__init__(self, element)
        cpp.Expression(element.value_shape())
```

If the `ufl.FiniteElement` is not defined by the user, DOLFIN will automatically choose an element using the `auto_select_element` function. This function takes the value shape of the `Expression` as argument. This has to be supplied by the user for vector- or tensor-valued `Expressions`, by overloading the `value_shape` method. The base class `cpp.Expression` is initialized using the value shape of the `ufl.FiniteElement`.

The actual code is considerably more complex than indicated above, as the same class, `Expression`, is used to handle both JIT compiled and pure Python `Expressions`. Also note that the actual subclass is eventually generated by a *metaclass* in Python, which makes it possible to include sanity checks for the declared subclass.

The `cpp.Expression` class is wrapped by a so-called *director class* in the SWIG-generated C++ layer. This means that the whole Python class is wrapped by a C++ subclass of `dolfin::Expression`. Each virtual method of the C++ base class is implemented by the SWIG-generated subclass in C++. These methods call the Python version of the method, which the user eventually implements by subclassing `cpp.Expression` in Python.

Just-in-time compilation of expressions. The performance of a pure Python `Expression` may be sub-optimal because of the callback from C++ to Python each time the `Expression` is evaluated. To circumvent this, a user can instead subclass the C++ version of `Expression` using a JIT compiled `Expression`. Because the subclass is implemented in C++, it will not involve any callbacks to Python,

and can therefore be significantly faster than a pure Python `Expression`. A JIT compiled `Expression` is generated by passing a string of C++ code to the `Expression` constructor:

Python code

```
e = Expression("sin(x[0])")
```

The passed string is used to generate a subclass of `dolfin::Expression` in C++, where it is inlined into an overloaded `eval` method. The final code is JIT compiled and wrapped to Python using Instant (see Chapter 14). The generated Python class is then imported into Python. The class is not yet instantiated, as the final JIT compiled `Expression` also needs to inherit from `ufl.Coefficient`. To accomplish this, we dynamically create a class which inherits from both the generated class and `ufl.Coefficient`.

Classes in Python can be created during run-time by using the `type` function. The logic of creating a class and returning an instance of that class is handled in the `__new__` method of `dolfin.Expression`, as illustrated by the following pseudo code:

Python code

```
class Expression(object):
    def __new__(cls, cppcode=None):
        if cls.__name__ != "Expression":
            return object.__new__(cls)
        cpp_base = compile_expressions(cppcode)
        def __init__(self, cppcode):
            ...
        generated_class = type("CompiledExpression",
                               (Expression, ufl.Coefficient, cpp_base),
                               {"__init__": __init__})
        return generated_class()
```

The `__new__` method is called when a JIT compiled `Expression` is instantiated. However, it will also be called when a pure Python subclass of `Expression` is instantiated during initialization of the base-class. We handle the two different cases by checking the name of the instantiated class. If the name of the class is not `"Expression"`, then the call originates from the instantiation of a subclass of `Expression`. When a pure Python `Expression` is instantiated, like the `Source` instance in the code example above, the `__new__` method of `object` is called and the instantiated object is returned. In the other case, when a JIT compiled `Expression` is instantiated, we need to generate the JIT compiled base class from the passed Python string, as explained above. This is done by calling the function `compile_expressions`. Before `type` is called to generate the final class, an `__init__` method for the class is defined. This method initiates the new object by automatically selecting the element type and setting dimensions for the created `Expression`. This procedure is similar to what is done for the Python derived `Expression` class. Finally, we construct the new class which inherits the JIT compiled class and `ufl.Coefficient` by calling `type`.

The `type` function takes three arguments: the name of the class (`"CompiledExpression"`), the bases of the class (`Expression, ufl.Coefficient, cpp_base`), and a `dict` defining the interface (methods and attributes) of the class. The only new method or attribute we provide to the generated class is the `__init__` method. After the class is generated, we instantiate it and the object is returned to the user.

Assembly of UFL forms. The `assemble` function in the Python interface of DOLFIN enables a user to directly assemble a declared UFL form:

Python code

```
mesh = UnitSquare(8, 8)
V = FunctionSpace(mesh, "Lagrange", 1)
u = TrialFunction(V)
v = TestFunction(V)
c = Expression("sin(x[0])")
a = c*dot(grad(u), grad(v))*dx
A = assemble(a)
```

The `assemble` function is a thin wrapper layer around the wrapped `cpp.assemble` function. The following pseudo code illustrates what happens in this layer:

Python code

```
def assemble(form, tensor=None, mesh=None):
    dolfin_form = Form(form)
    if tensor is None:
        tensor = create_tensor(dolfin_form.rank())
    if mesh is not None:
        dolfin_form.set_mesh(mesh)
    cpp.assemble(dolfin_form, tensor)
    return tensor
```

Here, `form` is a `ufl.Form`, which is used to generate a `dolfin.Form`, as explained below. In addition to the `form` argument, a user can choose to provide a tensor and/or a mesh. If a tensor is not provided, one will automatically be generated by the `create_tensor` function. The optional mesh is needed if the form does not contain any `Arguments`, or `Functions`; for example when a functional containing only `Expressions` is assembled. Note that the length of the above signature has been shortened. Other arguments to the `assemble` function exist but are skipped here for clarity.

The following pseudo code demonstrates what happens in the constructor of `dolfin.Form`, where the base class `cpp.Form` is initialized from a `ufl.Form`:

Python code

```
class Form(cpp.Form):
    def __init__(self, form):
        compiled_form, form_data = jit(form)
        function_spaces = extract_function_spaces(form_data)
        coefficients = extract_coefficients(form_data)
        cpp.Form.__init__(self, compiled_form, function_spaces, coefficients)
```

The `form` is first passed to the `dolfin.jit` function, which calls the registered form compiler to generate code and JIT compile it. There are presently two form compilers that can be chosen: `"ffc"` and `"sfc"` (see Chapters 11 and 15). Each one of these form compilers defines its own `jit` function, which eventually will receive the call. The form compiler can be chosen by setting:

Python code

```
parameters["form_compiler"]["name"] = "sfc"
```

The default form compiler is `"ffc"`. The `jit` function of the form compiler returns the JIT compiled `ufc::form` together with a `ufl.FormData` object. The latter is a data structure containing metadata for the `ufl.form`, which is used to extract the function spaces and coefficients that are needed to instantiate a `cpp.Form`. The extraction of these data is handled by the `extract_function_spaces` and the `extract_coefficients` functions.

10.4.4 NumPy and SciPy integration

The values of the `Matrix` and `Vector` classes in the Python interface of DOLFIN can be viewed as NumPy arrays. This is done by calling the `array` method of the vector or matrix:

Python code

```
A  = assemble(a)
AA = A.array()
```

Here, `A` is a matrix assembled from the form `a`. The NumPy array `AA` is a dense structure and all values are copied from the original data. The `array` function can be called on a distributed matrix or vector, in which case it will return the locally stored values.

Direct access to linear algebra data. Direct access to the underlying data is possible for the uBLAS and MTL4 linear algebra backends. A NumPy array view into the data will be returned by the method `data`:

Python code

```
parameters["linear_algebra_backend"] = "uBLAS"
b = assemble(L)
bb = b.data()
```

Here, `b` is a uBLAS vector and `bb` is a NumPy view into the data of `b`. Any changes to `bb` will directly affect `b`. A similar method exists for matrices:

Python code

```
parameters["linear_algebra_backend"] = "MTL4"
A = assemble(a)
rows, columns, values = A.data()
```

The data is returned in a compressed row storage format as the three NumPy arrays `rows`, `columns` and `values`. These are also views of the data that represent `A`. Any changes in `values` will directly result in a corresponding change in `A`.

Sparse matrix and SciPy integration. The `rows`, `columns` and `values` data structures can be used to instantiate a `csr_matrix` from the `scipy.sparse` module (Jones et al., 2009):

Python code

```
from scipy.sparse import csr_matrix
rows, columns, values = A.data()
csr = csr_matrix((values, columns, rows))
```

The `csr_matrix` can then be used with other Python modules that support sparse matrices, such as the `scipy.sparse` module and `pyamg`, which is an algebraic multigrid solver (Bell et al., 2011).

Slicing vectors. NumPy provides a convenient slicing interface for NumPy arrays. The Python interface of DOLFIN also provides such an interface for vectors (see Chapter 19 for details of the implementation). A slice can be used to access and set data in a vector:

Python code

```
# Create copy of vector
b_copy = b[:]

# Slice assignment (c can be a scalar, a DOLFIN vector or a NumPy array)
```

```
b[:] = c

# Set negative values to zero
b[b < 0] = 0

# Extract every second value
b2 = b[::2]
```

A difference between a NumPy slice and a slice of a DOLFIN vector is that a slice of a NumPy array provides a view into the original array, whereas in DOLFIN we provide a copy. A list/tuple of integers or a NumPy array can also be used to both access and set data in a vector:

Python code

```
b1 = b[[0, 4, 7, 10]]
b2 = b[array((0, 4, 7, 10))]
```

10.5 Historical notes

The first public version of DOLFIN, version 0.2.0, was released in 2002. At that time, DOLFIN was a self-contained C++ library with minimal external dependencies. All functionality was then implemented as part of DOLFIN itself, including linear algebra and finite element form evaluation. Although only piecewise linear elements were supported, DOLFIN provided rudimentary automated finite element assembly of variational forms. The form language was implemented by C++ operator overloading. For an overview of the development of the FEniCS form language and an example of the early form language implemented in DOLFIN, see Chapter 11.

Later, parts of the functionality of DOLFIN have been moved to either external libraries or other FEniCS components. In 2003, the FEniCS project was born and shortly after, with the release of version 0.5.0 in 2004, the form evaluation system in DOLFIN was replaced by an automated code generation system based on FFC and FIAT. In the following year, the linear algebra was replaced by wrappers for PETSc data structures and solvers. At this time, the DOLFIN Python interface (PyDOLFIN) was introduced. Since then, the Python interface has developed from a simple auto-generated wrapper layer for the DOLFIN C++ functionality to a mature problem-solving environment with support for just-in-time compilation of variational forms and integration with external Python modules like NumPy.

In 2006, the DOLFIN mesh data structures were simplified and reimplemented to improve efficiency and expand functionality. The new data structures were based on a light-weight object-oriented layer on top of an underlying data storage by plain contiguous C/C++ arrays and improved the efficiency by orders of magnitude over the old implementation, which was based on a fully object-oriented implementation with local storage of all mesh entities like cells and vertices. The first release of DOLFIN with the new mesh library was version 0.6.2.

In 2007, the UFC interface was introduced and the FFC form language was integrated with the DOLFIN Python interface. Just-in-time compilation was also introduced. The following year, the linear algebra interfaces of DOLFIN were redesigned to allow flexible handling of multiple linear algebra backends. In 2009, a major milestone was reached when parallel computing was introduced in DOLFIN.

Over the years, DOLFIN has undergone a large number of changes to its design, interface and implementation. However, since the release of DOLFIN 0.9.0, which introduced a redesign of the DOLFIN function classes based on the new function space abstraction, only minor changes have been made to the interface. Since the release of version 0.9.0, most work has gone into refining the interface, implementing missing functionality, fixing bugs and improving documentation, in anticipation of the first stable release of DOLFIN, version 1.0.

11 FFC: the FEniCS form compiler

By Anders Logg, Kristian B. Ølgaard, Marie E. Rognes and Garth N. Wells

One of the key features of FEniCS is automated code generation for the general and efficient solution of finite element variational problems. This automated code generation relies on a form compiler for offline or just-in-time compilation of code for individual forms. Two different form compilers are available as part of FEniCS. This chapter describes the form compiler FFC. The other form compiler, SFC, is described in Chapter 15.

11.1 Compilation of variational forms

In simple terms, the solution of finite element variational problems is based on two ingredients: the assembly of linear or nonlinear systems of equations and the solution of those equations. As a result, many finite element codes are similar in their design and implementation. In particular, a central part of most finite element codes is the assembly of sparse matrices from finite element bilinear forms. In Chapter 6, we saw that one may formulate a general algorithm for assembly of sparse tensors from variational forms. However, this algorithm relies on the computation of the element tensor A_T as well as the local-to-global mapping ι_T. Both A_T and ι_T differ greatly between different finite elements and different variational forms. Special-purpose code is therefore needed. As a consequence, the code for computing A_T and ι_T must normally be developed by hand for a given application. This is both tedious and error-prone.

The issue of having to develop code for A_T and ι_T by hand can be resolved by a form compiler. A form compiler generates code for computing A_T and ι_T. This code may then be called by a general purpose routine for assembly of finite element matrices and vectors. In addition to reduced development time, performance may be improved by using code generation since the form compiler can generate efficient code for the computation of A_T by using optimization techniques that are not readily applicable if the code is developed by hand. In Chapters 7, 8 and 9, two different approaches to the optimized computation of the element tensor A_T are presented.

From an input describing a finite element variational problem in mathematical notation, the form compiler FFC generates code for the efficient computation of A_T and ι_T, as well as code for computing related quantities. More specifically, FFC takes as input a variational form specified in the UFL form language (described in Chapter 17) and generates as output C++ code that conforms to the UFC interface (described in Chapter 16). This process is illustrated schematically in Figure 11.1.

Figure 11.1: The form compiler FFC generates C++ code in UFC format from a given finite element variational form in UFL format.

11.2 Compiler interfaces

FFC provides three different interfaces: a Python interface, a command-line interface, and a just-in-time (JIT) compilation interface. The first two are presented here, while the third is discussed below in Section 11.7. Although FFC provides three different interfaces, many users are never confronted with any of these interfaces; Python users mostly rely on DOLFIN to handle the communication with FFC. The command-line interface is familiar for DOLFIN C++ users, who must call FFC on the command-line to generate code for inclusion in their C++ programs. The JIT interface is rarely called directly by users, but it is the main interface between DOLFIN and FFC, which allows DOLFIN to seamlessly generate and compile code when running solver scripts implemented in Python.

11.2.1 Python interface

The Python interface to FFC takes the form of a standard Python module. There are two main entry point functions to the functionality of FFC: `compile_form` and `compile_element`, to compile forms and elements, respectively.

The `compile_form` function provides the main functionality of FFC, which is to generate code for assembly of matrices and vectors (tensors) from finite element variational forms. The `compile_form` function expects a form or a list of forms as input along with a set of optional arguments:

Python code

```
compile_form(forms,
             object_names={},
             prefix="Form",
             parameters=default_parameters(),
             common_cell=None)
```

The above call generates UFC conforming code for each of the given forms and each of the finite elements involved in the definition of the forms, as well as their corresponding degree-of-freedom maps. The `object_names` dictionary is an optional argument that specifies the names of the coefficients that were used to define the form. This is used by the command-line interface of FFC to allow a user to refer to any coefficients in a form by their names (`f`, `g`, etc.). The `prefix` argument can be used to control the prefix of the output file containing the generated code; the default is "Form". The suffix ".h" will be added automatically. The optional argument `parameters` should be a Python dictionary with code generation parameters and is described further below. The optional argument `common_cell` is mostly useful from within DOLFIN and is there used to specify the domain (cell) of integration for forms containing partly undefined expressions.

Sometimes, it may be desirable to compile single elements, which means generating code for run-time evaluation of basis functions and other entities associated with the definition of a finite element. The `compile_element` function expects a finite element or a list of finite elements as its first argument. In addition, a set of optional arguments can be provided:

Python code

```
compile_element(elements,
                prefix="Element",
                parameters=default_parameters())
```

The above call generates UFC conforming code for the specified finite element spaces and their corresponding degree-of-freedom maps. The arguments `prefix` and `parameters` play the same role as for `compile_form`.

As an illustration, we list in Figure 11.2 the specification and compilation of a variational formulation of Poisson's equation in two dimensions using the Python interface. The last line calls

Figure 11.2: Compiling a form using the FFC Python interface.

Python code

```
from ufl import *
from ffc import *

element = FiniteElement("Lagrange", triangle, 1)
u = TrialFunction(element)
v = TestFunction(element)
f = Coefficient(element)

a = inner(grad(u), grad(v))*dx
L = f*v*dx

compile_form([a, L], prefix="Poisson")
```

the `compile_form` function. When run, code will be generated for the forms `a` and `L`, and the finite element and degree-of-freedom map associated with the element `element`, and then written to the file "Poisson.h". In Figure 11.3, we list (a part of) the generated C++ code for the input displayed in Figure 11.2.

In Figure 11.4, we list the specification and compilation of a piecewise continuous quartic finite element (Lagrange element of degree 4) in three dimensions using the FFC Python interface. The two first lines import the UFL and FFC modules respectively. The third line specifies the finite element in UFL syntax. The last line calls the FFC `compile_element` function. The generated code is written to the file `P4tet.h`, as specified by the argument `prefix`. In Figure 11.5, we list (a part of) the generated C++ code for the input displayed in Figure 11.4.

11.2.2 Command-line interface

The command-line interface takes a UFL form file or a list of form files as input:

Bash code

```
$ ffc FormFile.ufl
```

The form file should contain the specification of elements and/or forms in UFL syntax, and is very similar to the FFC Python interface, as illustrated by the following specification of the same variational problem as in Figure 11.2:

UFL code

```
element = FiniteElement("Lagrange", triangle, 1)
u = TrialFunction(element)
v = TestFunction(element)
f = Coefficient(element)

a = inner(grad(u), grad(v))*dx
L = f*v*dx
```

The contents of each form file are wrapped in a Python script and then executed. Such a script is simply a copy of the form file that includes the required imports of FFC and UFL and calls to `compile_element` or `compile_form` from the FFC Python interface. The variable names `a`, `L` and `element` are recognized as a bilinear form, a linear form and a finite element, respectively. In addition, FFC recognizes the variable name `M` as a functional.

C++ code

```
virtual void tabulate_tensor(double* A,
                             const double * const * w,
                             const ufc::cell& c) const
{
  // Extract vertex coordinates
  const double * const * x = c.coordinates;

  // Compute Jacobian of affine map from reference cell
  const double J_00 = x[1][0] - x[0][0];
  const double J_01 = x[2][0] - x[0][0];
  const double J_10 = x[1][1] - x[0][1];
  const double J_11 = x[2][1] - x[0][1];

  // Compute determinant of Jacobian
  double detJ = J_00*J_11 - J_01*J_10;

  // Compute inverse of Jacobian
  const double K_00 =  J_11 / detJ;
  const double K_01 = -J_01 / detJ;
  const double K_10 = -J_10 / detJ;
  const double K_11 =  J_00 / detJ;

  // Set scale factor
  const double det = std::abs(detJ);

  // Compute geometry tensor
  const double G0_0_0 = det*(K_00*K_00 + K_01*K_01);
  const double G0_0_1 = det*(K_00*K_10 + K_01*K_11);
  const double G0_1_0 = det*(K_10*K_00 + K_11*K_01);
  const double G0_1_1 = det*(K_10*K_10 + K_11*K_11);

  // Compute element tensor
  A[0] = 0.5*G0_0_0 + 0.5*G0_0_1 + 0.5*G0_1_0 + 0.5*G0_1_1;
  A[1] = -0.5*G0_0_0 - 0.5*G0_1_0;
  A[2] = -0.5*G0_0_1 - 0.5*G0_1_1;
  A[3] = -0.5*G0_0_0 - 0.5*G0_0_1;
  A[4] = 0.5*G0_0_0;
  A[5] = 0.5*G0_0_1;
  A[6] = -0.5*G0_1_0 - 0.5*G0_1_1;
  A[7] = 0.5*G0_1_0;
  A[8] = 0.5*G0_1_1;
}
```

Figure 11.3: Excerpt of the C++ code generated for the input listed in Figure 11.2. In this example, the element tensor is evaluated by computing a tensor contraction between a reference tensor A^0 (containing values that are either zero or 0.5) and the geometry tensor G_T computed based on geometrical data from the current cell. See Chapter 8 for further details.

Python code

```
from ufl import *
from ffc import *
element = FiniteElement("Lagrange", tetrahedron, 4)
compile_element(element, prefix="P4tet")
```

Figure 11.4: Compiling an element using the FFC Python interface.

Figure 11.5: Excerpt of the C++ code generated for the input listed in Figure 11.4. The evaluation of a basis function is a complex process that involves mapping the given point back to a reference cell and evaluating the given basis function as a linear combination of a special set of basis functions (the "prime basis") on the reference cell. The code generated by FFC is based on information given to FFC by FIAT at compile-time.

C++ code

```
virtual void evaluate_basis(unsigned int i,
                            double* values,
                            const double* coordinates,
                            const ufc::cell& c) const
{
  // Extract vertex coordinates
  const double * const * x = c.coordinates;

  // Compute Jacobian of affine map from reference cell
  const double J_00 = x[1][0] - x[0][0];
  const double J_01 = x[2][0] - x[0][0];
  const double J_02 = x[3][0] - x[0][0];
  const double J_10 = x[1][1] - x[0][1];
  const double J_11 = x[2][1] - x[0][1];
  [...]

  // Reset values
  *values = 0.0;
  switch (i)
  {
  case 0:
    {
      [...]

      // Declare helper variables
      double tmp0 = 0.5*(2.0 + Y + Z + 2.0*X);
      double tmp1 = 0.25*(Y + Z)*(Y + Z);
      double tmp2 = 0.5*(1.0 + Z + 2.0*Y);
      double tmp3 = 0.5*(1.0 - Z);
      double tmp4 = tmp3*tmp3;

      // Compute basisvalues
      basisvalues[0] = 1.0;
      basisvalues[1] = tmp0;
      basisvalues[4] = 1.5*tmp0*basisvalues[1]
                     - 0.5*tmp1*basisvalues[0];
      [...]

      // Table(s) of coefficients
      static const double coefficients0[35] = \
      {-0.0137464349807054, -0.0144900147488139, -
          0.00836581391578936, ...

      // Compute value(s)
      for (unsigned int r = 0; r < 35; r++)
      {
        *values += coefficients0[r]*basisvalues[r];
      }

      break;
    }
    [...]
  }
}
```

11.3 Parameters affecting code generation

The code generated by FFC can be controlled by a number of optional parameters. Through the Python interface, parameters are set in the dictionary `parameters` which is passed to the `compile` functions. The default values for these may be obtained by calling the function `default_parameters` from the Python interface. Most parameters can also be set on the command-line. All available command-line parameters are listed on the FFC manual page (`man ffc`). We here list some of the parameters which affect the code generation. We list the dictionary key associated with each parameter, and the command-line version in parentheses, if available.

`"format"` (`-l`) This parameter controls the output format for the generated code. The default value is "`ufc`", which indicates that the code is generated according to the UFC specification. Alternatively, the value "`dolfin`" may be used to generate code according to the UFC format with a small set of additional DOLFIN-specific wrappers.

`"representation"` (`-r`) This parameter controls the representation used for the generated element tensor code. There are three possibilities: "`auto`" (the default), "`quadrature`" and "`tensor`". See Section 11.5, and Chapters 7 and 8 for more details on the different representations. In the case "`auto`", either the quadrature or tensor representation is selected by FFC. FFC attempts to select the representation which will lead to the most efficient code for the given form.

`"split"` (`-f split`) This option controls the output of the generated code into a single or multiple files. The default is `False`, in which case the generated code is written to a single file. If set to `True`, separate header (.h) and implementation (`.cpp`) files are generated.

`"optimize"` (`-O`) This option controls code optimization features, and the default is `False`. If set to `True`, the code generated for the element tensor is optimized for run-time performance. The optimization strategy used depends on the chosen representation. In general, this will increase the time required for FFC to generate code, but should reduce the run-time for the generated code.

`"log_level"` This option controls the verbosity level of the compiler. The possible values are, in order of decreasing verbosity: `DEBUG`, `INFO` (default), `WARNING`, `ERROR` and `CRITICAL`.

11.4 Compiler design

FFC breaks compilation into several stages. The output generated at each stage serves as input for the following stage, as illustrated in Figure 11.6. We describe each of these stages below. The individual compiler stages may be accessed through the `ffc.compiler` module. We consider here only the stages involved when compiling forms. For compilation of elements a similar (but simpler) set of stages is used.

Compiler stage 0: Language (parsing). In this stage, the user-specified form is interpreted and stored as a UFL abstract syntax tree (AST). The actual parsing is handled by Python and the transformation to a UFL form object is implemented by operator overloading in UFL.

Input: Python code or `.ufl` file
Output: UFL form

Compiler stage 1: Analysis. This stage preprocesses the UFL form and extracts form metadata (`FormData`), such as which elements were used to define the form, the number of coefficients and the cell type (intervals, triangles or tetrahedra). This stage also involves selecting a suitable

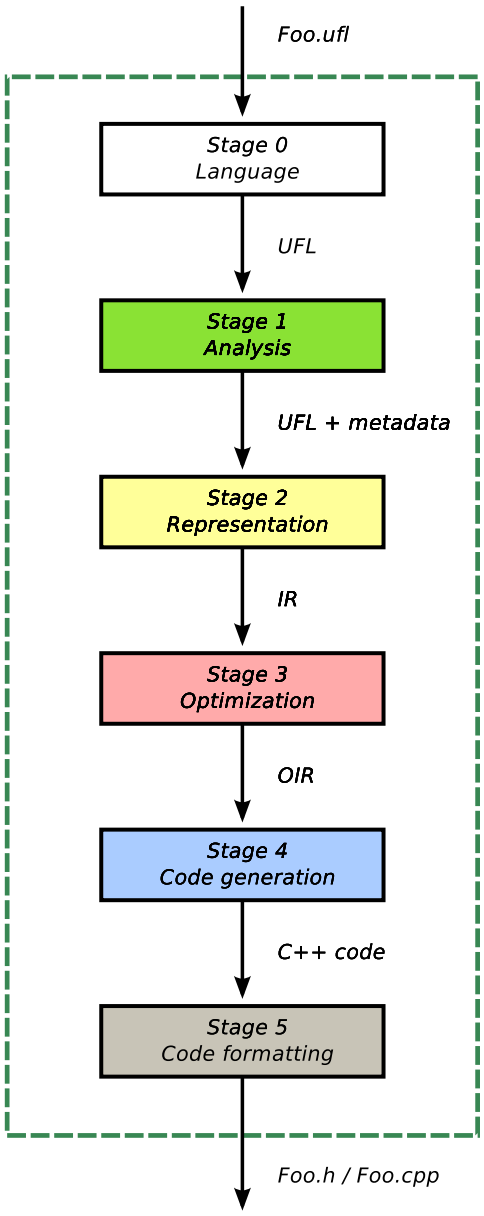


Figure 11.6: Form compilation broken into six sequential stages: Language, Analysis, Representation, Optimization, Code generation and Code Formatting. Each stage generates output based on input from the previous stage. The input/output data consist of a UFL form file (in the case of calling FFC from the command-line), a UFL object, a UFL object and metadata computed from the UFL object, an intermediate representation (IR), an optimized intermediate representation (OIR), C++ code and, finally, C++ code files.

representation for the form if that has not been specified by the user (see Section 11.5 below).

Input: UFL form
Output: preprocessed UFL form and form metadata

Compiler stage 2: Code representation. This stage examines the input and generates all data needed for the code generation. This includes generation of finite element basis functions, extraction of data for mapping of degrees of freedom, and possible precomputation of integrals. Most of the complexity of compilation is handled in this stage.

The intermediate representation is stored as a dictionary, mapping names of UFC functions to the data needed for generation of the corresponding code. In simple cases, like `ufc::form::rank`, this data may be a simple number like `2`. In other cases, like `ufc::cell_tensor::tabulate_tensor`, the data may be a complex data structure that depends on the choice of form representation.

Input: preprocessed UFL form and form metadata
Output: intermediate representation (IR)

Compiler stage 3: Optimization. This stage examines the intermediate representation and performs optimizations. Such optimization may involve FErari based optimizations as discussed in Chapter 12 or symbolic optimization as discussed in Chapter 7. Data stored in the intermediate representation dictionary is then replaced by new data that encode an optimized version of the function in question.

Input: intermediate representation (IR)
Output: optimized intermediate representation (OIR)

Compiler stage 4: Code generation. This stage examines the optimized intermediate representation and generates the actual C++ code for the body of each UFC function. The code is stored as a dictionary, mapping names of UFC functions to strings containing the C++ code. As an example, the data generated for `ufc::form::rank` may be the string "`return 2;`".

We emphasize the importance of separating stages 2, 3 and 4. This allows stages 2 and 3 to focus on algorithmic aspects related to finite elements and variational forms, while stage 4 is concerned only with generating C++ code from a set of instructions prepared in earlier compilation stages.

Input: optimized intermediate representation (OIR)
Output: C++ code

Compiler stage 5: Code formatting. This stage examines the generated C++ code and formats it according to the UFC format, generating as output one or more `.h/.cpp` files conforming to the UFC specification. This is where the actual writing of C++ code takes place. This stage relies on templates for UFC code available as part of the UFC module `ufc_utils`.

Input: C++ code
Output: C++ code files

11.5 Form representation

Two different approaches to code generation are implemented in FFC. One based on traditional quadrature and another on a special tensor representation. We address these representations here briefly and refer readers to Chapter 7 for details of the quadrature representation and to Chapter 8 for details of the tensor representation.

11.5.1 Quadrature representation

The quadrature representation in FFC is selected using the option `-r quadrature`. As the name suggests, the method to evaluate the local element tensor A_T involves a loop over integration points and adding the contribution from each point to A_T. To generate code for quadrature, FFC calls FIAT during code generation to tabulate finite element basis functions and their derivatives at a suitable set of quadrature points on the reference element. It then goes on to generate code for computing a weighted average of the integrand defined by the UFL AST at these quadrature points.

11.5.2 Tensor representation

When FFC is called with the `-r tensor` option, it attempts to extract a monomial representation of the given UFL form; that is, rewrite the given form as a sum of products of basis functions and their derivatives. Such a representation is not always possible, in particular if the form is expressed using operators other than addition, multiplication and linear differential operators. If unsuccessful, FFC falls back to using quadrature representation.

If the transformation is successful, FFC computes the tensor representation $A_T = A^0 : G_T$, as described in Chapter 8, by calling FIAT to compute the reference tensor A^0. Code is then generated for computing the element tensor. Each entry of the element tensor is obtained by computing an inner product between the geometry tensor G_T and a particular slice of the reference tensor. It should be noted that the entries of the reference tensor are known during code generation, so these numbers enter directly into the generated code.

11.5.3 Automatic selection of representation

If the user does not specify which representation to use, FFC will try to automatically select the "best" representation; that is, the representation that is believed to yield the best run-time performance. As described in Chapter 7, the run-time performance depends on many factors and it might not be possible to give a precise *a priori* answer as to which representation will be best for a particular variational form. In general, the more complex the form (in terms of the number of derivatives and the number of function products), the more likely quadrature is to be preferable. See Ølgaard and Wells (2010) for a detailed discussion on form complexity and comparisons between tensor and quadrature representations. In Ølgaard and Wells (2010), it was suggested that the selection should be based on an estimate of the operation count to compute the element tensor A_T. However, it turns out to be difficult to obtain an estimate that is accurate enough for this purpose. Therefore, the following crude strategy to select the representation has been implemented. First, FFC will try to generate the tensor representation and in case it fails, quadrature representation will be selected. If the tensor representation is generated successfully, each monomial is investigated and if the number of coefficients plus derivatives is greater than three, then quadrature representation is selected.

11.6 Optimization

The optimization stage of FFC is concerned with the run-time efficiency of the generated code for computing the local finite element tensor. Optimization is available for both quadrature and tensor representations, and they both operate on the intermediate representation generated in stage two. The output in both cases is a new set of instructions (an optimized intermediate representation) for the code generation stage. The goal of the optimization is to reduce the number of operations needed to compute the element tensor A_T.

Due to the dissimilar nature of the quadrature and tensor representations, the optimizations applied to the two representations are different. To optimize the tensor representation, FFC relies on

the Python module FErari (see Chapter 12) to perform the optimizations. Optimization strategies for the quadrature representation are implemented as part of the FFC module itself and are described in Chapter 7. For both representations, the optimizations come at the expense of an increased generation time for FFC and for very complicated variational forms, hardware limitations can make the compilation impossible.

Optimizations are switched on by using the command-line option `-O` or through the Python interface by setting the parameter `optimize` equal to `True`. For the quadrature representation, there exist four optimization strategy options, and these can be selected through the command-line interface by giving the additional options `-f eliminate_zeros`, `-f simplify_expressions`, `-f precompute_ip_const` and `-f precompute_basis_const`, and through the Python interface by setting these parameters equal to `True` in the options dictionary. The option `-f eliminate_zeros` can be combined with any of the other three options. Only one of the optimizations `-f simplify_expressions`, `-f precompute_ip_const` and `-f precompute_basis_const` can be switched on at one time, and if two are given `-f simplify_expressions` takes precedence over `-f precompute_ip_const` which in turn takes precedence over the option `-f precompute_basis_const`. If no specific optimization options are given; that is, only `-O` is specified, the default is to switch on the optimizations `-f eliminate_zeros` and `-f simplify_expressions`.

11.7 Just-in-time compilation

FFC can also be used as a just-in-time (JIT) compiler. In a scripted environment, UFL objects can be passed to FFC, and FFC will return Python modules. Calling the JIT compiler involves calling the `jit` function available as part of the FFC Python module:

Python code

```
(compiled_object, compiled_module, form_data, prefix) \
    = jit(ufl_object, parameters=None, common_cell=None)
```

where `ufl_object` is either a UFL form or finite element object, `parameters` is an optional dictionary containing form compiler parameters and `common_cell` is an optional argument. The `common_cell` argument may be used to specify the cell (interval, triangle or tetrahedron) when the cell is not specified as part of the form[1]. The `jit` function returns a tuple, where `compiled_form` is a Python object which wraps either `ufc::form` or `ufc::finite_element` (depending on the type of UFL object passed to the form compiler), `compiled_module` is a Python module which wraps all the generated UFC code (this includes finite elements, degree of freedom maps, etc.), `form_data` is a UFL object that contains form metadata such as the number of coefficient functions in a form, and `prefix` is a string identifier for the form.

When the JIT compiler is called, FFC generates internally UFC code for the given form or finite element, compiles the generated code using a C++ compiler, and then wraps the result as a Python module using SWIG and Instant (see Chapter 14). The returned objects are ready to be used from Python. The generated and wrapped code is cached by the JIT compiler, so if the JIT compiler is called twice for the same form or finite element, the cached version is used. The cache directory is created by Instant, and can be cleaned by running the command `instant-clean`. The interactions of various components in the JIT process are illustrated in Figure 11.7.

The Python interface of DOLFIN makes extensive use of JIT compilation. It makes it possible to combine the performance features of generated C++ code with the ease of a scripted interface.

[1] This is used by DOLFIN to allow simple specification of expressions such as `f = Expression("sin(x[0])")` where the choice of cell type is not specified as part of the expression.

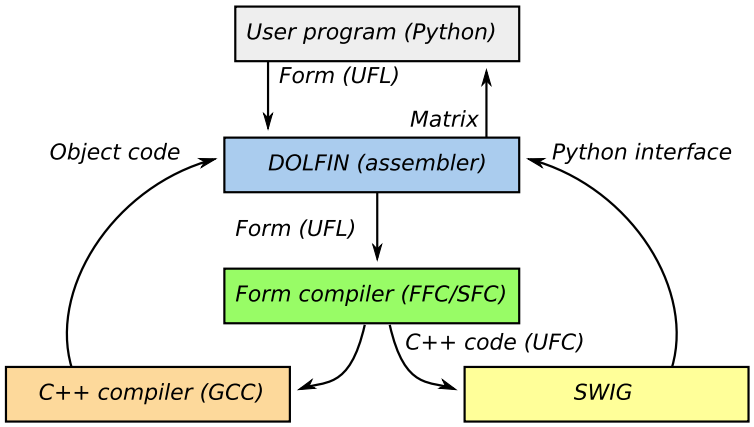


Figure 11.7: JIT compilation of variational forms coordinated by DOLFIN, and relying on UFL, FFC, UFL, SWIG, and GCC.

11.8 Extending FFC

FFC may be extended to add support for other languages, architectures and code generation techniques. For code that conforms to the UFC interface specification, only compiler stage 4 is affected. In this stage, the compiler needs to translate the intermediate representation of the form into actual C++ code that will later be formatted as part of UFC C++ classes and functions. Possible extensions in this stage of the compilation process can be to replace loops by special-purpose library calls (like low-level BLAS calls), SSE instructions or code targeted for graphical processing units (GPU).

Functionality that requires extending the UFC interface is usually handled by adding new experimental virtual (but non-abstract) functions[2] to the UFC interface, which may later be proposed to be included in the next stable specification of the UFC interface. Extensions to other languages are also possible by replacing the UFC code generation templates.

11.9 Historical notes

FFC was first released in 2004 as a research code capable of generating C++ code for simple variational forms (Kirby and Logg, 2006, 2007). Ever since its first release, FFC has relied on FIAT as a backend for computing finite element basis functions. In 2005, the DOLFIN assembler was redesigned to rely on code generated by FFC at compile-time for evaluation of the element tensor. Earlier versions of DOLFIN were based on a run-time system for evaluation of variational forms in C++ via operator overloading; see Figures 11.8–11.10.

Important milestones in the development of FFC include support for mixed elements (2005), FErari-based optimizations (2006), JIT compilation (2007), discontinuous Galerkin methods (2007) (Ølgaard et al., 2008), $H(\mathrm{div})/H(\mathrm{curl})$ elements (2007–2008) (Rognes et al., 2009), code generation based on quadrature (2007) (Ølgaard and Wells, 2010), the introduction of the UFC interface (2007), and optimized quadrature code generation (2008). In 2009, the FFC form language was replaced by the new UFL form language. In 2011, support was added for automated generation of dual problems and *a posteriori* error estimators (Rognes and Logg, 2011).

[2]The functions are made virtual, but non-abstract to ensure backwards compatibility with old generated code.

C++ code

```
class Poisson : public PDE
{
public:

  Poisson(Function& source) : PDE(3)
  {
    add(f, source);
  }

  real lhs(const ShapeFunction& u,
           const ShapeFunction& v)
  {
    return (grad(u), grad(v))*dx;
  }

  real rhs(const ShapeFunction& v)
  {
    return f*v*dx;
  }

private:

  ElementFunction f;

};
```

Figure 11.8: Implementation of Poisson's equation in DOLFIN 0.5.2 using C++ operator overloading. Note the use of `operator,` for inner product.

Python code

```
name = "Poisson"
element = FiniteElement("Lagrange", "triangle", 1)

v = BasisFunction(element)
u = BasisFunction(element)
f = Function(element)

a = v.dx(i)*u.dx(i)*dx
L = v*f*dx
```

Figure 11.9: Implementation of Poisson's equation in DOLFIN 0.5.3 using the new FFC form language. Note that the `grad` operator was missing in FFC at this time. It was also at this time that the test and trial functions changed places.

UFL code

```
element = FiniteElement("Lagrange", triangle, 1)

u = TrialFunction(element)
v = TestFunction(element)
f = Coefficient(element)

a = inner(grad(u), grad(v))*dx
L = f*v*dx
```

Figure 11.10: Implementation of Poisson's equation in DOLFIN 1.0 using the new UFL form language which was introduced in FFC 0.6.2. The order of trial and test functions has been restored.

12 *FErari: an optimizing compiler for variational forms*

By Robert C. Kirby and Anders Logg

In Chapter 8, we presented a framework for efficient evaluation of multilinear forms based on expressing the multilinear form as a special tensor contraction. This allows generation of efficient low-level code for assembly of a range of multilinear forms. Moreover, in Chapter 9 it was shown that the tensor contraction may sometimes possess a special structure that allows the contraction to be performed in a reduced number of arithmetic operations. This has led to the FErari project (Kirby et al., 2005, 2006; Kirby and Scott, 2007; Kirby and Logg, 2008), which provides an option within the form compiler FFC described in Chapter 11 to apply graph-based optimizations at compile-time. In this chapter, we describe the interface between FFC and FErari and present empirical results indicating the practical effect of the FErari optimizations on run-time evaluation of variational forms. In particular, we study the effect of optimizations on the run-time cost of forming the cell tensor A_T defined in Chapter 5.

Before proceeding, it is important to put these optimizations in the proper context. While FErari does not reduce the overall order of complexity of finite element calculations, it provides a practical benefit of reducing run-time from a few percent to sometimes tens of percent. Viewed as a domain-specific compiler optimization, this is quite respectable.

12.1 Optimized form compilation

FFC supports two different modes of code generation depending on how the multilinear form is represented. A user may select the tensor representation $A_T = A^0 : G_T$ discussed in Chapter 8 by supplying the `-r tensor` option to FFC, or alternatively select quadrature representation by supplying the `-r quadrature` option. While running in tensor mode, FFC constructs the reference tensor A^0 and generates code for contracting it with G_T. Sometimes, the form is expressed as a sum of tensor contractions. FFC then generates code for computing a sum of tensor contractions. When optimizations are enabled (using the `-O` option), the standard code generator for $A^0 : G_T$ is bypassed. The reference tensor A^0 is then passed to FErari. Initially, FErari computes a graph indicating relationships between the elements of A_T based on the entries of A^0 as described in Chapters 8 and 9. The edges are annotated with the cost of the calculation and the type of dependency such as collinearity or Hamming distance. Then, this graph is sequenced by topological sorting so that the entries of A_T appear after those upon which they depend. The edge annotations are then used by FFC to generate straight-line code for evaluating each entry of A_T. In Figures 12.1 and 12.2, we display the code generated by FFC for evaluation of the cell tensor A_T for Poisson's equation using standard and optimized tensor representation, respectively.

C++ code

```
/// Tabulate the tensor for the contribution from a local cell
virtual void tabulate_tensor(double* A,
                             const double * const * w,
                             const ufc::cell& c) const
{
  [...]

  // Extract vertex coordinates
  const double * const * x = c.coordinates;

  // Compute Jacobian of affine map from reference cell
  const double J_00 = x[1][0] - x[0][0];
  const double J_01 = x[2][0] - x[0][0];
  const double J_10 = x[1][1] - x[0][1];
  const double J_11 = x[2][1] - x[0][1];

  // Compute determinant of Jacobian
  double detJ = J_00*J_11 - J_01*J_10;

  // Compute inverse of Jacobian
  const double K_00 =  J_11 / detJ;
  const double K_01 = -J_01 / detJ;
  const double K_10 = -J_10 / detJ;
  const double K_11 =  J_00 / detJ;

  // Set scale factor
  const double det = std::abs(detJ);

  // Compute geometry tensor
  const double G0_0_0 = det*(K_00*K_00 + K_01*K_01);
  const double G0_0_1 = det*(K_00*K_10 + K_01*K_11);
  const double G0_1_0 = det*(K_10*K_00 + K_11*K_01);
  const double G0_1_1 = det*(K_10*K_10 + K_11*K_11);

  // Compute element tensor
  A[0] = 0.5*G0_0_0 + 0.5*G0_0_1 + 0.5*G0_1_0 + 0.5*G0_1_1;
  A[1] = -0.5*G0_0_0 - 0.5*G0_1_0;
  A[2] = -0.5*G0_0_1 - 0.5*G0_1_1;
  A[3] = -0.5*G0_0_0 - 0.5*G0_0_1;
  A[4] = 0.5*G0_0_0;
  A[5] = 0.5*G0_0_1;
  A[6] = -0.5*G0_1_0 - 0.5*G0_1_1;
  A[7] = 0.5*G0_1_0;
  A[8] = 0.5*G0_1_1;
}
```

Figure 12.1: Code generated by FFC for evaluation of the cell tensor for the Laplacian using piecewise linears on triangles (standard tensor representation). The first part of the code is standard non-optimized code for computing the entries of the geometry tensor based on coordinate data (inverse of the Jacobian). The second part (computing the cell tensor) is the FFC generated non-optimized tensor contraction for the Laplacian.

Figure 12.2: Code generated by FFC for evaluation of the cell tensor for the Laplacian using piecewise linears on triangles (FErari optimized tensor representation).

C++ code

```
virtual void tabulate_tensor(double* A,
                             const double * const * w,
                             const ufc::cell& c) const
{
  [...]

  // ... omitting identical code for geometry tensor

  A[1] = -0.5*G0_0_0 - 0.5*G0_1_0;
  A[5] = 0.5*G0_0_1;
  A[0] = -A[1] + 0.5*G0_0_1 + 0.5*G0_1_1;
  A[7] = 0.5*G0_1_0;
  A[6] = -A[7] - 0.5*G0_1_1;
  A[8] = 0.5*G0_1_1;
  A[2] = -A[8] - 0.5*G0_0_1;
  A[4] = 0.5*G0_0_0;
  A[3] = -A[4] - 0.5*G0_0_1;
}
```

12.2 Performance of optimizations

Now, we turn to the practical effect of using these optimizations. Several things are to be observed. First, running FErari within FFC leads to significantly increased times to generate the C++ code. Part of this increase results from a naive Python implementation of graph optimizations as part of FErari. Similar optimizations in Wolf and Heath (2009) have been implemented in C++ and run quite fast. Moreover, the code generated by FErari/FFC is itself quite large since one line of code is generated for each entry of A_T. It is often significantly larger than the code generated using quadrature, but marginally smaller than the standard tensor-contraction code generated by FFC. Because the generated C++ source code is quite large, it is also expensive to compile to machine code, both in terms of memory usage and CPU time. In situations where the source code size and compile-time are paramount, the quadrature mode of FFC is a better choice.

On the other hand, once the code is actually generated and compiled, we find modest improvements in its execution time. We compare below FErari-optimized code to standard tensor contraction, which we denote by the corresponding FFC command-line options `-r tensor -O` and `-r tensor`, respectively. FFC may also generate code based on quadrature, with and without optimization as discussed in Chapter 7. These options are denoted by `-r quadrature -O` and `-r quadrature`, respectively. All calculations were performed using FErari 0.2.0 and FFC 0.9.2 on a system running Ubuntu GNU/Linux 10.04 with an Intel 2.83 GHz quad core processor and 16 GB of RAM. The benchmarks may be repeated by running the script `bench/bench.py` available as part of FFC. The C++ compiler used was GCC 4.4.3 without any optimization flags. The reported timings are the CPU time in seconds for computing the cell tensor A_T.

12.2.1 Mass matrix for H^1

We consider forming the standard mass matrix on triangles defined by the bilinear form

$$a(u, v) = \int_\Omega uv \, \mathrm{d}x, \tag{12.1}$$

where we use Lagrange basis functions of orders one through five. The timing results, as well as speedup relative to non-optimized quadrature, are shown in Figure 12.3. As can be seen, tensor

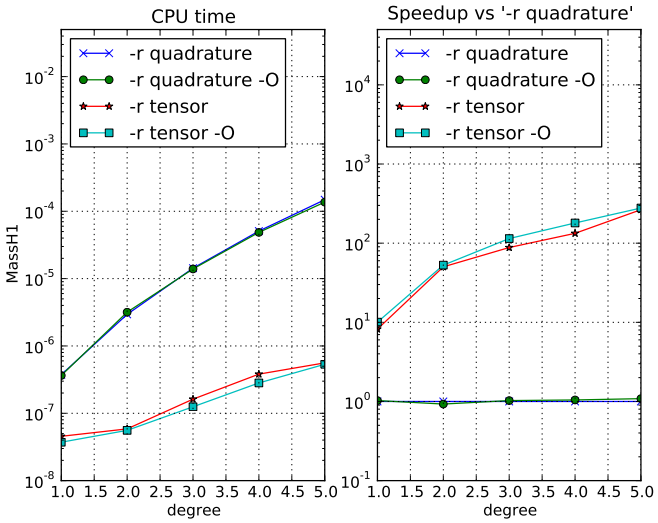


Figure 12.3: Speedup results for two-dimensional mass matrix using Lagrange polynomials.

contraction is to be preferred over quadrature for this form (each cell tensor is a scaled version of the reference tensor), and FErari optimizations accelerate the calculation over tensor contraction by up to about 10%.

12.2.2 *Stiffness matrix for H^1*

Next, we consider the stiffness matrix on triangles defined by

$$a(u, v) = \int_\Omega \nabla u \cdot \nabla v \, \mathrm{d}x, \tag{12.2}$$

again using Lagrange elements of orders one through five. The speedup results for this case are shown in Figure 12.4.

Again, we see that tensor contraction is preferred to quadrature for this form. Unlike the mass matrix, we find that FErari optimizations yield little result in the lowest order cases, but improve significantly as the degree increases.

12.2.3 *Variable coefficient stiffness matrix*

We also consider the stiffness matrix with a variable coefficient,

$$a(w; u, v) = \int_\Omega w \nabla u \cdot \nabla v \, \mathrm{d}x, \tag{12.3}$$

where w lies in the same polynomial space as u and v; that is, Lagrange elements of orders one through five. The speedup results are shown in Figure 12.5.

The difference between quadrature and tensor methods is smaller than for the bilinear case with no coefficient, but tensor contraction is still faster. FErari improves the tensor contraction by about 5-20% in each case.

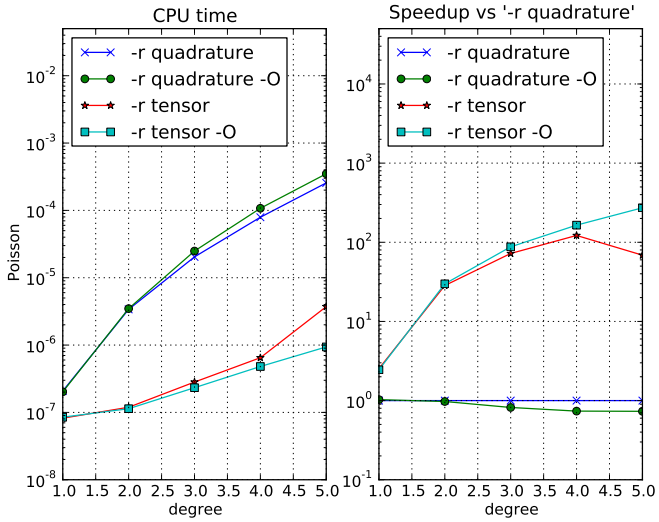


Figure 12.4: Speedup results for two-dimensional stiffness matrix using Lagrange polynomials.

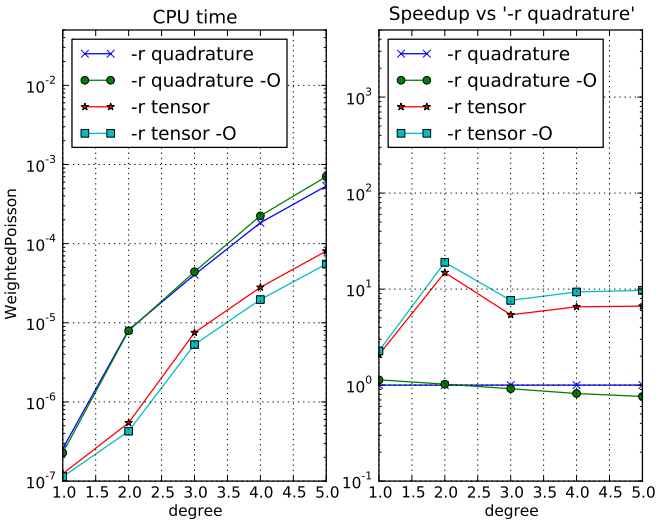


Figure 12.5: Speedup results for two-dimensional variable coefficient stiffness matrix using Lagrange polynomials.

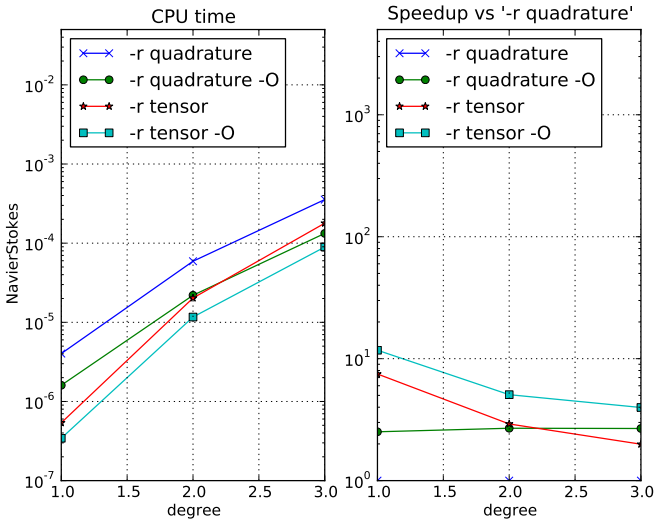


Figure 12.6: Speedup results for the two-dimensional convective term in Navier–Stokes using Lagrange polynomials.

12.2.4 *Navier–Stokes convective term*

Another problem where a variable coefficient taken from a finite element space naturally arises is the Navier–Stokes equations. For typical linearizations, one must evaluate the matrix associated with the form

$$a(w, \rho; u, v) = \int_T \rho \nabla u \, w \cdot v \, \mathrm{d}x, \tag{12.4}$$

where w is taken from the same finite element space as u and v, namely vector-valued piecewise polynomials. The function ρ is a scalar-valued polynomial of the same degree as the other functions. Such a function ρ will appear when one solves problems with a spatially variable fluid density.

This problem is far more challenging than the previous ones and we only consider up to cubic functions (not to exhaust system resources). The two coefficient functions w and ρ tend to make the quadrature-based methods more competitive with tensor contraction. Still, even for this more complicated form, FErari delivered on the order of 10% speedup over the tensor-based method and outperforms quadrature.

12.2.5 *Mass matrices for $H(\mathrm{div})$ and $H(\mathrm{curl})$*

Next we consider again the mass matrix (12.1), but for $H(\mathrm{div})$ and $H(\mathrm{curl})$ elements. For a discussion of the treatment of the required Piola transforms, see Rognes et al. (2009). In these cases, the Piola transforms make the computational pattern similar to the H^1 stiffness matrix, but with different numerical values in the reference tensor and hence potentially different speedup results for FErari. We consider the Brezzi–Douglas–Marini elements of orders one through five for $H(\mathrm{div})$ and the first kind Nédélec elements for $H(\mathrm{curl})$. The speedup plots are posted in Figures 12.7 and 12.8.

Tensor contraction methods outperform quadrature methods for these forms. For the $H(\mathrm{div})$ case, speedup of FErari over standard tensor contraction ranges from a few percent to nearly a factor of two. However, for $H(\mathrm{curl})$, FErari offers very little speedup.

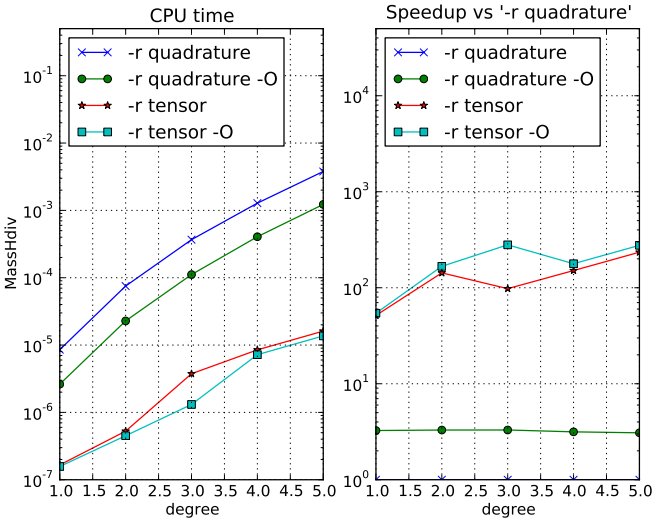


Figure 12.7: Speedup results for two-dimensional $H(\mathrm{div})$ mass matrix using BDM elements.

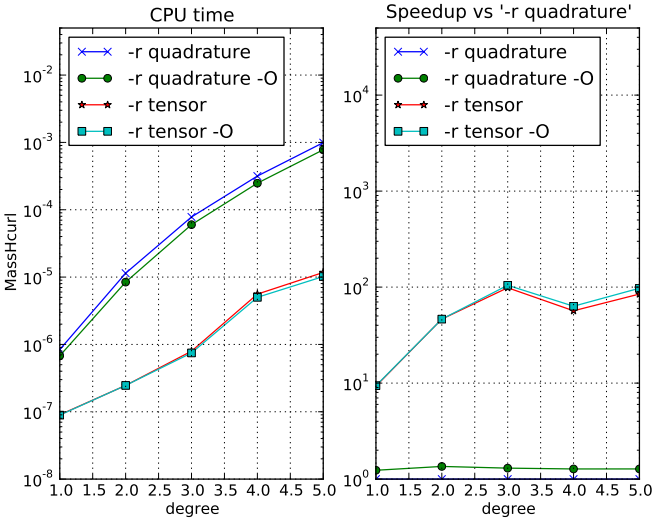


Figure 12.8: Speedup results for two-dimensional $H(\mathrm{curl})$ mass matrix using Nédélec elements.

12.3 Conclusions

We have studied a range of forms of various complexity. In most cases, FErari-based optimizations provide modest to considerable speedup in the run-time evaluation of variational forms. On the other hand, they can greatly increase the time FFC requires to generate code and so are less suitable for a development phase or a just-in-time compilation strategy. As a general guideline, one may also state that quadrature becomes more efficient relative to tensor contraction when the complexity of a form increases as measured in the number of coefficients and the number of differential operators, while the tensor contraction approach is relatively more efficient for simple forms and high order polynomials. Moreover, the construction of cell tensors is only part of the overall consideration in making finite element methods efficient.

12.4 Historical notes

Support for FErari optimizations was introduced in FFC version 0.3.2 in 2006 but was lost in a later rewrite of FFC. Starting with FErari 0.2.0 and FFC 0.9.1, which were released in 2010, FErari optimizations are again supported in FFC.

13 FIAT: numerical construction of finite element basis functions

By Robert C. Kirby

The FIAT project (Kirby, 2004, 2006a) implements the mathematical framework described in Chapter 4 as a Python package, working mainly in terms of numerical linear algebra. Although an implementation in floating-point arithmetic presents some challenges relative to symbolic computation, it can allow greater efficiency in terms of work and memory usage, especially for high order elements. To obtain efficiency in Python, the compute-intensive operations are expressed in terms of numerical linear algebra and performed using the widely distributed NumPy package. FIAT is one of the first FEniCS projects, providing the basis function back-end for FFC and enabling high-order H^1, $H(\mathrm{div})$ and $H(\mathrm{curl})$ elements.

This chapter works in the context of a Ciarlet triple $(T, \mathcal{V}, \mathcal{L})$ (Ciarlet, 2002), where T is a fixed reference domain, typically a triangle or tetrahedron, $\mathcal{V}$ is a finite-dimensional polynomial space, though perhaps vector- or tensor-valued and not coincident with polynomials of some fixed degree, and $\mathcal{L} = \{\ell_i\}_{i=1}^{|\mathcal{V}|}$ is a set of linear functionals spanning $\mathcal{V}'$. Recalling Chapter 4, the goal is first to enumerate a convenient basis $\{\phi_i\}_{i=1}^{|\mathcal{V}|}$ for $\mathcal{V}$ and then to form a generalized Vandermonde system

$$VA = I, \tag{13.1}$$

where $V_{ij} = \ell_i(\phi_j)$. Of course, forming this matrix requires some calculations, and we will discuss this further in a later section. The columns of $A = V^{-1}$ store the expansion coefficients of the nodal basis for $(T, \mathcal{V}, \mathcal{L})$ in terms of some other basis $\{\phi_i\}$.

13.1 Prime basis: collapsed-coordinate polynomials

High order polynomials in floating-point arithmetic require stable evaluation algorithms. FIAT uses the so-called collapsed-coordinate polynomials (Karniadakis and Sherwin, 2005) on the triangle and tetrahedra. Let $P_i^{\alpha,\beta}(x)$ denote the Jacobi polynomial of degree i with weights α and β. On the triangle T with vertices $(-1,-1)$, $(1,-1)$, $(-1,1)$ and Cartesian coordinates x and y, the polynomials are of the form

$$D^{p,q}(x,y) = P_p^{0,0}(\eta_1)\left(\frac{1-\eta_2}{2}\right)^p P_q^{2p+1,0}(\eta_2). \tag{13.2}$$

Here, η_1 and η_2 are the Cartesian coordinates on the biunit square, and the so-called collapsed-coordinate mapping

$$\eta_1 = 2\left(\frac{1+x}{1-y}\right) - 1$$
$$\eta_2 = y$$

maps from the triangle to the square. The set $\{D^{p,q}(x,y)\}_{p,q\geqslant 0}^{p+q\leqslant n}$ forms a basis for polynomials of degree n). Moreover, they are orthogonal in the $L^2(T)$ inner product. Recently, it has been shown that these polynomials may be computed directly on the triangle without reference to the singular mapping (Kirby, 2010a). This means that no special treatment of the singular point is required, allowing use of standard automatic differentiation techniques to compute derivatives.

The recurrences are obtained by rewriting the polynomials as

$$D^{p,q}(x,y) = \chi^p(x,y)\psi^{p,q}(y),$$

where

$$\chi^p(x,y) = P_p^{0,0}(\eta_1)\left(\frac{1-\eta_2}{2}\right)^p$$

and

$$\psi^{p,q}(y) = P_q^{2p+1,0}(\eta_2) = P_q^{2p+1,0}(y).$$

This representation is not separable in η_1 and η_2, which may seem to be a drawback to readers familiar with the usage of these polynomials in spectral methods. However, they do still admit sum-factorization techniques. More importantly for present purposes, each χ^p is in fact a polynomial in x and y and may be computed by recurrence. $\psi^{p,q}$ is just a Jacobi polynomial in y and so has a well-known three-term recurrence. The recurrences derived in Kirby (2010a) are presented in Algorithm 4, where, the coefficients $a_n^{\alpha,\beta}, b_n^{\alpha,\beta}, c_n^{\alpha,\beta}$ refer to those used in the Jacobi polynomial recurrences.

$$\begin{aligned} a_n^{\alpha,\beta} &= \frac{(2n+1+\alpha+\beta)(2n+2+\alpha+\beta)}{2(n+1)(n+1+\alpha+\beta)} \\ b_n^{\alpha,\beta} &= \frac{(\alpha^2-\beta^2)(2n+1+\alpha+\beta)}{2(n+1)(2n+\alpha+\beta)(n+1+\alpha+\beta)} \\ c_n^{\alpha,\beta} &= \frac{(n+\alpha)(n+\beta)(2n+2+\alpha+\beta)}{(n+1)(n+1+\alpha+\beta)(2n+\alpha+\beta)}. \end{aligned} \tag{13.3}$$

13.2 Representing polynomials and functionals

Even using recurrence relations and NumPy vectorization for arithmetic, further care is required to optimize performance. In this section, standard operations on polynomials will be translated into vector operations, and then batches of such operations cast as matrix multiplication. This helps eliminate the interpretive overhead of Python by moving numerical computation into optimized library routines, since `numpy.dot` wraps level 3 BLAS and other functions such as `numpy.svd` wrap relevant LAPACK routines.

Since polynomials and functionals over polynomials both form vector spaces, it is natural to represent each of them as vectors containing expansion coefficients in some basis. So, let $\{\phi_i\}$ be the Dubiner polynomials described above, where we have assumed some linear indexing of the Dubiner

Algorithm 4 Compute all triangular orthogonal polynomials up to degree d by recurrence

$D^{0,0}(x,y) := 1$
$D^{1,0}(x,y) := \frac{1+2x+y}{2}$
for $p \leftarrow 1, d-1$ **do**
    $D^{p+1,0}(x,y) := \left(\frac{2p+1}{p+1}\right)\left(\frac{1+2x+y}{2}\right) D^{p,0}(x,y) - \left(\frac{p}{p+1}\right)\left(\frac{1-y}{2}\right)^2 D^{p-1,0}(x,y)$
end for
for $p \leftarrow 0, d-1$ **do**
    $D^{p,1}(x,y) := D^{p,0}(x,y)\left(\frac{1+2p+(3+2p)y}{2}\right)$
end for
for $p \leftarrow 0, d-1$ **do**
    for $q \leftarrow 1, d-p-1$ **do**
        $D^{p,q+1}(x,y) := \left(a_q^{2p+1,0} y + b_q^{2p+1,0}\right) D^{p,q}(x,y) - c_q^{2p+1,0} D^{p,q-1}(x,y)$
    end for
end for

polynomials.

Now, any $p \in \mathcal{V}$ is written as a linear combination of the basis functions $\{\phi_i\}$. Introduce a mapping $\mathcal{R}$ from $\mathcal{V}$ into $\mathbb{R}^{|\mathcal{V}|}$ by taking the expansion coefficients of p in terms of $\{\phi_i\}$. That is,

$$p = \mathcal{R}(p)_i \phi_i,$$

where summation is implied over i.

A polynomial p may then be evaluated at a point x as follows. Let Φ be the vector of basis functions tabulated at x. That is,

$$\Phi_i = \phi_i(x). \tag{13.4}$$

Then, evaluating p follows by a simple dot product:

$$p(x) = \mathcal{R}(p)_i \Phi_i. \tag{13.5}$$

More generally in FIAT, a set of polynomials $\{p_i\}$ will need to be evaluated simultaneously, such as evaluating all of the members of a finite element basis. The coefficients of the set of polynomials may be stored in the rows of a matrix C, so that

$$C_{ij} = \mathcal{R}(p_i)_j.$$

Tabulating this entire set of polynomials at a point x is simply obtained by matrix-vector multiplication. Let Φ_i be as in (13.4). Then,

$$p_i(x) = C_{ij}\Phi_j.$$

The basis functions are typically needed at a set of points, such as those of a quadrature rule. Let $\{x_j\}$ now be a collection of points in T and let

$$\Phi_{ij} = \phi_i(x_j),$$

where the rows of Φ run over the basis functions and the columns over the collection of points. As

before, the set of polynomials may be tabulated at all the points by

$$p_i(x_j) = C_{ik}\Phi_{kj},$$

which is just the matrix product $C\Phi$ and may be efficiently carried out by a library operation, such as the `numpy.dot` wrapper to level 3 BLAS.

Finite element computation also requires the evaluation of derivatives of polynomials. In a symbolic context, differentiation presents no particular difficulty, but working in a numerical context requires some special care.

For some differential operator ∂, the derivatives $\partial\phi_i$ are computed at a point x, any polynomial $p = \mathcal{R}(p)_i\phi_i$ may be differentiated at x by

$$\partial p(x) = \mathcal{R}(p)_i(\partial\phi_i),$$

which is exactly analogous to (13.5). By analogy, sets of polynomials may be differentiated at sets of points just like evaluation.

The formulae in Algorithm 4 and their tetrahedral counterpart are fairly easy to differentiate, but derivatives may also be obtained through automatic differentiation. Some experimental support for this using the AD tools in Scientific Python has been developed in an unreleased version of FIAT.

The released version of FIAT currently evaluates derivatives in terms of linear operators, which allows the coordinate singularity in the standard recurrence relations to be avoided. For each Cartesian partial derivative $\frac{\partial}{\partial x_k}$, a matrix ∂^k is calculated such that

$$\mathcal{R}\left(\frac{\partial p}{\partial x_k}\right)_i = \partial^k_{ij}\mathcal{R}(p)_j.$$

Then, derivatives of sets of polynomials may be tabulated by premultiplying the coefficient matrix C with such a ∂^k matrix. These matrices are constructed by tabulating the partial derivatives of the Dubiner bases at a lattice of points and then multiplying by a Vandermonde-type matrix that converts the lattice point values to the expansion coefficients back in the Dubiner basis.

This paradigm may also be extended to vector- and tensor-valued polynomials, making use of the multidimensional arrays implemented in NumPy. Let P be a space of scalar-valued polynomials and $m > 0$ an integer. Then, a member of $[P]^m$, a vector with m components in P, may be represented as a two-dimensional array. Let $p \in [P]^m$ and p^j be the j^{th} component of p. Then $p^j = \mathcal{R}(p)_{jk}\phi_k$, so that $\mathcal{R}(p)_{jk}$ is the coefficient of ϕ_k for p^j.

The previous discussion of tabulating collections of functions at collections of points is naturally extended to this context. If $\{p_i\}$ is a set of members of $[P]^m$, then their coefficients may be stored in an array C_{ijk}, where C_i is the two-dimensional array $\mathcal{R}(p)_{jk}$ of coefficients for p_i. As before, $\Phi_{ij} = \phi_i(x_j)$ contains the values of the basis functions at a set of points. Then, the j^{th} component of p at the point x_k is naturally given by a three-dimensional array

$$p_i(x_k)^j = C_{ijl}\phi_{lk}.$$

If C_{ijl} is stored contiguously in generalized row-major format, this is just a matrix product and no data motion is required to use a library call.

Returning for the moment to scalar-valued polynomials, linear functionals may also be represented as Euclidean vectors. Let $\ell : P \to \mathbb{R}$ be a linear functional. Then, for any $p \in P$,

$$\ell(p) = \ell(\mathcal{R}(p)_i\phi_i) = \mathcal{R}(p)_i\ell(\phi_i),$$

so that ℓ acting on p is determined entirely by its action on the basis $\{\phi_i\}$. As with $\mathcal{R}$, define $\mathcal{R}' : P' \to \mathbb{R}^{|P|}$ by

$$\mathcal{R}'(\ell)_i = \ell(\phi_i),$$

so that

$$\ell(p) = \mathcal{R}'(\ell)_i \mathcal{R}(p)_i.$$

Note that the inverse of $\mathcal{R}'$ is the Banach-space adjoint of $\mathcal{R}$.

Just as with evaluation, sets of linear functionals can be applied to sets of functions via matrix multiplication. Let $\{\ell_i\}_{i=1}^N \subset P'$ and $\{p_i\}_{i=1}^N \subset P$. The functionals are represented by a matrix

$$L_{ij} = \mathcal{R}'(\ell_i)_j$$

and the functions by

$$C_{ij} = \mathcal{R}(p_i)_j$$

Then, evaluating all of the functionals on all of the functions is computed by the matrix product

$$A_{ij} = L_{ik} C_{jk}, \tag{13.6}$$

or $A = LC^\top$. This is especially useful in the setting of the next section, where the basis for the finite element space needs to be expressed as a linear combination of orthogonal polynomials.

Also, the formalism of $\mathcal{R}'$ may be generalized to functionals over vector-valued spaces. As before, let P be a polynomial space of degree n with basis $\{\phi_i\}_{i=1}^{|P|}$ and to each $v \in [P]^m$ associate the representation $v^i = \mathcal{R}(v)_{ij}\phi_j$. In this notation, $v^i = \mathcal{R}(v)_{ij}\phi_j$ is the vector indexed over i. For any functional $\ell \in ([P]^m)'$, a representation $\mathcal{R}'(\ell)_{ij}$ must be defined such that

$$\ell(v) = \mathcal{R}'(\ell)_{ij} \mathcal{R}(v)_{ij},$$

with summation implied over i and j. To determine the representation of $\mathcal{R}'(\ell)$, let e^j be the canonical basis vector with $(e^j)_i = \delta_{ij}$ and write

$$\begin{aligned} \ell(v) &= \ell(\mathcal{R}(v)_{ij} e^i \phi_j) \\ &= \mathcal{R}(v)_{ij} \ell(e^i \phi_j). \end{aligned} \tag{13.7}$$

From this, it is seen that $\mathcal{R}'(\ell)_{ij} = \ell(e^i \phi_j)$.

Now, let $\{v_i\}_{i=1}^N$ be a set of vector-valued polynomials and $\{\ell_i\}_{i=1}^M$ a set of linear functionals acting on them. The polynomials may be stored by a coefficient tensor $C_{ijk} = \mathcal{R}(v_i)_{jk}$. The functionals may be represented by a tensor $L_{ijk} = \mathcal{R}'(\ell_i)_{jk}$. The matrix $A_{ij} = \ell_i(v_j)$ is readily computed by the contraction

$$A_{ij} = L_{ikl} C_{jkl}.$$

Despite having three indices, this calculation may still be performed by matrix multiplication. Since NumPy stores arrays in row-major format, a simple reshaping may be performed so that $A = \tilde{L}\tilde{C}^\top$, for $\tilde{L}$ and $\tilde{C}$ reshaped to two-dimensional arrays by combining the second and third axes.

13.3 Other polynomial spaces

Besides polynomial spaces of some fixed, complete degree, FIAT is motived by more complicated spaces. Once some basis for such spaces is obtained, the preceding techniques apply directly. Most

finite element polynomial spaces may be described either by adding a few basis functions to some polynomials of complete degree or else by constraining such a space by some linear functionals. We describe such techniques in this section.

13.3.1 Supplemented polynomial spaces

A classic example of the first case is the Raviart–Thomas element, where the function space of order q is

$$RT_q = \left(\mathcal{P}_{q-1}(T)\right)^d \oplus \left(\tilde{P}_{q-1}(T)\right) x,$$

where $x \in \mathbb{R}^d$ is the coordinate vector and $\tilde{\mathcal{P}}_q$ is the space of homogeneous polynomials of degree q. Given any basis $\{\phi_i\}$ for $\mathcal{P}_q(T)$ such as the Dubiner basis, it is easy to obtain a basis for $(\mathcal{P}_q(T))^d$ by taking vectors where one component is some ϕ_i and the rest are zero. The issue is obtaining a basis for the entire space.

Consider the case $d = 2$ (triangles). While monomials of the form $x^i y^{q-i}$ span the space of homogeneous polynomials, they are subject to ill-conditioning in numerical computations. On the other hand, the Dubiner basis of order q, $\{\phi_i\}_{i=1}^{|\mathcal{P}_q|}$ may be ordered so that the last $q+1$ functions, $\{\phi_i\}_{i=|\mathcal{P}_q|-q}^{|\mathcal{P}_q|}$, have degree exactly q. While they do not span $\tilde{\mathcal{P}}_q$, the span of $\{x\phi_i\}_{i=|\mathcal{P}_q|-q}^{|\mathcal{P}_q|}$ together with a basis for $(\mathcal{P}_q(T))^2$ does span RT_{q-1}.

So, this gives a basis for the Raviart–Thomas space that can be evaluated and differentiated using the recurrence relations in Algorithm 4. A similar technique may be used to construct elements that consist of standard elements augmented with some kind of bubble function, such as the PEERS element of elasticity or MINI element for Stokes flow.

13.3.2 Constrained polynomial spaces

An example of the second case is the Brezzi–Douglas–Fortin–Marini element (Brezzi and Fortin, 1991). Let $\mathcal{E}(T)$ be the set of facets of T (edges in 2d, faces in 3d). Then the function space is

$$BDFM_q(T) = \{u \in (\mathcal{P}_q(T))^d : u \cdot n|_\gamma \in \mathcal{P}_{q-1}(\gamma),\ \ \gamma \in \mathcal{E}(T)\}$$

This space is naturally interpreted as taking a function space, $(\mathcal{P}_q(T))^d$, and imposing linear constraints. For the case $d = 2$, there are exactly three such constraints. For $\gamma \in \mathcal{E}(T)$, let μ^γ be the Legendre polynomial of degree q mapped to γ. Then, if a function $u \in (\mathcal{P}_q(T))^d$, it is in $BDFM_q(T)$ if and only if

$$\int_\gamma (u \cdot n)\mu^\gamma \, \mathrm{d}s = 0$$

for each $\gamma \in \mathcal{E}(T)$.

Number the edges by $\{\gamma_i\}_{i=1}^3$ and introduce linear functionals $\ell_i(u) = \int_{\gamma_i} (u \cdot n)\mu^{\gamma_i} \, \mathrm{d}s$. Then,

$$BDFM_q(T) = \cap_{i=1}^3 \mathrm{null}(\ell_i).$$

This may naturally be cast into linear algebra and so evaluated with LAPACK. Following the techniques for constructing Vandermonde matrices, a *constraint matrix* may be constructed. Let $\{\bar{\phi}_i\}$ be a basis for $(\mathcal{P}_q(T))^2$. Define the $3 \times |(\mathcal{P}_q)|^2$ matrix

$$C_{ij} = \ell_i(\phi_j).$$

Then, a basis for the null space of this matrix is constructed using the singular value

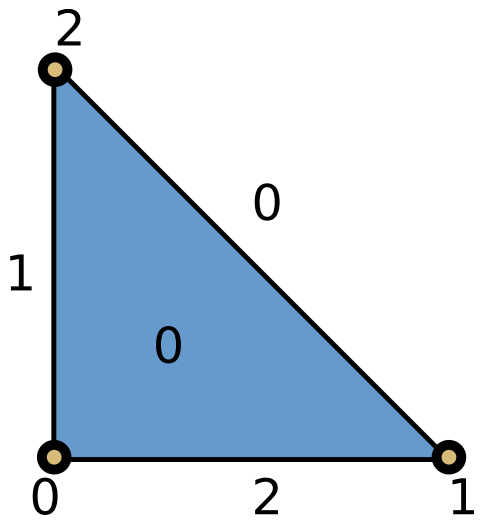


Figure 13.1: The reference triangle, with vertices, edges, and the face numbered.

decomposition (Golub and Van Loan, 1996). The vectors of this null-space basis are readily seen to contain the expansion coefficients of a basis for $BDFM_q$ in terms of a basis for $\mathcal{P}_q(T)^2$. With this basis in hand, the nodal basis for $BDFM_q(T)$ is obtained by constructing the generalized Vandermonde matrix.

This technique may be generalized to three dimensions, and it also applies to Nédélec (Nédélec, 1980), Arnold-Winther (Arnold and Winther, 2002), Mardal-Tai-Winther (Mardal et al., 2002), and many other elements.

13.4 Conveying topological information to clients

Most of this chapter has provided techniques for constructing finite element bases and evaluating and differentiating them. FIAT must also indicate which degrees of freedom are associated with which entities of the reference element. This information is required when local-global mappings are generated by a form compiler such as FFC.

The topological information is provided by a "graded incidence relation" (Kirby, 2006b; Knepley and Karpeev, 2009) and is similar to the presentation of finite element meshes in Logg (2009). Each entity in the reference element is labeled by its topological dimension (e.g. 0 for vertices and 1 for edges), and then the entities of the same dimension are ordered by some convention. To each entity, a list of the local nodes is associated. For example, the reference triangle with entities labeled is shown in Figure 13.1, and the cubic Lagrange triangle with nodes in the dual basis labeled is shown in Figure 13.2.

For this example, the graded incidence relation is stored as

```
{ 0: { 0: [ 0 ] ,
       1: [ 1 ] ,
       2: [ 2 ] } ,
  1: { 0: [ 3 , 4 ] ,
       1: [ 5 , 6 ] ,
       2: [ 7 , 8 ] } ,
  2: { 0: [ 9 ] } }
```

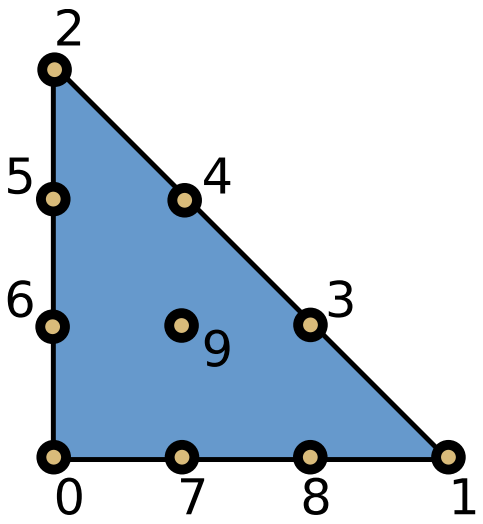


Figure 13.2: The cubic Lagrange triangle, with nodes in the dual basis labelled. Note that the labels in this figure correspond to the FIAT reference element numbering which is different from the numbering imposed by the UFC ordering convention explained in Chapter 16.

13.5 Functional evaluation

In order to construct nodal interpolants or strongly enforce boundary conditions, FIAT also provides information to numerically evaluate linear functionals. These rules are typically exact for a certain degree polynomial and only approximate on general functions. For scalar functions, these rules may be represented by a collection of points and corresponding weights $\{x_i\}, \{w_i\}$ so that

$$\ell(f) \approx w_i f(x_i). \tag{13.8}$$

For example, pointwise evaluation at a point x is simply represented by the coordinates of x together with a weight of one. If the functional is an integral moment, such as

$$\ell(f) = \int_T gf \, \mathrm{d}x, \tag{13.9}$$

then the points $\{x_i\}$ will be those of some quadrature rule and the weights will be $w_i = \omega_i g(x_i)$, where the ω_i are the quadrature weights.

This framework is extended to support vector- and tensor-valued function spaces, by including a component corresponding to each point and weight. If v is a vector-valued function and v_α is its component, then functionals are written in the form

$$\ell(v) \approx w_i v_{\alpha_i}(x_i), \tag{13.10}$$

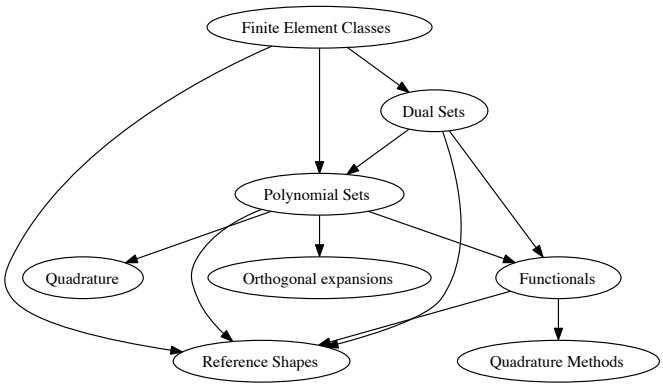
so that the sets of weights, components, and points must be conveyed to the client.

This framework may also support derivative-based degrees of freedom by including a multi-index at each point corresponding to a particular partial derivative.

13.6 Overview of fundamental class structure

Many FEniCS users will never directly use FIAT; for them, interaction will be moderated through a form compiler such as FFC. Others will want to use the FIAT basis functions in other contexts. At a basic level, a user will access FIAT through top-level classes such as `Lagrange` and `RaviartThomas` that implement the elements. Typically, the class constructors accept the reference element and order

Figure 13.3: General relationship between the kinds of classes in FIAT.



of function space as arguments. This gives an interface that is parametrized by dimension and degree. The classes such as `Lagrange` derive from a base class `FiniteElement` that provides access to the three components of the Ciarlet triple.

The function space P is modelled by the base class `PolynomialSet`, which contains a rule for constructing the base polynomials ϕ_i (e.g. the Dubiner basis) and a multidimensional array of expansion coefficients for the basis of P. Special subclasses of this provide (possibly array-valued) orthogonal bases as well as the rules for constructing supplemented and constrained bases. These classes provide mechanisms for tabulating and differentiating the polynomials at input points as well as basic queries such as the dimension of the space.

The set of finite element nodes is similarly modeled by a class `DualBasis`. This provides the functionals of the dual basis as well as their connection to the reference element facets. The functionals are modeled by a `FunctionalSet` object, which is a collection of `Functional` objects. Each `Functional` object contains a reference to the `PolynomialSet` over which it is defined and the array of coefficients representing it and owns a `FunctionalType` class providing the information described in the previous section. The `FunctionalSet` class batches these coefficients together in a single large array.

The constructor for the `FiniteElement` class takes a `PolynomialSet` modeling the starting basis and a `DualBasis` defined over this basis and constructs a new `PolynomialSet` by building and inverting the generalized Vandermonde matrix.

Beyond this basic finite element structure, FIAT provides quadrature such as Gauss-Jacobi rules in one dimension and collapsed-coordinate rules in higher dimensions. It also provides routines for constructing lattices of points on each of the reference element shapes and their facets.

In the future, FIAT will include the developments discussed already (more general reference element geometry/topology and automatic differentiation). Automatic differentiation will make it easier to construct finite elements with derivative-type degrees of freedom such as Hermite, Morley, and Argyris. Additionally, we hope to expand the collection of quadrature rules and provide more advanced point distributions, such as Warburton's warp-blend points (Warburton, 2005).

Finally, we may group the classes used in FIAT into several kinds, and the relationship between these kinds of classes is expressed in Figure 13.3. Top-level classes implement particular finite elements, such as Lagrange or Raviart–Thomas. These depend on classes that implement the underlying reference shapes, polynomial sets, and dual bases. The polynomial sets are linear combinations of orthogonal expansions. Sometimes those linear combinations are constructed via projection (requiring quadrature) or null spaces of linear functionals. Dual bases are collections of linear functionals that can act on a polynomial set over some domain.

14 *Instant: just-in-time compilation of C/C++ in Python*

By Ilmar M. Wilbers, Kent-Andre Mardal and Martin S. Alnæs

Instant is a small Python module for just-in-time (JIT) compilation (or inlining) of C/C++ code. Instant accepts plain C/C++ code and is therefore conveniently combined with the code generating tools in DOLFIN, FFC and SFC.

14.1 Brief overview of Instant and its role in FEniCS

In FEniCS, FFC and SFC are form compilers that generate UFC compliant C++ code based on the language UFL. Within FFC and SFC, Instant is used to JIT-compile the C++ code to a Python module. Similarly, Instant is used in DOLFIN to JIT-compile `Expressions` and `SubDomains`. See the Chapters 11, 15 and 19 for more information on these topics.

Instant relies on SWIG; Beazley (1996) for the generation of wrapper code needed for making the C/C++ code usable from Python. The code to be inlined, in addition to the wrapper code, is then compiled into a Python extension module (a shared library with functionality as specified by the Python C-API) by using Distutils or CMake. To check whether the C/C++ code has changed since the last execution, Instant computes the SHA1 sum (Hansen and Wollman) of the code and compares it to the SHA1 checksum of the code used in the previous execution. Finally, Instant has implemented a set of SWIG typemaps, allowing the user to transfer NumPy arrays between the Python code and the C/C++ code.

14.2 Examples

14.2.1 Hello world

Our first example demonstrates the usage of Instant in a very simple case:

Python code

```
from instant import inline
c_code = r"""
double add(double a, double b)
{
  printf("Hello world! C function add is being called...\n");
  return a+b;
}"""
add_func = inline(c_code)
sum = add_func(3, 4.5)
print "The sum of 3 and 4.5 is", sum
```

When run, this script produces the following output:

Output

```
 > python ex1.py
--- Instant: compiling ---
Hello world! C function add is being called...
The sum of 3 and 4.5 is 7.5
```

Here Instant will wrap the C-function `add` into a Python extension module by using SWIG and Distutils. The inlined function is written in standard C. SWIG supports almost all of C and C++, including classes and templates. The first time the Python script is run, it will use a few second to compile the C code. The next time, however, the compilation is omitted, given that no changes have been made to the C source code.

Although Instant notifies the user when it is compiling, it might sometimes be necessary, e.g. when debugging, to see the details of the Instant internals. We can do this by setting the logging level before calling any other Instant functions:

Python code

```
from instant import output
output.set_logging_level("DEBUG")
```

14.2.2 NumPy arrays

One basic problem with wrapping C and C++ code is how to handle dynamically allocated arrays. Arrays allocated dynamically are typically represented in C/C++ by a pointer to the first element of an array and a separate integer variable holding the array size. In Python the array variable is itself an object containing the data array, array size, type information etc. SWIG provides typemaps to specify mappings between Python and C/C++ types. We will not go into details on typemaps in this chapter, but the reader should be aware that it is a powerful tool that may greatly enhance your code, but also lead to mysterious bugs when used wrongly. Typemaps are discussed in Chapter 19 and at length in the SWIG documentation. In this chapter, it is sufficient to illustrate how to deal with arrays in Instant using the NumPy module.

To illustrate the use of NumPy arrays with Instant, we introduce a solver for an ordinary differential equation (ODE) modeling blood pressure by using a Windkessel model. The ODE is as follows:

$$\frac{d}{dt}p(t) = BQ(t) - Ap(t), \quad t \in (0,1), \tag{14.1}$$

$$p(0) = p_0. \tag{14.2}$$

Here $p(t)$ is the blood pressure, $Q(t)$ is the volume flux of blood, while A and B are real numbers representing resistance and compliance, respectively. An explicit scheme is:

$$p_i = p_{i-1} + \Delta t(BQ_{i-1} - Ap_{i-1}), \quad \text{for} \quad i = 1, \ldots, N-1, \tag{14.3}$$

$$p_0 = p_0. \tag{14.4}$$

The scheme can be implemented in Python as follows using NumPy arrays:

Python code

```
def time_loop_py(p, Q, A, B, dt, N, p0):
    p[0] = p0
    for i in range(1, N):
        p[i] = p[i-1] + dt*(B*Q[i-1] - A*p[i-1])
```

The corresponding C code is:

C++ code

```
void time_loop_c(int n, double* p,
                 int m, double* Q,
                 double A, double B,
                 double dt, int N, double p0)
{
    if ( n != m || N != m )
    {
        printf("n, m and N should be equal\n");
        return;
    }

    p[0] = p0;
    for (int i=1; i<n; i++)
    {
        p[i] = p[i-1] + dt*(B*Q[i-1] - A*p[i-1]);
    }
}
```

In this example, `(int n, double* p)` represents an array of doubles with length n. However, this can not be determined by the function signature:

C++ code

```
void time_loop_c(int n, double* p, int m, double* Q, ...)
```

For example, `double* p` may be an array of length m or it may simply be output. In Instant you must therefore specify what the arrays are:

Python code

```
time_loop_c = inline_with_numpy(c_code,
                                arrays = [["n", "p"],
                                          ["m", "Q"]])
```

Here, we tell Instant that `(int n, double* p)` and `(int m, double* Q)` are NumPy arrays (and Instant then generates the proper typemaps). Notice that the order of the elements in the array specification is: 1) the length of the array and 2) the array pointer. The order of the arguments in the *C* code may differ from the order in the `array` specification. We may then call the `time_loop` function as follows:

Python code

```
time_loop_c(p, Q, 1.0, 1.0, 1.0/(N-1), N, 1.0)
```

In Table 14.1 we compare the above mentioned code with pure C code, pure Python, and NumPy. We obtain a speed-up of about a factor 350 when compared with NumPy, using 10^5 time steps. The performance of the code using Instant is actually the same as a pure C program. The comparison between NumPy and Instant may not be completely fair. NumPy is primarily intended for algorithms that can be vectorized, which is not the case with ODEs. In fact, utilizing pure Python lists instead of NumPy arrays, reduces the speed-up to a factor 65. For code that can be vectorized, the speed-up is about one order of magnitude, when using Instant instead of NumPy (Wilbers et al., 2009). The result of solving the ODE can be seen in Figure 14.1.

The complete code for this example can be found in `ex2.py`.

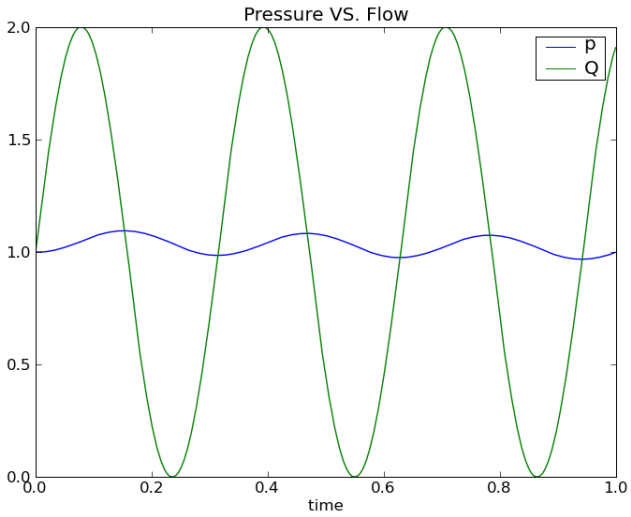


Figure 14.1: Plot of pressure and blood volume flux computed by solving the Windkessel model.

N	10^2	10^3	10^4	10^5	10^6
CPU time with NumPy	3.9e-4	3.9e-3	3.8e-2	3.8e-1	3.8
CPU time with Python	0.7e-4	0.7e-3	0.7e-2	0.7e-1	0.7
CPU time with Instant	5.0e-6	1.4e-5	1.0e-4	1.0e-3	1.1e-2
CPU time with C	4.0e-6	1.1e-5	1.0e-4	1.0e-3	1.1e-2

Table 14.1: CPU times (in seconds) for solving the ODEs from the Windkessel model using different implementations.

14.2.3 NumPy arrays and OpenMP

It is easy to speed up code on parallel computers with OpenMP. In the following code preprocessor directives like `#pragma omp ...` are OpenMP directives and OpenMP functions always start with `omp`. In this example, we want to solve a standard 2-dimensional wave equation in a heterogeneous medium with local wave velocity k:

$$\frac{\partial^2 u}{\partial t^2} = \nabla \cdot [k \nabla u] \,. \tag{14.5}$$

We set the boundary condition to $u = 0$ for the whole boundary of a rectangular domain $\Omega = (0,1) \times (0,1)$. Further, u has the initial value $I(x,y)$ at $t = 0$ while $\partial u / \partial t = 0$. We solve the wave equation using the following finite difference scheme:

$$\begin{aligned} u^l_{i,j} = &\left(\frac{\Delta t}{\Delta x}\right)^2 [k_{i+\frac{1}{2},j}(u_{i+1,j} - u_{i,j}) - k_{i-\frac{1}{2},j}(u_{i,j} - u_{i-1,j})]^{l-1} \\ &+ \left(\frac{\Delta t}{\Delta y}\right)^2 [k_{i,j+\frac{1}{2}}(u_{i,j+1} - u_{i,j}) - k_{i,j-\frac{1}{2}}(u_{i,j} - u_{i,j-1})]^{l-1}. \end{aligned} \tag{14.6}$$

Here, $u^l_{i,j}$ represents u at the grid point x_i and y_j at time level t_l, where

$$x_i = i\Delta x, i = 0, \ldots, n \tag{14.7}$$
$$y_i = j\Delta y, j = 0, \ldots, m \text{ and} \tag{14.8}$$
$$t_l = l\Delta t, \tag{14.9}$$

Also, $k_{i+\frac{1}{2},j}$ is short for $k(x_{i+\frac{1}{2}}, y_j)$.

The code for calculating the next time step using OpenMP looks like:

C++ code

```
void stencil(double dt, double dx, double dy,
             int ux, int uy, double* u,
             int umx, int umy, double* um,
             int kx, int ky, double* k,
             int upn, double* up){
#define index(u, i, j) u[(i)*m + (j)]
  int i=0, j=0, m = ux, n = uy;
  double hx, hy, k_c, k_ip, k_im, k_jp, k_jm;
  hx = pow(dt/dx, 2);
  hy = pow(dt/dy, 2);
  j = 0;   for (i=0; i<m; i++) index(up, i, j) = 0;
  j = n-1; for (i=0; i<m; i++) index(up, i, j) = 0;
  i = 0;   for (j=0; j<n; j++) index(up, i, j) = 0;
  i = m-1; for (j=0; j<n; j++) index(up, i, j) = 0;
  #pragma omp for
  for (i=1; i<m-1; i++){
    for (j=1; j<n-1; j++){
      k_c = index(k, i, j);
      k_ip = 0.5*(k_c + index(k, i+1, j));
      k_im = 0.5*(k_c + index(k, i-1, j));
      k_jp = 0.5*(k_c + index(k, i, j+1));
      k_jm = 0.5*(k_c + index(k, i, j-1));
      index(up, i, j) = 2*index(u, i, j) - index(um, i, j) +
        hx*(k_ip*(index(u, i+1, j) - index(u, i, j)) -
            k_im*(index(u, i, j) - index(u, i-1, j))) +
        hy*(k_jp*(index(u, i, j+1) - index(u, i, j)) -
            k_jm*(index(u, i, j) - index(u, i, j-1)));
```

N	1e+8	2e+8
CPU time with Instant 1 CPU	0.80	1.59
CPU time with Instant 2 CPU	0.42	0.81
CPU time with Instant 3 CPU	0.37	0.75
CPU time with Instant 4 CPU	0.34	0.67

Table 14.2: CPU times (in seconds) for the implementation of the solution of a wave equation using Instant and OpenMP on different numbers of CPUs/threads.

```
    }
  }
}
```

We also need to add the OpenMP header `omp.h` and compile with the flag `-fopenmp` and link with the OpenMP shared library, e.g. `libgomp.so` for Linux (specified with `-lgomp`). This can be done as follows:

Python code

```
instant_ext = \
  build_module(code=c_code,
               system_headers=["numpy/arrayobject.h",
                               "omp.h"],
               include_dirs=[numpy.get_include()],
               init_code="import_array();",
               cppargs=["-fopenmp"],
               lddargs=["-lgomp"],
               arrays=[["ux", "uy", "u"],
               ["umx", "umy", "um"],
               ["kx", "ky", "k"],
               ["upn", "up", "out"]])
```

Note that the arguments `include_headers`, `init_code`, and the first element of `system_headers` could have been omitted if we used `inline_module_with_numpy`(see below) instead of `build_module`. The complete code can be found in `ex3.py`.

In Table 14.2 we have compared the timings of running with different numbers of CPUs. The timings in this table are performed on a quad-core machine with 32GB memory. We see a speed-up of factor two when doubling the number of CPUs, but further increasing the number of CPUs has a limited effect. We have not been able to investigate this further, but suspect that the physical layout of the machine with two dual cores causes this, as the two CPUs on the same core share some of the resources.

14.3 Errors encountered when using Instant

There are basically three different types of errors you can encounter when using Instant. These are: 1) errors caused by non-compilable C/C++ code, 2) errors caused by wrong usage of SWIG, and 3) errors related to importing the module from the cache. We will now go briefly through these three different types of errors.

Let us start by removing a `";"` in the C++ code of `ex2.py`, making the C++ compiler unable to compile the code. We will then get errors on the following form:

Output

```
--- Instant: compiling ---
In instant.recompile: The module did not compile,
   see "/tmp/tmpZ4M_ZO2010-11-9-08-24_instant/instant_module_dff94651124193a[...]
Traceback (most recent call last):
      File "test2.py", line 21, in <module>
          sum_func = inline_with_numpy(c_code, arrays = [["n1", "array1"]])
        File "/usr/local/lib/python2.6/dist-packages/instant/inlining.py", line 95, in
            inline_with_numpy
          module = build_module(**kwargs)
        File "/usr/local/lib/python2.6/dist-packages/instant/build.py", line 474, in build_module
          recompile(modulename, module_path, setup_name, new_compilation_checksum)
        File "/usr/local/lib/python2.6/dist-packages/instant/build.py", line 100, in recompile
          "compile, see '%s'" % compile_log_filename)
        File "/usr/local/lib/python2.6/dist-packages/instant/output.py", line 49, in instant_error
          raise RuntimeError(text)
      RuntimeError: In instant.recompile: The module did not compile,
      see "/tmp/tmpZ4M_ZO2010-11-9-08-24_instant/instant_module_dff946511241[...]
```

The error message from the compiler is located in the file `compile.log` in the temporary directory `/tmp/tmpZ4M_ZO2010-11-9-08-24_instant/instant_module_dff946511241aab327593a2d71105c5fc/`. The compile error message will here refer to line numbers in the wrapper code generated by SWIG. You should still be able to locate the C++ error, by looking at the error message in `compile.log` and the file containing the wrapper code (named `*_wrap.cxx`) in the temporary directory.

The second type of error occurs when SWIG is not able to parse the code. These errors are easily identified by the first line in the error message, namely `Error: Syntax error in input(1).`

Output

```
instant_module_815d9b7181988c1596a71b62f8a17936a77e5944.i:39: Error: Syntax error in input(1).
running build_ext
building "_instant_module_815d9b7181988c1596a71b62f8a17936a77e5944" extension
creating build
creating build/temp.linux-i686-2.6
gcc -pthread -fno-strict-aliasing -DNDEBUG -g -fwrapv -O2 -Wall -Wstrict-prototypes -fPIC
   -I/usr/lib/python2.6/dist-packages/numpy/core/include -I/usr/include/python2.6
   -c instant_module_815d9b7181988c1596a71b62f8a17936a77e5944_wrap.cxx
   -o build/temp.linux-i686-2.6/instant_module_815d9b7181988c1596a71b62f8a1[...].o -O2
gcc: instant_module_815d9b7181988c1596a71b62f8a17936a77e5944_wrap.cxx: No such file or directory
gcc: no input files
```

SWIG reports that it is unable to parse the Instant generated interface file (named `*.i`) and that the problem arises at line 39. In this case, you should have a look in the generated interface file in the temporary directory.

Finally, Python may not be able to import the module from the cache. There might be numerous reasons for this; the cache may be old and incompatible with the current version of Python, the cache may be corrupted due to disk failure, some shared libraries might be missing from `$LD_LIBRARY_PATH` and so on. Such error messages look like:

Output

```
In instant.import_module_directly:
Failed to import module "instant_module_4b41549bc6282877d3f97d54ef664d4" from
     "/home/fenics/.instant/cache".
Traceback (most recent call last):
  File "test2.py", line 21, in <module>
    sum_func = inline_with_numpy(c_code, arrays = [["n1", "array1"]])
  File "/usr/local/lib/python2.6/dist-packages/instant/inlining.py", line 95, in inline_with_numpy
    module = build_module(**kwargs)
```

```
  File "/usr/local/lib/python2.6/dist-packages/instant/build.py", line 383, in build_module
    module = check_disk_cache(modulename, cache_dir, moduleids)
  File "/usr/local/lib/python2.6/dist-packages/instant/cache.py", line 121, in check_disk_cache
    module = import_and_cache_module(path, modulename, moduleids)
  File "/usr/local/lib/python2.6/dist-packages/instant/cache.py", line 67, in
       import_and_cache_module
    instant_assert(module is not None, "Failed to import module found in cache."
  File "/usr/local/lib/python2.6/dist-packages/instant/output.py", line 55, in instant_assert
    raise AssertionError(text)
```

In this case it is advantageous to make a local cache in the current working directory, using `cache_dir="test_cache"`, and go to the local cache to find the error.

14.4 Instant explained

The previous section concentrated on the usage of Instant. In this section we explain what Instant does. We will again use our first example, but we set the module name explicitly with the keyword argument `modulename` to see more clearly what happens:

Python code

```
from instant import inline
code = r"""
double add(double a, double b)
{
  printf("Hello world! C function add is being called...\n");
  return a+b;
}"""
add_func = inline(code, modulename="ex4")
sum = add_func(3, 4.5)
print "The sum of 3 and 4.5 is", sum
```

After running this code there is a new directory `ex4` in our directory. The contents are:

Output

```
~/instant_doc/code$ cd ex4/
~/instant_doc/code/ex4$ ls -g
total 224
drwxr-xr-x 4 fenics  4096 2009-05-18 16:52 build
-rw-r--r-- 1 fenics   844 2009-05-18 16:52 compile.log
-rw-r--r-- 1 fenics   183 2009-05-18 16:52 ex4-0.0.0.egg-info
-rw-r--r-- 1 fenics    40 2009-05-18 16:52 ex4.checksum
-rw-r--r-- 1 fenics   402 2009-05-18 16:53 ex4.i
-rw-r--r-- 1 fenics  1866 2009-05-18 16:52 ex4.py
-rw-r--r-- 1 fenics  2669 2009-05-18 16:52 ex4.pyc
-rwxr-xr-x 1 fenics 82066 2009-05-18 16:52 _ex4.so
-rw-r--r-- 1 fenics 94700 2009-05-18 16:52 ex4_wrap.cxx
-rw-r--r-- 1 fenics    23 2009-05-18 16:53 __init__.py
-rw-r--r-- 1 fenics   448 2009-05-18 16:53 setup.py
```

The file `ex4.i` is the SWIG interface file. Another central file is the Distutils file `setup.py`, which is generated and executed. During execution, `setup.py` first runs SWIG on the interface file, producing `ex4_wrap.cxx` and `ex4.py`. The first file is then compiled into a shared library `_ex4.so` (note the leading underscore). The file `ex4-0.0.0.egg-info` and the directory `build` are also created by Distutils. The output from executing the Distutils file is stored in the file `compile.log`. Finally, a checksum file named `ex4.checksum` is generated, containing a checksum based on the files present in the directory. The final step consists of moving the whole directory from its temporary location to either cache or a user-specified directory. The file `__init__.py` imports the module `ex4` into Python.

The script `instant-clean` removes compiled modules from the Instant cache, located in the directory `.instant` in the home directory of the user running it. The script `instant-showcache` shows the modules located in the Instant cache.

14.4.1 Arrays and typemaps

Instant has support for converting NumPy arrays to C arrays and vice versa. Each array specification is a list containing the names of the variables describing that array in the C code. For a 1D array, this means the names of the variables containing the length of the array (an `int`), and the array pointer. The array pointer can have several types, but the default is `double`. For 2D arrays we need three strings, two for the length in each dimension, and one for the array pointer. This following example illustrate the array specification:

Python code

```
arrays = [["len_a", "a"],                    # a 1D array / vector
          ["len_bx", "len_by", "b"],         # a matrix
          ["len_cx", "len_cy", "len_cz", "c"]] # a 3D tensor
```

The variables names specified reflect the variable names in the C function signature. It is important that the order of the variables in the signature is retained for each array; that is, the signature should be:

C++ code

```
double sum (int len_a, double*a,
            int len_bx, int len_by, double* b,
            int len_cx, int len_cy, int_cz, double* c)
```

The arrays are assumed to be of type `double` by default, but several other types are supported. These types are `float`, `short`, `int`, `long`, `long long`, `unsigned short`, `unsigned int`, `unsigned long`, and `unsigned long long`. The type can be specified by adding an additional value to the list describing the array, e.g.

Python code

```
arrays = [["len_a", "a", "long"]]
```

It is important that there is correspondence between the type of the NumPy array and the type in the signature of the C function. For arrays that are changed in-place (the arrays are both input and output) the types have to match exactly. For arrays that are input or output (see next paragraph), one has to make sure that the implicit casting is done to a type with higher precision. For input arrays, the C type must be of the same or higher precision as the NumPy array, while for output arrays the NumPy array type must be of the same or higher precision as the C array. The NumPy type `float32` corresponds to the C type `float`, while `float64` corresponds to `double`. The NumPy type `float` is the same as `float64`. For integer arrays, the mapping between NumPy types and C types depends on your system. Using `long` as the C type will work in most cases.

Instant supports both input, output and in-place (input-output) arrays. The default behavior is to treat the arrays as in-place arrays, provided that the input are NumPy arrays. Python lists and sequences are converted to NumPy arrays automatically. The following code shows an example where we calculate the matrix-vector multiplication $x = Ab$. The integer matrix A and double vector b are marked as input, while the double vector x is output. The code can be found in: `ex5.py`.

Python code

```
c_code = """
void dot_c(int Am, int An, long* A, int bn, long* b, int xn, double* x)
{
    for (int i=0; i<Am; i++)
    {
        x[i] = 0;
        for (int j=0; j<An; j++)
        {
            x[i] += A[i*Am + j]*b[j];
        }
    }
}
"""
dot = inline_with_numpy(c_code,
                        arrays = [["Am", "An", "A", "in", "long"],
                                  ["bn", "b", "in", "long"],
                                  ["xn", "x", "out", "double"]])
a = arange(9)
a.shape = (3, 3)
b = arange(3)

x1 = dot(a, b, a.shape[1])
```

Notice that we obtain the desired behavior, namely that `b` is input and `x` is output that should have dimension `a.shape[1]`.

Finally, it is possible to work with arrays that are more than 3-dimensional. However, the typemaps used for this employ less error checking, and can currently only be used for the C type `double`. The list describing the array should contain the variable name for holding the number of dimensions, the variable name for an integer array holding the size in each dimension, the variable name for the array, and the argument `"multi"`, indicating that it has more than 3 dimensions. The `arrays` argument could for example be:

Python code

```
arrays = [["m", "mp", "ar1", "multi"],
          ["n", "np", "ar2", "multi"]]
```

In this case, the C function signature should look like:

C++ code

```
void sum (int m, int* mp, double* ar1, int n,
          int* np, double* ar2)
```

14.4.2 Module name, signature, and cache

The Instant cache resides in the directory `.instant` in the home directory of the user. It is possible to specify a different directory, but the `instant-clean` script will not remove these when executed. The three keyword arguments `modulename`, `signature`, and `cache_dir` are related. If none of them are given, the default behavior is to create a signature from the contents of the files and arguments to the `build_module` function. In this case the resulting name starts with `instant_module_` and is followed by a long checksum. The resulting code is copied to the Instant cache unless `cache_dir` is set to a specific directory. Note that changing the arguments, code or compile arguments will result in a new directory in the Instant cache. Before compiling a module, Instant will always check if the module is cached in either the Instant cache or in the current working directory.

If `modulename` is used, the directory with the resulting code is named accordingly, but not copied to the Instant cache. Instead, it is stored in the current working directory. Any changes to the argument or the source files will automatically result in a recompilation. The argument `cache_dir` is ignored.

When `signature` is given as argument, Instant uses the signature instead of computing the checksum. The resulting directory has the same name if the signature contains less than or equal to 100 characters (letters, numbers, or underscores). If this is not the case, the module name is generated based on the checksum of this string, resulting in a module name starting with `instant_module_` followed by the checksum. Because the user specifies the signature herself, changes in the arguments or source code will not cause a recompilation.

In addition to the disk cache discussed so far, Instant also has a memory cache. All modules used during the life-time of a program are stored in memory for faster access. The memory cache is always checked before the disk cache.

14.4.3 Locking

Instant provides file locking functionality for cache modules. If multiple processes are working on the same module, race conditions could potentially occur, where two or more processes believe the module is missing from the cache and try to write it simultaneously. To avoid race conditions, lock files have been introduced. The lock files reside in the Instant cache, and locking is only enabled for modules that should be cached; that is, where the module name is not given explicitly as argument to `build_module` or one of its wrapper functions. The first process to reach the stage where the module is copied from its temporary location to the Instant cache will acquire a lock, and other processes cannot access this module while it is being copied.

14.5 Instant API

In this section we will describe the various Instant functions and their arguments. The first six functions are the core Instant functions. The function `build_module` is the main function, while the five next functions are wrappers around this function. Finally, there are also four helper functions available, intended for using Instant with other applications.

14.5.1 `build_module`

This function is the most important one in Instant, and for most applications the only one that developers need to use (together with the wrapper functions). The return argument is the compiled module, which can be used directly in the calling code.

There are a number of keyword arguments, and we will explain them in detail here. Although one of the aims of Instant is to minimize the direct interaction with SWIG, some of the keywords require knowledge of SWIG in order to make sense. In this way, Instant can be used both by programmers new to the use of extension languages for Python, as well as by experienced SWIG programmers. The keywords arguments are as follows:

- `modulename`
  - Default: `None`
  - Type: String
  - Comment: The name you want for the module. If specified, the module will not be cached. If missing, a name will be constructed based on a checksum of the other arguments, and the module will be placed in the global cache.

- `source_directory`
  - Default: "."
  - Type: String
  - Comment: The directory where user supplied files reside. The files given in `sources`, `wrap_headers`, and `local_headers` are expected to exist in this directory.
- `code`
  - Default: `""`
  - Type: String
  - Comment: The C or C++ code to be compiled and wrapped.
- `init_code`
  - Default: `""`
  - Type: String
  - Comment: Code that should be executed when the Instant module is initialized. An example of initialization code is the call `import_array()` required for initialization of NumPy.
- `additional_definitions`
  - Default: `""`
  - Type: String
  - Comment: Additional definitions needed in the interface file. These definitions should be additional code that is not found elsewhere, but is needed by the wrapper code. These definitions should be given as triple-quoted strings in the case they span multiple lines, and are placed both in the initial block for C/C++ code (`%{,%}`-block), and the main section of the interface file.
- `additional_declarations`
  - Default: `""`
  - Type: String
  - Comment: Additional declarations needed in the interface file. These declarations should be declarations of code that is found elsewhere, but is needed to make SWIG generate wrapper code properly. These declarations should be given as triple-quoted strings in the case they span multiple lines, and are placed in the main section of the interface file.
- `sources`
  - Default: `[]`
  - Type: List of strings
  - Comment: Source files to compile and link with the module. These files are compiled together with the SWIG-generated wrapper file into the shared library file. Should reside in the directory specified in `source_directory`.
- `wrap_headers`

  - Default: `[]`
  - Type: List of strings
  - Comment: Local header files that should be wrapped by SWIG. The files specified will be included both in the initial block for C/C++ code (with a C directive) and in the main section of the interface file (with a SWIG directive). Should reside in the directory specified in `source_directory`.

- `local_headers`
  - Default: `[]`
  - Type: List of strings
  - Comment: Local header files required to compile the wrapped code. The files specified will be included in the initial block for C/C++ code (with a C directive). Should reside in the directory specified in `source_directory`.

- `system_headers`
  - Default: `[]`
  - Type: List of strings
  - Comment: System header files required to compile the wrapped code. The files specified will be included in the initial block for C/C++ code (with a C directive).

- `include_dirs`
  - Default: `[]`
  - Type: List of strings
  - Comment: Directories to search for header files for building the extension module. Need to be absolute path names.

- `library_dirs`
  - Default: `[]`
  - Type: List of strings
  - Comment: Directories to search for libraries (`-l`) for building the extension module. Need to be absolute paths.

- `libraries`
  - Default: `[]`
  - Type: List of strings
  - Comment: Libraries needed by the Instant module. The libraries will be linked in from the shared object file. The initial `-l` is added automatically.

- `swigargs`
  - Default: `["-c++", "-fcompact", "-O", "-I.", "-small"]`
  - Type: List of strings
  - Comment: Arguments to swig, e.g. `["-lpointers.i"]` to include the SWIG library `pointers.i`.

- `swig_include_dirs`
  - Default: `[]`
  - Type: List of strings
  - Comment: Directories to include in the `swig` command.
- `cppargs`
  - Default: `["-O2"]`
  - Type: List of strings
  - Comment: Arguments to the C++ compiler (except include directories) e.g. `["-Wall", "-fopenmp"]`.
- `lddargs`
  - Default: `[]`
  - Type: List of strings
  - Comment: Arguments to the linker, other than libraries and library directories, e.g. `["-E", "-U"]`.
- `arrays`
  - Default: `[]`
  - Type: List of strings
  - Comment: A nested list describing the C arrays to be made from the NumPy arrays. For 1D arrays, the list should contain strings with the variable names for the length of the arrays and the array itself. Matrices should contain the names of the dimensions in the two directions as well as the name of the array, and 3D tensors should contain the names of the dimensions in the three directions in addition to the name of the array. If the NumPy array has more than three dimensions, the list should contain strings with variable names for the number of dimensions, the length in each dimension as a pointer, and the array itself, respectively.
- `generate_interface`
  - Default: `True`
  - Type: Boolean
  - Comment: Indicate whether you want to generate the interface files.
- `generate_setup`
  - Default: `True`
  - Type: Boolean
  - Comment: Indicate if you want to generate the `setup.py` file.
- `signature`
  - Default: `None`
  - Type: String

  - Comment: A signature string to identify the form instead of the source code. See Section 14.4.2.

- `cache_dir`
  - Default: None
  - Type: String
  - Comment: A directory to look for cached modules and place new ones. If missing, a default directory is used. Note that the module will not be cached if `modulename` is specified.

14.5.2 `inline`

The function `inline` returns a compiled function if the input is a valid C/C++ function and a module if not.

14.5.3 `inline_module`

The same as `inline`, but returns the whole module rather than a single function.

14.5.4 `inline_with_numpy`

The difference between this function and the `inline` function is that C arrays can be used. This means that the necessary arguments (`init_code` (`import_array`), `system_headers`, and `include_dirs`) for converting NumPy arrays to C arrays are set by the function.

14.5.5 `inline_module_with_numpy`

The difference between this function and the `inline_module` function is that C arrays can be used. This means that the necessary arguments (`init_code`, `system_headers`, and `include_dirs`) for converting NumPy arrays to C arrays are set by the function.

14.5.6 `import_module`

This function can be used to import cached modules from the current work directory or the Instant cache. It has one mandatory argument, `moduleid`, and one keyword argument `cache_dir`. If the latter is given, Instant searches the specified directory instead of the Instant cache, if this directory exists. If the module is not found, `None` is returned. The `moduleid` arguments can be either the module name, a signature, or an object with a function `signature`.

Using the module name or signature, assuming the module `instant_ext` exists in the current working directory or the Instant cache, we import a module in the following way:

Python code

```
instant_ext = import_module("instant_ext")
```

An object and a directory can be used as input provided that this object includes a function `signature()` and that the module is located in the directory:

Python code

```
instant_ext = import_module(object, dir)
```

If the module is found, the imported module is placed in the memory cache.

14.5.7 header_and_libs_from_pkgconfig

This function returns a list of include files, flags, libraries and library directories obtained from `pkg-config`. It takes any number of arguments, one string for every package name. It returns four or five arguments. Unless the keyword argument `returnLinkFlags` is given with the value `True`, it returns lists with the include directories, the compile flags, the libraries, and the library directories of the package names given as arguments. If `returnLinkFlags` is `True`, the link flags are returned as a fifth list. It is used as follows:

Python code

```
inc_dirs, comp_flags, libs, lib_dirs, link_flags = \
header_and_libs_from_pkgconfig("ufc-1", "libxml-2.0",
                               "numpy-1",
                               returnLinkFlags=True)
```

14.5.8 get_status_output

This function provides a platform-independent way of running processes in the terminal and extracting the output using the Python module `subprocess`. The one mandatory argument is the command we want to run. Further, there are three keyword arguments. The first is `input`, which should be a string containing input to the process once it is running. The other two are `cwd` and `env`. We refer to the documentation of `subprocess`[1] for a more detailed description of these, but in short the first is the directory in which the process should be executed, while the second is used for setting the necessary environment variables.

14.5.9 get_swig_version

The function returns the SWIG version number like `"1.3.36"`.

14.5.10 check_swig_version

Takes a single argument, which should be a string on the same format as the output of `get_swig_version`. Returns `True` if the version of the installed SWIG is equal to or greater than the version passed to the function. It also has a keyword argument `same` for testing whether the two versions are the same.

14.6 Related work

There exist several packages that are similar to Instant. We mention Weave, Cython, and F2PY. Weave, which is part of SciPy, allows inlining of C code directly in Python code. Unlike Instant, Weave does not require the specification of a function signature. For specific examples of Weave and the other mentioned packages, we refer to (Wilbers et al., 2009). F2PY, which is part of NumPy, is primarily intended for wrapping Fortran code although it can be used for wrapping C code. Cython is a rather new project, branched from the Pyrex. Cython is attractive because of its integration with NumPy arrays. Cython differs from the other projects by being a programming language of its own, which extends Python with static typing. Cython can be used to wrap C code and to transform Python code to C, and is currently gaining a lot of momentum.

[1] This documentation can be accessed by running the command `pydoc subprocess`.

15 SyFi and SFC: symbolic finite elements and form compilation

By Martin Sandve Alnæs and Kent-Andre Mardal

This chapter concerns the finite element library SyFi and its form compiler SFC. SyFi is a framework for defining finite elements symbolically, using the C++ library GiNaC (Bauer et al., 2002) and its Python interface Swiginac (Skavhaug and Čertík, 2009). In many respects, SyFi is the equivalent of FIAT (see Chapter 13), whereas SFC corresponds to FFC (see Chapter 11). SyFi and SFC come with an extensive manual (Alnæs and Mardal, 2009) which can be found on the FEniCS web page. SFC can be used in FEniCS as a form compiler. Similarly to FFC, it translates UFL code (see Chapter 17) into UFC code (see Chapter 16), which can be used by the DOLFIN assembler described in Chapter 6. The UFC code is JIT-compiled using Instant, see Chapter 14.

This chapter is deliberately short and only gives the reader a taste of the capabilities of SyFi and SFC. However, most features are covered by the more comprehensive manual. This chapter is organized as follows. We begin with a short description of GiNaC and Swiginac before we present how finite elements are used and defined using SyFi. Then, we present how to use SFC in the DOLFIN environment, and end with a short description of the basic structure of SFC. SyFi is implemented in C++, but has a Python interface. SFC is implemented in Python because code generation is much more convenient in this language.

15.1 GiNaC and Swiginac

GiNaC (Bauer et al., 2002) is an open source C++ library for symbolic calculations. It contains the tools for doing basic manipulations of polynomials like algebraic operations, differentiation, and integration. The following example shows basic usage of the library:

C++ code

```
// create a polynomial function
symbol x("x");
ex f = x*(1-x);

// evaluate f
ex fvalue = f.subs(x == 0.5);
std::cout << " f(0.5) = " << fvalue << std::endl;

// differentiation
ex dfdx  = diff(f,x);
std::cout << " df/dx = " << dfdx << std::endl;

// integration
ex intf  = integral(x,0,1,f).eval_integ();
std::cout << " integral of f from 0 to 1 is: " << intf << std::endl;
```

We will not go deeply into GiNaC here, but refer the reader to the GiNaC tutorial and reference which can be found on its web page. There are, however, a few issues we need to address to explain basic GiNaC usage. First of all, GiNaC contains many different types like `symbol`, `matrix`, `function`, etc. Normally, one does not need to worry about these types since the type `ex`, which was used above, can represent any mathematical object (`ex` is basically a place-holder for the underlying object). Still, there are mathematical operations that can only be applied to particular types. For instance, functions can only be differentiated with respect to symbols, as shown above. Notice also that GiNaC overloads operators like == to enable creation of equations and inequalities, which may be represented as expressions of type `relations` (or `ex`).

Symbolic calculations can be computationally demanding. Therefore, GiNaC separates between the construction and evaluation of expressions. This is illustrated in the above example by the fact that we create an integral object using the function `integral`, but we need to explicitly call the function `eval_integ` to compute the integral. In a similar fashion one may use functions like `eval`, `evalm`, `expand`, `simplify`, and `collect_common_factors` etc. to evaluate and simplify expressions. Finally, GiNaC implements its own memory management system using reference counting. The complete code can be found at `syfi-sfc/ginac`.

Swiginac is a Python interface to GiNaC created using SWIG. Swiginac provides a more or less direct translation of GiNaC to Python, but has features that makes it easy to program in a Pythonic way. For instance, Swiginac unwraps the `ex` objects and provides typemaps between Python lists and GiNaC lists (`lst`). The following code translates the above C++ example to Python, using Swiginac:

Python code

```
from swiginac import *
x = symbol("x")
f = x*(1-x)

fvalue = f.subs(x == 0.5)
print "fvalue = ", fvalue

dfdx = diff(f,x)
print "df/dx = ", dfdx

intf = integral(x,0,1,f).eval_integ()
print "integral of f from 0 to 1 is:", intf
```

15.2 SyFi: symbolic finite elements

GiNaC provides the basic utilities for SyFi in the sense that it provides manipulation of polynomials, as well as differentiation and integration with respect to one variable. SyFi extends GiNaC with polynomial spaces and differentiation operators like grad, div, and curl, in addition to integration over a number of polygonal domains. With these utilities it is easy to define finite elements.

Some elements that have been implemented include: continuous and discontinuous Lagrange elements, the Crouzeix–Raviart element, the Raviart–Thomas element, various $H(\mathrm{div})$ and $H(\mathrm{curl})$ Nédélec elements, and the Hermite elements. See Chapter 3 for a description of the above-mentioned elements. A complete list of elements can be found in the user manual. The mentioned elements are defined for arbitrary order, except for the Crouzeix–Raviart and Hermite elements. Not all of these elements are, however, supported by the form compiler.

The following example illustrates how to use SyFi to do finite element calculations in Python. Here, we create a Lagrange element of second order and use the basis functions to compute a element stiffness matrix on a reference triangle. We also print both the integrand and the element matrix entries to the screen.

Python code

```
from swiginac import *
from SyFi import *

#initialize SyFi in 2D
initSyFi(2)

# create reference triangle
t = ReferenceTriangle()

# create second order Langrange element
fe = Lagrange(t,2)

for i in range(0, fe.nbf()):
    for j in range(0, fe.nbf()):
        integrand = inner(grad(fe.N(i)), grad(fe.N(j)))
        print "integrand[%d, %d]  =%s;" % (i, j, integrand.printc())
        integral = t.integrate(integrand)
        print "A[%d, %d]          =%s;" % (i, j, integral.printc())
```

The output from executing the above code is:

C++ code

```
integrand[0, 0]  =2.0*pow( 4.0*y+4.0*x-3.0,2.0);
A[0, 0]          =1.0;
integrand[0, 1]  = -4.0*( 4.0*y+4.0*x-3.0)*x+-4.0*( y+2.0*x-1.0)*( 4.0*y+4.0*x-3.0);
A[0, 1]          =-(2.0/3.0);
integrand[0, 2]  =( 4.0*y+4.0*x-3.0)*( 4.0*x-1.0);
A[0, 2]          =(1.0/6.0);
integrand[0, 3]  = -4.0*y*( 4.0*y+4.0*x-3.0)+-4.0*( 4.0*y+4.0*x-3.0)*( 2.0*y+x-1.0);
A[0, 3]          =-(2.0/3.0);
....
```

Here, we see that the expressions are printed to the screen as symbolic expressions in C++ syntax. Hence, the output is very reader–friendly and this can be very useful during debugging. We remark that also Python and LaTeX output can be generated using the `printpython` and `printlatex` functions.

All elements in SyFi are implemented in C++. Here, however, for simplicity we list a definition of the Crouzeix–Raviart element in Python. The following code is the complete finite element definition:

Python code

```
from swiginac import *
from SyFi import *

class CrouzeixRaviart(object):
    """Python implementation of the Crouzeix-Raviart element."""

    def __init__(self, polygon):
        """Constructor"""
        self.Ns = []
        self.dofs = []
        self.polygon = polygon
        self.compute_basis_functions()

    def compute_basis_functions(self):
        """Compute the basis functions and degrees of freedom
        and put them in Ns and dofs, respectively."""

        # create the polynomial space
        N, variables, basis = bernstein(1,self.polygon,"a")
```

```
        # define the degrees of freedom
        for i in range(0,3):
            edge = self.polygon.line(i)
            dofi = edge.integrate(N)
            self.dofs.append(dofi)

        # compute and solve the system of linear equations
        for i in range(0,3):
            equations = []
                for j in range(0,3):
                    equations.append(self.dofs[j] == dirac(i,j))
                    sub = lsolve(equations, variables)
                    Ni = N.subs(sub)
                    self.Ns.append(Ni);

    def N(self,i): return self.Ns[i]
    def dof(self,i): return self.dofs[i]
    def nbf(self): return len(self.Ns)
```

The process of defining a finite element in SyFi is similar for all elements. As the above code shows, it resembles the Ciarlet definition closely, see also the Chapters 3 and 4. First, we construct a polynomial space. In the code above, this is performed by calling the `bernstein` function. The `bernstein` function takes as input a simplex and the order of the Bernstein polynomial. Arbitrary order polynomials are supported. This function produces a tuple consisting of the polynomial, its coefficients (or degrees of freedom), and the basis functions representing the polynomial space P:

Python code

```
In  : bernstein(1, triangle, "a")
Out : [-a0_2*(-1+y+x)+y*a0_0+x*a0_1, [a0_0, a0_1, a0_2], [y, x, 1-y-x]]
```

In the above code, we used a triangle and the order of the polynomial was one. The next task is to define a set of degrees of freedom; that is, a set of functionals $L_i : P \to \mathbb{R}$. For the Crouzeix–Raviart element, the degrees of freedom are simply the integrals over an edge; that is, $L_i(P) = \int_{E_i} P \, \mathrm{d}x$, where E_i for $i = (0,1,2)$ are the edges of the triangle. Alternatively we could have used the value at the midpoint of the edges since the polynomial P is linear. Finally, the different basis functions $\{N_i\}$ are determined by the set of equations $L_i(N_j) = \delta_{ij}$. These equations are then solved, using `lsolve`, to compute the basis functions of the elements; that is, the coefficients `[a0_0, a0_1, a0_2]` are determined for each specific basis function.

The basis functions of this element can then be displayed as follows:

Python code

```
p0 = [0,0,0]; p1 = [1,0,0]; p2 = [0,1,0];
triangle = Triangle(p0, p1, p2)

fe = CrouzeixRaviart(triangle)
for i in range(0,fe.nbf()):
  print "N(%d)       = "%i,   fe.N(i)
  print "grad(N(%d)) = "%i,   grad(fe.N(i))
```

giving the following output:

C++ code

```
N(0)       =  1/6*(-3+3*x+3*y+z)*2**(1/2)+1/6*2**(1/2)*(3*x-z)+1/6*2**(1/2)*(3*y-z)
grad(N(0)) =  [[2**(1/2)],[2**(1/2)],[-1/6*2**(1/2)]]
N(1)       =  1-2*x-1/3*z
```

```
grad(N(1)) =  [[-2],[0],[-1/3]]
...
```

15.3 SFC: SyFi form compiler

As mentioned earlier, SFC translates UFL code to UFC code. Consider the following UFL code for defining the variational problem for solving the Poisson problem, implemented in a file `Poisson.ufl`:

Python code

```
cell = triangle
element = FiniteElement("Lagrange", cell, 1)

u = TrialFunction(element)
v = TestFunction(element)
c = Coefficient(element)
f = Coefficient(element)
g = Coefficient(element)

a = c*dot(grad(u),grad(v))*dx
L = -f*v*dx + g*v*ds
```

SFC translates this UFL form to UFC code as follows:

Bash code

```
sfc -w1  -ogenerated_code Poisson.ufl
```

Here, `-w1` means that DOLFIN wrappers are generated, while `-ogenerated_code` means that the generated code should be located in the directory `generated_code`. Notice that the flags and corresponding options are not separated by spaces. A complete list of options is obtained with `sfc -h`. The generated code can be utilized in DOLFIN in a standard fashion. For a complete example consider the demo `demo/Poisson2D/cpp` that comes with the SyFi package.

In DOLFIN, the form compiler may be chosen at run-time by setting:

Python code

```
parameters["form_compiler"]["name"] = "sfc"
```

The form compiler can be tuned with a range of options. A complete list of options is obtained as follows:

Python code

```
from sfc.common.options import default_options
sfc_options = default_options()
```

The object `sfc_options` is of type `ParameterDict`, which is a dictionary with some additional functionality. One may use the following forms to set options before passing them to `assemble`.

Python code

```
sfc_options.code.integral.integration_method = "symbolic" # default is "quadrature"
# alternatively:
# sfc_options["code"]["integral"]["integration_method"] = "symbolic"
A = assemble(a, form_compiler_parameters=sfc_options)
```

Earlier versions of SFC produced slow code for complicated nonlinear equations as shown in Alnæs and Mardal (2010). Furthermore, the code generation was expensive both in terms of memory

and the number of operations required in the computations, because the SFC implementation did not scale linearly with the complexity of the equations. However, a significant speed-up came with the introduction of UFL with its expression tree traversal algorithms. Now, quite complicated equations can be handled without losing computational efficiency. Consider for example an elasticity problem where the constitutive law is a quite complicated variant of Fung (1993), described by the following equations:

$$F = I + (\operatorname{grad} u), \tag{15.1}$$
$$C = F^{\top} : F, \tag{15.2}$$
$$E = (C - I)/2, \tag{15.3}$$
$$\psi = \frac{\lambda}{2} \operatorname{tr}(E)^2 + K \exp((EA, E)), \tag{15.4}$$
$$P = \frac{\partial \psi}{\partial E}, \tag{15.5}$$
$$L = \int_{\Omega} P : (\operatorname{grad} v) \, \mathrm{d}x, \tag{15.6}$$
$$J_F = \frac{\partial L}{\partial u}. \tag{15.7}$$

Here, u is the unknown displacement, v is a test function, I is the identity matrix, A is a matrix, λ and K are material parameters, L is the system of nonlinear equations to be solved, and J_F is the corresponding Jacobian. This variational form is implemented in DOLFIN as follows:

Python code

```
mesh = UnitSquare(N, N)
V = VectorFunctionSpace(mesh, "Lagrange", order)
Q = FunctionSpace(mesh, "Lagrange", order)

U = Function(V)

v = TestFunction(V)
u = TrialFunction(V)

lamda = Constant(1.0)
A = Expression ((("1.0", "0.3"), ("0.3", "2.3")))
K = Constant(1.0)
n = U.cell().n

I = Identity(U.cell().d)
F = I + grad(U)
J = det(F)

C = F.T*F
E = (C-I)/2
E = variable(E)

psi = lamda/2 * tr(E)**2  +  K*exp(inner(A*E,E))
P = F*diff(psi, E)
a_f = psi*dx
L = inner(P, grad(v))*dx
J = derivative(L, U, u)

A = assemble(J)
```

N	100	200	400
$J_L, p = 1$	0.08	0.27	1.04
$J_L, p = 2$	0.36	1.41	5.45
$J_F, p = 1$	0.33	0.84	3.36
$J_F, p = 2$	0.68	2.26	8.55

Table 15.1: Comparison of the time (in seconds) for computing the Jacobian matrix for the two elasticity problems on a $N \times N$ unit square mesh for linear ($p = 1$) and quadratic elements ($p = 2$).

To test the computational efficiency of the generated code for this problem, we compare the assembly of J_F with the assembly of a corresponding linear elasticity problem with the following matrix J_L:

$$J_L = \int_\Omega \lambda \operatorname{div} u \operatorname{div} v + (\lambda + \mu) \operatorname{grad} u : \operatorname{grad} v \, \mathrm{d}x. \tag{15.8}$$

In Table 15.1 we see a comparison of the efficiency for the above examples. Clearly, the nonlinear example is no more than 4 times as slow as the linear problem when using linear elements, and only a factor 2 when using quadratic elements.

We refer to Alnæs and Mardal (2010); Ølgaard et al. (2008); Kirby and Logg (2008); Ølgaard and Wells (2010) for more discussions on the topic of efficient compilation of linear and nonlinear variational formulations.

15.4 Code generation design

Finally, we briefly describe the overall design of the code generation software. UFC defines the interface of the code produced by SFC. In SFC, each UFC class is mirrored by subclasses of the class `CodeGenerator` called `FormCG`, `DofMapCG`, `FiniteElementCG`, and `CellIntegralCG`, etc. These classes are used to generate code for the corresponding UFC classes, `form`, `dofmap`, `finite_element`, `cell_integral`, etc. These classes have a common function for generating the code, called `generate_code_dict`. The function `generate_code_dict` generates a dictionary containing named pieces of UFC code, most of which are function body implementations. This dictionary with code is then combined with format strings from the UFC utility Python module to generate UFC compliant code. An example of a format string is shown below.

Python code

```
cell_integral_implementation = """\
/// Constructor
%(classname)s::%(classname)s() : ufc::cell_integral()
{
%(constructor)s
}

/// Destructor
%(classname)s::~%(classname)s()
{
%(destructor)s
}

/// Tabulate the tensor for the contribution from a local cell
void %(classname)s::tabulate_tensor(double* A,
                                    const double * const * w,
                                    const ufc::cell& c) const
{
```

```
%(tabulate_tensor)s
}
"""
```

Using this template, the code generation in SFC is then performed as follows:

Python code

```
def generate_cell_integral_code(integrals, formrep):
    sfc_debug("Entering generate_cell_integral_code")
    itgrep = CellIntegralRepresentation(integrals, formrep)

    cg = CellIntegralCG(itgrep)
    vars = cg.generate_code_dict()
    supportcode = cg.generate_support_code()

    hcode = ufc_utils.cell_integral_header % vars
    ccode = supportcode + "\n"*3 + ufc_utils.cell_integral_implementation % vars

    includes = cg.hincludes() + cg.cincludes()

    system_headers = common_system_headers()

    hincludes = "\n".join('#include "%s"' % inc for inc in cg.hincludes())
    cincludes =  "\n".join('#include <%s>' % f for f in system_headers)
    cincludes += "\n"
    cincludes += "\n".join('#include "%s"' % inc for inc in cg.cincludes())

    hcode = _header_template         % \
             { "body": hcode, "name": itgrep.classname, "includes": hincludes }
    ccode = _implementation_template % \
             { "body": ccode, "name": itgrep.classname, "includes": cincludes }
    sfc_debug("Leaving generate_cell_integral_code")
    return itgrep.classname, (hcode, ccode), includes
```

As seen above, the `CellIntegralCG` class is again mirrored by a corresponding class `CellIntegralRepresentation`,

Python code

```
class CellIntegralRepresentation(IntegralRepresentation):
    def __init__(self, integrals, formrep):
        IntegralRepresentation.__init__(self, integrals, formrep, False)

    def compute_A(self, data, iota, facet=None):
        "Compute expression for A[iota]."

        if data.integration_method == "quadrature":
            if self.options.safemode:
                integrand = data.integral.integrand()
                data.evaluator.update(iota)
                integrand = data.evaluator.visit(integrand)
            else:
                n = len(data.G.V())
                integrand = data.vertex_data_set[iota][n-1]

            D = self.formrep.D_sym
            A = integrand * D
            ...
```

The representation classes are quite involved, in particular when using quadrature where the generated code involves multiple loops and quite a few temporary variables. To generate quadrature based

code, the computational graph algorithms from UFL (in particular the class `ufl.algorithms.Graph`) are used to split the expression tree into smaller subexpressions. SFC makes GiNaC symbols that represent temporary variables for all subexpressions. To place the temporary variables inside the correct loops in the generated code, the computational graph is partitioned based on the dependencies of subexpressions. See Chapter 17 for an explanation of the partitioning algorithm provided by UFL. The subexpression associated with each temporary variable is then translated to a C/C++ string using GiNaC. Finally, SFC puts it all together into a tabulate tensor implementation in the code generation classes (`*CG`).

16 UFC: a finite element code generation interface

By Martin Sandve Alnæs, Anders Logg and Kent-Andre Mardal

A central component of FEniCS is the UFC interface (Unified Form-assembly Code). UFC is an interface between problem-specific and general-purpose components of finite element programs. In particular, the UFC interface defines the structure and signature of the code that is generated by the form compilers FFC and SFC for DOLFIN. The UFC interface applies to a wide range of finite element problems (including mixed finite elements and discontinuous Galerkin methods) and may be used with libraries that differ widely in their design. For this purpose, the interface does not depend on any other FEniCS components (or other libraries) and consists only of a minimal set of abstract C++ classes using plain C arrays for data transfer. This chapter gives a short overview of the UFC interface. For a more comprehensive discussion, we refer to the UFC manual (Alnæs et al., 2007) and the paper Alnæs et al. (2009).

16.1 Overview

A key step in the solution of partial differential equations by the finite element method is the assembly of linear and nonlinear systems of equations. The implementation of such solvers is much helped by the existence of generic software libraries that provide data structures and algorithms for computational meshes and linear algebra. This allows the implementation of a generic assembly algorithm that may be partly reused from one application to another. However, since the inner loop of the assembly algorithm inherently depends on the partial differential equation being solved and the finite elements used to produce the discretization, this inner loop must typically be supplied by the user. Writing the inner loop is a challenging task that is prone to errors, and which prohibits rapid prototyping and experimentation with models and discretization methods.

The FEniCS toolchain of FIAT–UFC–FFC/SFC–UFC–DOLFIN is an attempt to solve this problem. By generating automatically the inner loop based on a high-level description of the finite element variational problem (in the UFL form language), FEniCS is able to provide a completely generic implementation of the assembly algorithm as part of DOLFIN. This is illustrated in Figure 16.1. We note from this figure that the user input is partitioned into two sets: a first subset consisting of the finite element variational problem that requires code generation, and a second subset consisting of the mesh and coefficient data that is given as input to the assembler.

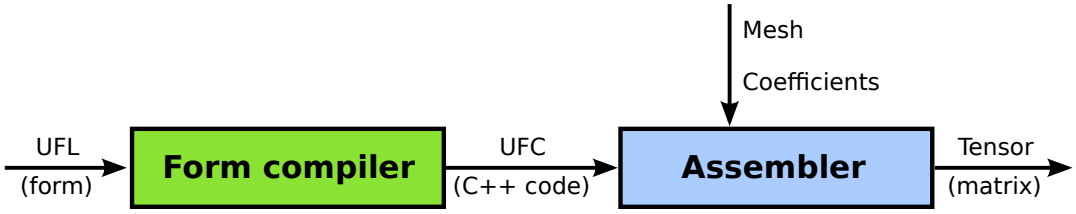


Figure 16.1: A flow diagram of finite element assembly in FEniCS.

16.2 Finite element discretization and assembly

In Chapter 6, we described the assembly algorithm for computing the global rank ρ tensor A corresponding to a multilinear form a of arity ρ:

$$\begin{aligned} &a : W_{1,h} \times W_{2,h} \times \cdots \times W_{n,h} \times V_{\rho,h} \times \cdots \times V_{2,h} \times V_{1,h} \to \mathbb{R}, \\ &a \mapsto a(w_1, w_2, \ldots, w_n; v_\rho, \ldots, v_2, v_1). \end{aligned} \tag{16.1}$$

Here, $\{V_{j,h}\}_{j=1}^{\rho}$ is a sequence of discrete function spaces for the *arguments* $\{v_j\}_{j=1}^{\rho}$ of the form and $\{W_{j,h}^{j}\}_{j=1}^{n}$ is a sequence of discrete function spaces for the *coefficients* $\{w_j\}_{j=1}^{n}$ of the form. Typically, the arity is $\rho = 1$ for a linear form or $\rho = 2$ for a bilinear form. In the simplest case, all function spaces are equal but there are many important examples, such as mixed methods, where the arguments come from different function spaces. The choice of coefficient function spaces depends on the application; a polynomial basis simplifies exact integration, while in some cases evaluating coefficients at quadrature points may be required.

As we saw in Chapter 6, the global tensor A can be computed by summing contributions from the cells and facets of a mesh. We refer to these contributions as either *cell tensors* or *facet tensors*. Although one may formulate a generic assembly algorithm, the cell and facet tensors must be computed differently depending on the variational form, and their entries must be inserted differently into the global tensor depending on the choice of finite element spaces. This is handled in FEniCS by implementing a generic assembly algorithm (as part of DOLFIN) that relies on special-purpose generated code (by FFC or SFC) for computing the cell and facet tensors, and for computing the local-to-global map for insertion of the cell and facet tensors into the global matrix.

The UFC interface assumes that the multilinear form a in (16.1) can be expressed as a sum of integrals over the cells $\mathcal{T}_h$, the exterior facets ∂_h, and the interior facets ∂_h^0 of the mesh. The integrals may then be expressed on disjoint subsets $\mathcal{T}_h = \cup_{k=1}^{n_c} \mathcal{T}_{h,k}$, $\partial_h = \cup_{k=1}^{n_f} \partial_{h,k}$ and $\partial_h^0 = \cup_{k=1}^{n_f^0} \partial_{h,k}^0$, respectively. In particular, it is assumed that the multilinear form can be expressed in the following canonical form:

$$\begin{aligned} a(w_1, w_2, \ldots, w_n; v_\rho, \ldots, v_2, v_1) = &\sum_{k=1}^{n_c} \sum_{T \in \mathcal{T}_{h,k}} \int_T I_k^c(w_1, w_2, \ldots, w_n; v_\rho, \ldots, v_2, v_1) \, \mathrm{d}x \\ &+ \sum_{k=1}^{n_f} \sum_{S \in \partial_{h,k}} \int_S I_k^f(w_1, w_2, \ldots, w_n; v_\rho, \ldots, v_2, v_1) \, \mathrm{d}s \\ &+ \sum_{k=1}^{n_f^0} \sum_{S^0 \in \partial_{h,k}^0} \int_{S^0} I_k^{f,0}(w_1, w_2, \ldots, w_n; v_\rho, \ldots, v_2, v_1) \, \mathrm{d}S. \end{aligned} \tag{16.2}$$

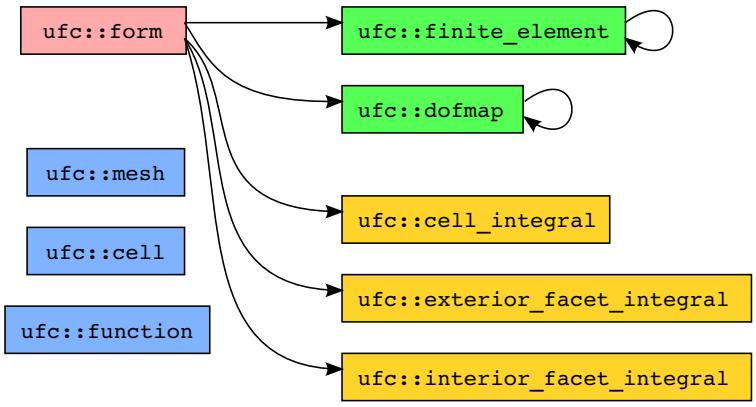


Figure 16.2: Schematic overview of some of the UFC classes. Arrows indicate dependencies.

We refer to an integral I_k^c over a cell T as a *cell integral*, an integral I_k^f over an exterior facet S as an *exterior facet integral* (typically used to implement Neumann and Robin type boundary conditions), and to an integral $I_k^{f,0}$ over an interior facet S^0 as an *interior facet integral* (typically used in discontinuous Galerkin methods).

16.3 The UFC interface

The UFC interface[1] consists of a small collection of abstract C++ classes that represent common components for assembling tensors using the finite element method. The full UFC interface is specified in a single header file `ufc.h`. The UFC classes are accompanied by a set of conventions for numbering of cell data and other arrays. Data is passed as plain C arrays for efficiency and minimal dependencies. Most functions are declared `const`, reflecting that the operations they represent should not change the outcome of future operations.[2]

16.3.1 Class relations

Figure (16.2) shows all UFC classes and their relations. The classes `mesh`, `cell` and `function` provide the means for communicating mesh and coefficient function data as arguments. The integrals of (16.2) are represented by one of the following classes:

- `cell_integral`,
- `exterior_facet_integral`,
- `interior_facet_integral`.

Subclasses of `form` must implement factory functions which may be called to create integral objects. These objects in turn know how to compute their respective contribution from a cell or facet during assembly. A code fragment from the `form` class declaration is shown below.

C++ code

```
class form
{
public:

  ...
```

[1] This chapter describes version 2.0 of the UFC interface.
[2] The exceptions are the functions to initialize a `dofmap`.

```
  /// Create a new cell integral on sub domain i
  virtual cell_integral* create_cell_integral(unsigned int i) const = 0;

  /// Create a new exterior facet integral on sub domain i
  virtual exterior_facet_integral* create_exterior_facet_integral(unsigned int i) const = 0;

  /// Create a new interior facet integral on sub domain i
  virtual interior_facet_integral* create_interior_facet_integral(unsigned int i) const = 0;

};
```

The `form` class also specifies functions for creating `finite_element` and `dofmap` objects for the finite element function spaces $\{V_{j,h}\}_{j=1}^{\rho}$ and $\{W_{j,h}\}_{j=1}^{n}$ of the variational form. The `finite_element` object provides functionality such as evaluation of degrees of freedom and evaluation of basis functions and their derivatives. The `dofmap` object provides functionality such as tabulating the local-to-global map of degrees of freedom on a single element, as well as tabulation of subsets associated with particular mesh entities, which are used to apply Dirichlet boundary conditions and build connectivity information.

Both the `finite_element` and `dofmap` classes can represent mixed elements, in which case it is possible to obtain `finite_element` and `dofmap` objects for each subelement in a hierarchical manner. Vector elements composed of scalar elements are in this context seen as special cases of mixed elements where all subelements are equal. As an example, consider the `dofmap` for a $\mathcal{P}_2$–$\mathcal{P}_1$ Taylor–Hood element. From this `dofmap` it is possible to extract one `dofmap` for the quadratic vector element and one `dofmap` for the linear scalar element. From the vector element, a `dofmap` for the quadratic scalar element of each vector component can be obtained. This can be used to access subcomponents from the solution of a mixed system.

16.3.2 Stages in the assembly algorithm

Next, we focus on a few key parts of the interface and explain how these can be used to implement the assembly algorithm presented in Chapter 6. The general algorithm consists of three stages: (i) assembling the contributions from all cells, (ii) assembling the contributions from all exterior facets and (iii) assembling the contributions from all interior facets.

Each of the three assembly stages (i)–(iii) is further composed of five steps. In the first step, a cell T is fetched from the mesh, typically implemented by filling a `cell` structure (see Figure 16.3) with coordinate data and global numbering of the mesh entities in the cell. This step depends on the specific mesh being used.

In the second step, the coefficients in $\{W_{j,h}\}_{j=1}^{n}$ are restricted to the local cell T. If a coefficient w_j is not given as a linear combination of basis functions for $W_{j,h}$, it must at this step be interpolated into $W_{j,h}$, using the interpolant defined by the degrees of freedom of $W_{j,h}$. One common choice of interpolation is point evaluation at the set of nodal points. In this case, the coefficient function is passed as an implementation of the `function` interface (a simple functor) to the function `evaluate_dofs` in the UFC `finite_element` class.

Figure 16.3: Data structure for communicating cell data.

C++ code

```
class cell
{
public:

  /// Constructor
  cell(): cell_shape(interval),
          topological_dimension(0), geometric_dimension(0),
          entity_indices(0), coordinates(0),
          index(0), local_facet(-1), mesh_identifier(-1) {}

  /// Destructor
  virtual ~cell() {}

  /// Shape of the cell
  shape cell_shape;

  /// Topological dimension of the mesh
  unsigned int topological_dimension;

  /// Geometric dimension of the mesh
  unsigned int geometric_dimension;

  /// Array of global indices for the mesh entities of
  /// the cell
  unsigned int** entity_indices;

  /// Array of coordinates for the vertices of the cell
  double** coordinates;

  /// Cell index (short-cut for
  /// entity_indices[topological_dimension][0])
  unsigned int index;

  /// Local facet index
  int local_facet;

  /// Unique mesh identifier
  int mesh_identifier;

};
```

The `evaluate_dofs` function has the following signature:

C++ code

```
/// Evaluate linear functionals for all dofs on the function f
virtual void evaluate_dofs(double* values,
                           const function& f,
                           const cell& c) const = 0;
```

Here, `double* values` is a pointer to the first element of an array of double precision floating point numbers which will be filled with the values of the degrees of freedom of the function `f` on the current cell.

In the third step, the local-to-global map of degrees of freedom is tabulated for each of the function spaces. That is, for each of the local discrete finite element spaces on T, we tabulate the corresponding global degrees of freedom. This is handled by the `tabulate_dofs` function:

C++ code

```
/// Tabulate the local-to-global mapping of dofs on a cell
virtual void dofmap::tabulate_dofs(unsigned int* dofs,
                                   const mesh& m,
                                   const cell& c) const = 0;
```

Here, `unsigned int* dofs` is a pointer to the first element of an array of unsigned integers that will be filled with the local-to-global map on the current cell during the function call.

In the fourth step, the local element tensor contribution (cell or exterior/interior facet tensor) is computed. This is done by a call to the function `tabulate_tensor`, illustrated below for a cell integral.

C++ code

```
/// Tabulate the tensor for the contribution from a local cell
virtual void tabulate_tensor(double* A,
                             const double * const * w,
                             const cell& c) const = 0;
```

Here, `double* A` is a pointer to the first element of an array of double precision floating point numbers which will be filled with the values of the element tensor, flattened into one array of numbers. Similarly, one may evaluate interior and exterior facet contributions using slightly different function signatures.

Finally, at the fifth step, the local element tensor contributions are added to the global tensor, using the local-to-global maps previously obtained by calls to the `tabulate_dofs` function. This is an operation that depends on the linear algebra backend used to store the global tensor.

16.3.3 Code generation utilities

UFC provides a number of utilities that can be used by form compilers to simplify the code generation process, including templates for creating subclasses of UFC classes and utilities for just-in-time compilation. These are distributed as part of the `ufc_utils` Python module.

Templates are available for all UFC classes listed in Figure 16.2 and consist of format strings for the skeleton of each subclass. The following code illustrates how to generate a subclass of the UFC `form` class.

Python code

```
from ufc_utils import form_combined

implementation = {}
implementation["classname"] = "my_form"
implementation["members"] = ""
```

```
implementation["constructor_arguments"] = ""
implementation["initializer_list"] = ""
implementation["constructor"] = "// Do nothing"
implementation["destructor"] = "// Do nothing"
implementation["signature"] = 'return "my form"'
implementation["rank"] = "return 2;"
implementation["num_coefficients"] = "return 0;"
implementation["num_cell_domains"] = "return 3;"
implementation["num_interior_facet_domains"] = "return 1;"
implementation["num_exterior_facet_domains"] = "return 0;"
implementation["create_finite_element"] = """
switch (i)
{
case 0:
  return new my_finite_element_0();
case 1:
  return new my_finite_element_1();
default:
  return 0;
}"""
implementation["create_dofmap"] = """
switch (i)
{
case 0:
  return new my_dofmap_0();
case 1:
  return new my_dofmap_1();
default:
  return 0;
}"""
implementation["create_cell_integral"] = """
switch (i)
{
case 0:
  return new my_cell_integral_0();
case 1:
  return new my_cell_integral_1();
case 2:
  return new my_cell_integral_2();
default:
  return 0;
}"""
implementation["create_exterior_facet_integral"] = \
    "return new my_exterior_facet_integral();"
implementation["create_interior_facet_integral"] = "return 0;"

print form_combined % implementation
```

This generates code for a single header file that also contains the implementation of each function in the UFC `form` interface. It is also possible to generate code for separate header (`.h`) and implementation (`.cpp`) files by using the `form_header` and `form_implementation` templates.

The `ufc_utils` module also contains the utility function `build_ufc_module` that can be called to build a Python module based on generated UFC code. This process involves compilation, linking and loading of the generated C++ code as well as generating a Python wrapper module using Instant/SWIG as described in Chapter 14.

16.4 Examples

In this section, we demonstrate how UFC can be used in practice for assembly of finite element forms. First, we demonstrate how one may implement a simple assembler based on generated UFC code. We then show examples of input to the form compilers FFC and SFC as well as part of the corresponding UFC code generated as output.

16.4.1 Assembler

The simple assembler presented in this section assumes that the degrees of freedom of the finite element spaces involved depend only on vertices; that is, we assume piecewise linear elements. We also assume that the assembled form is a tensor of rank two (a matrix), that we may insert values into the given matrix data structure by simply assigning values to the entries of the matrix, that the form does not depend on any coefficients, and that the form is expressed as a single cell integral. In practice, the efficient insertion of entries into a sparse matrix typically requires a the use of a special optimized library call. For example, entries may be inserted (added) to a sparse PETSc matrix by a call to `MatSetValues` and to a sparse Trilinos/Epetra matrix by a call to `SumIntoGlobalValues`. For a complete implementation of an assembler for general rank tensors and generic linear algebra libraries that provide an insertion operation, we refer to the class `Assembler` in DOLFIN (`Assembler.cpp`). The code example presented below is available in the supplementary material for this chapter (`assemble.cpp`).

C++ code

```
void assemble(Matrix& A, ufc::form& form, dolfin::Mesh& mesh)
{
  // Get dimensions
  const uint D = mesh.topology().dim();
  const uint d = mesh.geometry().dim();

  // Initialize UFC mesh data structure
  ufc::mesh ufc_mesh;
  ufc_mesh.topological_dimension = D;
  ufc_mesh.geometric_dimension = d;
  ufc_mesh.num_entities = new uint[D + 1];
  for (uint i = 0; i <= D; i++)
    ufc_mesh.num_entities[i] = 0;
  ufc_mesh.num_entities[0] = mesh.num_vertices();
  ufc_mesh.num_entities[D] = mesh.num_cells();

  // Initialize UFC cell data structure, assuming that the
  // cell is a simplex and only vertices are used for dofs
  ufc::cell ufc_cell;
  switch (D)
  {
  case 1:
    ufc_cell.cell_shape = ufc::interval;
    break;
  case 2:
    ufc_cell.cell_shape = ufc::triangle;
    break;
  default:
    ufc_cell.cell_shape = ufc::tetrahedron;
    break;
  }
  ufc_cell.topological_dimension = D;
  ufc_cell.geometric_dimension = d;
```

```
ufc_cell.entity_indices = new uint * [D + 1];
for (uint i = 0; i <= D; i++)
  ufc_cell.entity_indices[i] = 0;
uint vertices_per_cell = D + 1;
ufc_cell.entity_indices[0] = new uint[vertices_per_cell];
ufc_cell.entity_indices[D] = new uint[1];
ufc_cell.coordinates = new double * [vertices_per_cell];
for (uint i = 0; i <= D; i++)
  ufc_cell.coordinates[i] = new double[d];

// Create cell integrals, assuming there is only one
ufc::cell_integral* cell_integral = form.create_cell_integral(0);

// Create dofmaps for rows and columns
ufc::dofmap* dofmap_0 = form.create_dofmap(0);
ufc::dofmap* dofmap_1 = form.create_dofmap(1);

// Initialize dofmaps
dofmap_0->init_mesh(ufc_mesh);
dofmap_1->init_mesh(ufc_mesh);

// Omitting code for dofmap initialization on cells, which is not
// needed for code generated by FFC but which is generally required

// Get local and global dimensions
uint m = dofmap_0->max_local_dimension();
uint n = dofmap_1->max_local_dimension();
uint M = dofmap_0->global_dimension();
uint N = dofmap_1->global_dimension();

// Initialize array of local-to-global maps
uint* dofs_0 = new uint[m];
uint* dofs_1 = new uint[n];

// Initialize array of values for the cell matrix
double* A_T = new double[m * n];

// Initialize global matrix
A.init(M, N);

// Iterate over the cells of the mesh
for (dolfin::CellIterator cell(mesh); !cell.end(); ++cell)
{
  // Update UFC cell data structure for current cell
  ufc_cell.entity_indices[D][0] = cell->index();
  for (dolfin::VertexIterator vertex(*cell); !vertex.end(); ++vertex)
  {
    ufc_cell.entity_indices[0][vertex.pos()] = vertex->index();
    for (uint i = 0; i < d; i++)
      ufc_cell.coordinates[vertex.pos()][i] = vertex->x(i);
  }

  // Compute local-to-global map for degrees of freedom
  dofmap_0->tabulate_dofs(dofs_0, ufc_mesh, ufc_cell);
  dofmap_1->tabulate_dofs(dofs_1, ufc_mesh, ufc_cell);

  // Compute the cell matrix A_T
  cell_integral->tabulate_tensor(A_T, 0, ufc_cell);

  // Add entries to global matrix
  for (uint i = 0; i < m; i++)
```

```
      for (uint j = 0; j < m; j++)
        A(dofs_0[i], dofs_1[j]) += A_T[i*n + j];
  }

  // Omitting code for deleting allocated arrays
}
```

16.4.2 Generated UFC code

The form language UFL described in Chapter 17 provides a simple language for specification of variational forms, which may be entered either directly in Python or in text files given to a form compiler. We consider the following definition of the bilinear form $a(u, v) = \langle \nabla u, \nabla v \rangle$ in UFL:

UFL code

```
element = FiniteElement("Lagrange", triangle, 1)

u = TrialFunction(element)
v = TestFunction(element)

a = inner(grad(u), grad(v))*dx
```

When compiling this code, a C++ header file is created, containing UFC code that may be used to assemble the global sparse stiffness matrix for Poisson's equation. Below, we present the code generated for evaluation of the element stiffness matrix for the bilinear form a using FFC. Similar code may be generated using SFC.

C++ code

```
/// Tabulate the tensor for the contribution from a local cell
virtual void tabulate_tensor(double* A,
                             const double * const * w,
                             const ufc::cell& c) const
{
  // Number of operations (multiply-add pairs) for Jacobian data:      11
  // Number of operations (multiply-add pairs) for geometry tensor:    8
  // Number of operations (multiply-add pairs) for tensor contraction: 11
  // Total number of operations (multiply-add pairs):                  30

  // Extract vertex coordinates
  const double * const * x = c.coordinates;

  // Compute Jacobian of affine map from reference cell
  const double J_00 = x[1][0] - x[0][0];
  const double J_01 = x[2][0] - x[0][0];
  const double J_10 = x[1][1] - x[0][1];
  const double J_11 = x[2][1] - x[0][1];

  // Compute determinant of Jacobian
  double detJ = J_00*J_11 - J_01*J_10;

  // Compute inverse of Jacobian
  const double K_00 =  J_11 / detJ;
  const double K_01 = -J_01 / detJ;
  const double K_10 = -J_10 / detJ;
  const double K_11 =  J_00 / detJ;

  // Set scale factor
  const double det = std::abs(detJ);
```

```
  // Compute geometry tensor
  const double G0_0_0 = det*(K_00*K_00 + K_01*K_01);
  const double G0_0_1 = det*(K_00*K_10 + K_01*K_11);
  const double G0_1_0 = det*(K_10*K_00 + K_11*K_01);
  const double G0_1_1 = det*(K_10*K_10 + K_11*K_11);

  // Compute element tensor
  A[0] = 0.5*G0_0_0 + 0.5*G0_0_1 + 0.5*G0_1_0 + 0.5*G0_1_1;
  A[1] = -0.5*G0_0_0 - 0.5*G0_1_0;
  A[2] = -0.5*G0_0_1 - 0.5*G0_1_1;
  A[3] = -0.5*G0_0_0 - 0.5*G0_0_1;
  A[4] = 0.5*G0_0_0;
  A[5] = 0.5*G0_0_1;
  A[6] = -0.5*G0_1_0 - 0.5*G0_1_1;
  A[7] = 0.5*G0_1_0;
  A[8] = 0.5*G0_1_1;
}
```

Having computed the element tensor, one needs to compute the local-to-global map in order to know where to insert the local contributions in the global tensor. This map may be obtained by calling the member function `tabulate_dofs` of the class `dofmap`. FFC uses an implicit ordering scheme, based on the indices of the topological entities in the mesh. This information may be extracted from the `cell` attribute `entity_indices`. For linear Lagrange elements on triangles, each degree of freedom is associated with a global vertex. Hence, FFC constructs the map by picking the corresponding global vertex number for each degree of freedom as demonstrated below.

C++ code

```
/// Tabulate the local-to-global mapping of dofs on a cell
virtual void tabulate_dofs(unsigned int* dofs,
                           const ufc::mesh& m,
                           const ufc::cell& c) const
{
  dofs[0] = c.entity_indices[0][0];
  dofs[1] = c.entity_indices[0][1];
  dofs[2] = c.entity_indices[0][2];
}
```

For quadratic Lagrange elements, a similar map is generated based on global vertex and edge numbers (entities of dimension zero and one respectively). We list the code for `tabulate_dofs` generated by FFC for quadratic Lagrange elements below.

C++ code

```
/// Tabulate the local-to-global mapping of dofs on a cell
virtual void tabulate_dofs(unsigned int* dofs,
                           const ufc::mesh& m,
                           const ufc::cell& c) const
{
  unsigned int offset = 0;
  dofs[0] = offset + c.entity_indices[0][0];
  dofs[1] = offset + c.entity_indices[0][1];
  dofs[2] = offset + c.entity_indices[0][2];
  offset += m.num_entities[0];
  dofs[3] = offset + c.entity_indices[1][0];
  dofs[4] = offset + c.entity_indices[1][1];
  dofs[5] = offset + c.entity_indices[1][2];
  offset += m.num_entities[1];
}
```

16.5 Numbering conventions

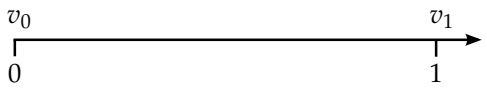
UFC relies on a set of numbering conventions for cells, vertices and other mesh entities. The numbering scheme ensures that form compilers (FFC and SFC) and assemblers (DOLFIN) can communicate data required for tabulating the cell and facet tensors as well as local-to-global maps.

16.5.1 Reference cells

The following five reference cells are covered by the UFC specification: the reference *interval*, the reference *triangle*, the reference *quadrilateral*, the reference *tetrahedron* and the reference *hexahedron*. The UFC specification assumes that each cell in a finite element mesh is always isomorphic to one of the reference cells. The UFC reference cells are listed in the table below.

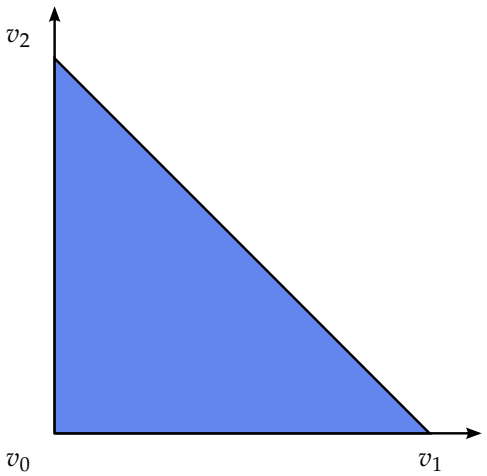
Reference cell	Dimension	#Vertices	#Facets
The reference interval	1	2	2
The reference triangle	2	3	3
The reference quadrilateral	2	4	4
The reference tetrahedron	3	4	4
The reference hexahedron	3	8	6

The reference interval. The reference interval and the coordinates of its two vertices are shown in the figure and table below.



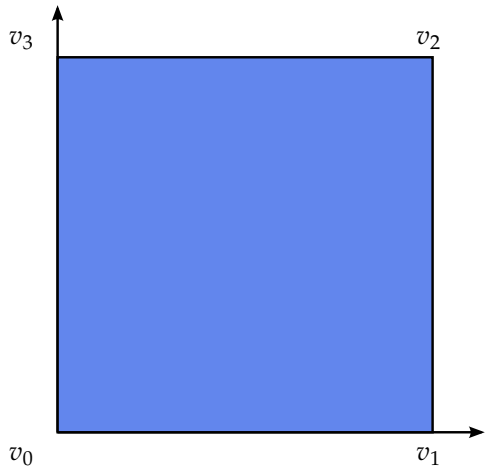
Vertex	Coordinates
v_0	$x = 0$
v_1	$x = 1$

The reference triangle. The reference triangle and the coordinates of its three vertices are shown in figure and table below.



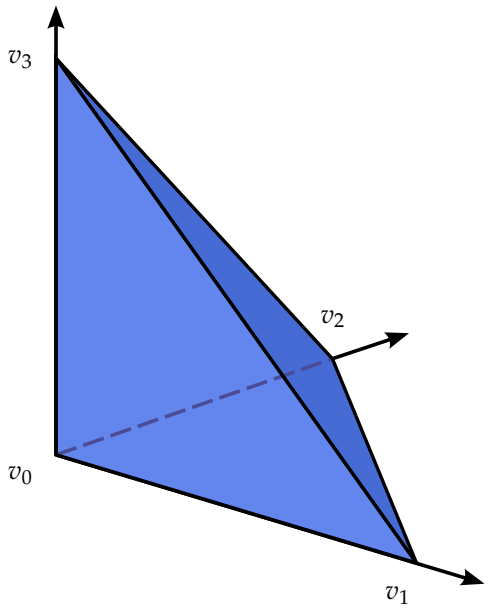
Vertex	Coordinates
v_0	$x = (0, 0)$
v_1	$x = (1, 0)$
v_2	$x = (0, 1)$

The reference quadrilateral. The reference quadrilateral and the coordinates of its four vertices are shown in the figure and table below.



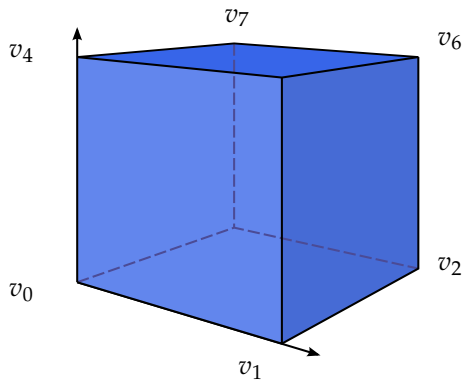
Vertex	Coordinates
v_0	$x = (0, 0)$
v_1	$x = (1, 0)$
v_2	$x = (1, 1)$
v_3	$x = (0, 1)$

The reference tetrahedron. The reference tetrahedron and the coordinates of its four vertices are shown in the figure and table below.



Vertex	Coordinates
v_0	$x = (0, 0, 0)$
v_1	$x = (1, 0, 0)$
v_2	$x = (0, 1, 0)$
v_3	$x = (0, 0, 1)$

The reference hexahedron. The reference hexahedron and the coordinates of its eight vertices are shown in the figure and table below.



Vertex	Coordinate
v_0	$x = (0, 0, 0)$
v_1	$x = (1, 0, 0)$
v_2	$x = (1, 1, 0)$
v_3	$x = (0, 1, 0)$
v_4	$x = (0, 0, 1)$
v_5	$x = (1, 0, 1)$
v_6	$x = (1, 1, 1)$
v_7	$x = (0, 1, 1)$

16.5.2 Numbering of mesh entities

The UFC specification dictates a certain numbering of the vertices, edges etc. of the cells of a finite element mesh. First, a specific numbering is picked for the vertices of each cell. Then, the remaining entities are ordered based on a simple rule, as described in detail below.

Basic concepts The topological entities of a cell (or mesh) are referred to as *mesh entities*. A mesh entity can be identified by a pair (d, i), where d is the topological dimension of the mesh entity and i is a unique index of the mesh entity. Mesh entities are numbered within each topological dimension from 0 to $n_d - 1$, where n_d is the number of mesh entities of topological dimension d.

For convenience, mesh entities of topological dimension 0 are referred to as *vertices*, entities of dimension 1 as *edges*, entities of dimension 2 as *faces*, entities of *codimension* 1 as *facets*, and entities of codimension 0 as *cells*. These concepts are summarized in the table below.

Entity	Dimension	Codimension
Vertex	0	D
Edge	1	$D - 1$
Face	2	$D - 2$
Facet	$D - 1$	1
Cell	D	0

Thus, the vertices of a tetrahedron are identified as $v_0 = (0, 0)$, $v_1 = (0, 1)$, $v_2 = (0, 2)$ and $v_3 = (0, 3)$, the edges are $e_0 = (1, 0)$, $e_1 = (1, 1)$, $e_2 = (1, 2)$, $e_3 = (1, 3)$, $e_4 = (1, 4)$ and $e_5 = (1, 5)$, the faces (facets) are $f_0 = (2, 0)$, $f_1 = (2, 1)$, $f_2 = (2, 2)$ and $f_3 = (2, 3)$, and the cell itself is $c_0 = (3, 0)$.

Numbering of vertices. For simplicial cells (intervals, triangles and tetrahedra) of a finite element mesh, the vertices are numbered locally based on the corresponding global vertex numbers. In particular, a tuple of increasing local vertex numbers corresponds to a tuple of increasing global vertex numbers. This is illustrated in Figure 16.4 for a mesh consisting of two triangles.

For non-simplicial cells (quadrilaterals and hexahedra), the numbering is arbitrary, as long as each cell is topologically isomorphic to the corresponding reference cell by matching each vertex with the corresponding vertex in the reference cell. This is illustrated in Figure 16.5 for a mesh consisting of two quadrilaterals.

Figure 16.4: The vertices of a simplicial mesh are numbered locally based on the corresponding global vertex numbers.

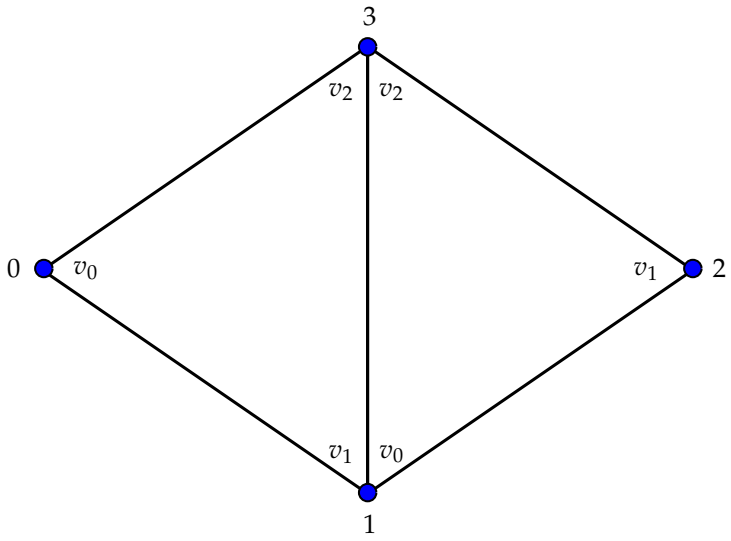
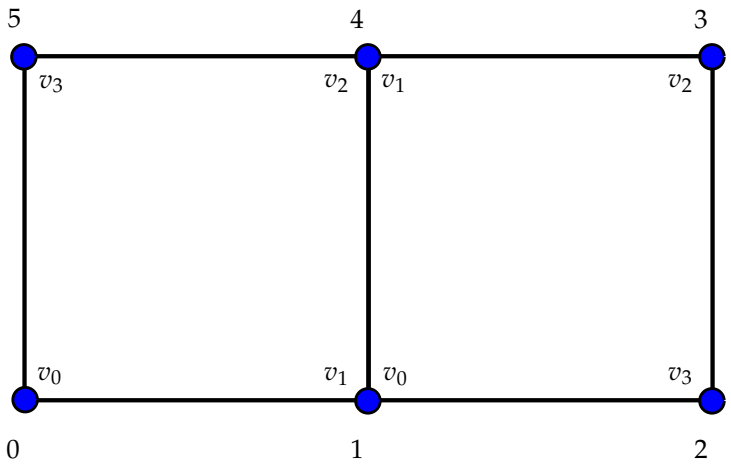


Figure 16.5: The local numbering of vertices of a non-simplicial mesh is arbitrary, as long as each cell is topologically isomorphic to the reference cell by matching each vertex to the corresponding vertex of the reference cell.



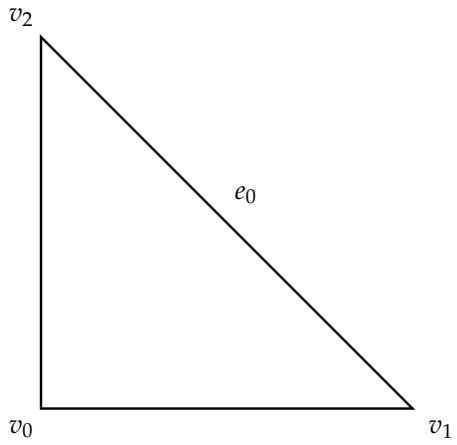


Figure 16.6: Mesh entities are ordered based on a lexicographical ordering of the corresponding ordered tuples of non-incident vertices. The first edge e_0 is non-incident to vertex v_0.

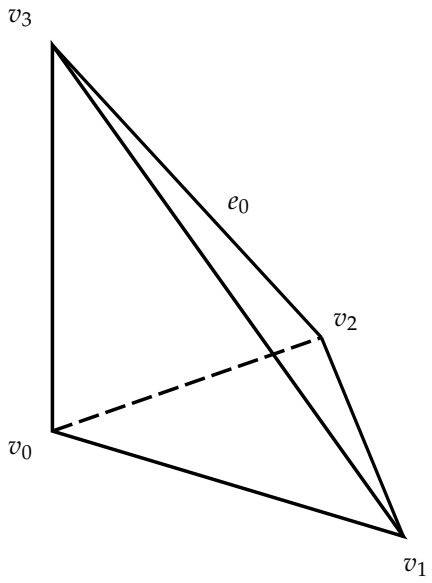


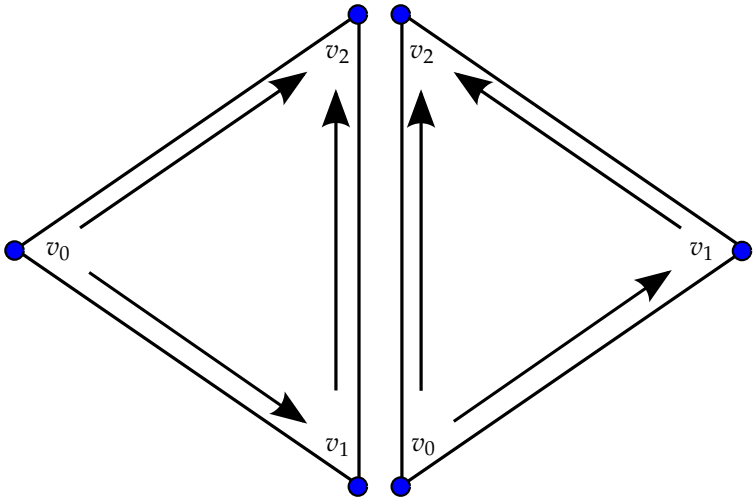
Figure 16.7: Mesh entities are ordered based on a lexicographical ordering of the corresponding ordered tuples of non-incident vertices. The first edge e_0 is non-incident to vertices v_0 and v_1.

Numbering of other mesh entities. When the vertices have been numbered, the remaining mesh entities are numbered within each topological dimension based on a *lexicographical ordering* of the corresponding ordered tuples of *non-incident vertices*.

As an illustration, consider the numbering of edges (the mesh entities of topological dimension one) on the reference triangle in Figure 16.6. To number the edges of the reference triangle, we identify for each edge the corresponding non-incident vertices. For each edge, there is only one such vertex (the vertex opposite to the edge). We thus identify the three edges in the reference triangle with the tuples (v_0), (v_1) and (v_2). The first of these is edge e_0 between vertices v_1 and v_2 opposite to vertex v_0, the second is edge e_1 between vertices v_0 and v_2 opposite to vertex v_1, and the third is edge e_2 between vertices v_0 and v_1 opposite to vertex v_2.

Similarly, we identify the six edges of the reference tetrahedron with the corresponding non-incident tuples (v_0, v_1), (v_0, v_2), (v_0, v_3), (v_1, v_2), (v_1, v_3) and (v_2, v_3). The first of these is edge e_0 between vertices v_2 and v_3 opposite to vertices v_0 and v_1 as shown in Figure 16.7.

Figure 16.8: Two incident triangles will always agree on the orientation of the common edge.



Relative ordering. The relative ordering of mesh entities with respect to other incident mesh entities follows by sorting the entities by their (global) indices. Thus, the pair of vertices incident to the first edge e_0 of a triangular cell is (v_1, v_2), not (v_2, v_1). Similarly, the first face f_0 of a tetrahedral cell is incident to vertices (v_1, v_2, v_3).

For simplicial cells, the relative ordering in combination with the convention of numbering the vertices locally based on global vertex indices means that two incident cells will always agree on the orientation of incident subsimplices. Thus, two incident triangles will agree on the orientation of the common edge and two incident tetrahedra will agree on the orientation of the common edge(s) and the orientation of the common face (if any). This is illustrated in Figure 16.8 for two incident triangles sharing a common edge. This leads to practical advantages in the assembly of higher-order, $H(\text{div})$ and $H(\text{curl})$ elements.

Limitations. The UFC specification is only concerned with the ordering of mesh entities with respect to entities of larger topological dimension. In other words, the UFC specification is only concerned with the ordering of incidence relations of the class $d - d'$ where $d > d'$. For example, the UFC specification is not concerned with the ordering of incidence relations of the class $0 - 1$; that is, the ordering of edges incident to vertices.

Numbering of mesh entities on intervals. The numbering of mesh entities on interval cells is summarized in the table below.

Entity	Incident vertices	Non-incident vertices
$v_0 = (0, 0)$	(v_0)	(v_1)
$v_1 = (0, 1)$	(v_1)	(v_0)
$c_0 = (1, 0)$	(v_0, v_1)	$\emptyset$

Numbering of mesh entities on triangular cells. The numbering of mesh entities on triangular cells is summarized in the table below.

Entity	Incident vertices	Non-incident vertices
$v_0 = (0, 0)$	(v_0)	(v_1, v_2)
$v_1 = (0, 1)$	(v_1)	(v_0, v_2)
$v_2 = (0, 2)$	(v_2)	(v_0, v_1)
$e_0 = (1, 0)$	(v_1, v_2)	(v_0)
$e_1 = (1, 1)$	(v_0, v_2)	(v_1)
$e_2 = (1, 2)$	(v_0, v_1)	(v_2)
$c_0 = (2, 0)$	(v_0, v_1, v_2)	$\emptyset$

Numbering of mesh entities on quadrilateral cells. The numbering of mesh entities on quadrilateral cells is summarized in the table below.

Entity	Incident vertices	Non-incident vertices
$v_0 = (0, 0)$	(v_0)	(v_1, v_2, v_3)
$v_1 = (0, 1)$	(v_1)	(v_0, v_2, v_3)
$v_2 = (0, 2)$	(v_2)	(v_0, v_1, v_3)
$v_3 = (0, 3)$	(v_3)	(v_0, v_1, v_2)
$e_0 = (1, 0)$	(v_2, v_3)	(v_0, v_1)
$e_1 = (1, 1)$	(v_1, v_2)	(v_0, v_3)
$e_2 = (1, 2)$	(v_0, v_3)	(v_1, v_2)
$e_3 = (1, 3)$	(v_0, v_1)	(v_2, v_3)
$c_0 = (2, 0)$	(v_0, v_1, v_2, v_3)	$\emptyset$

Numbering of mesh entities on tetrahedral cells. The numbering of mesh entities on tetrahedral cells is summarized in the table below.

Entity	Incident vertices	Non-incident vertices
$v_0 = (0, 0)$	(v_0)	(v_1, v_2, v_3)
$v_1 = (0, 1)$	(v_1)	(v_0, v_2, v_3)
$v_2 = (0, 2)$	(v_2)	(v_0, v_1, v_3)
$v_3 = (0, 3)$	(v_3)	(v_0, v_1, v_2)
$e_0 = (1, 0)$	(v_2, v_3)	(v_0, v_1)
$e_1 = (1, 1)$	(v_1, v_3)	(v_0, v_2)
$e_2 = (1, 2)$	(v_1, v_2)	(v_0, v_3)
$e_3 = (1, 3)$	(v_0, v_3)	(v_1, v_2)
$e_4 = (1, 4)$	(v_0, v_2)	(v_1, v_3)
$e_5 = (1, 5)$	(v_0, v_1)	(v_2, v_3)
$f_0 = (2, 0)$	(v_1, v_2, v_3)	(v_0)
$f_1 = (2, 1)$	(v_0, v_2, v_3)	(v_1)
$f_2 = (2, 2)$	(v_0, v_1, v_3)	(v_2)
$f_3 = (2, 3)$	(v_0, v_1, v_2)	(v_3)
$c_0 = (3, 0)$	(v_0, v_1, v_2, v_3)	$\emptyset$

Numbering of mesh entities on hexahedral cells. The numbering of mesh entities on hexahedral cells is summarized in the table below.

Entity	Incident vertices	Non-incident vertices
$v_0 = (0,0)$	(v_0)	$(v_1, v_2, v_3, v_4, v_5, v_6, v_7)$
$v_1 = (0,1)$	(v_1)	$(v_0, v_2, v_3, v_4, v_5, v_6, v_7)$
$v_2 = (0,2)$	(v_2)	$(v_0, v_1, v_3, v_4, v_5, v_6, v_7)$
$v_3 = (0,3)$	(v_3)	$(v_0, v_1, v_2, v_4, v_5, v_6, v_7)$
$v_4 = (0,4)$	(v_4)	$(v_0, v_1, v_2, v_3, v_5, v_6, v_7)$
$v_5 = (0,5)$	(v_5)	$(v_0, v_1, v_2, v_3, v_4, v_6, v_7)$
$v_6 = (0,6)$	(v_6)	$(v_0, v_1, v_2, v_3, v_4, v_5, v_7)$
$v_7 = (0,7)$	(v_7)	$(v_0, v_1, v_2, v_3, v_4, v_5, v_6)$
$e_0 = (1,0)$	(v_6, v_7)	$(v_0, v_1, v_2, v_3, v_4, v_5)$
$e_1 = (1,1)$	(v_5, v_6)	$(v_0, v_1, v_2, v_3, v_4, v_7)$
$e_2 = (1,2)$	(v_4, v_7)	$(v_0, v_1, v_2, v_3, v_5, v_6)$
$e_3 = (1,3)$	(v_4, v_5)	$(v_0, v_1, v_2, v_3, v_6, v_7)$
$e_4 = (1,4)$	(v_3, v_7)	$(v_0, v_1, v_2, v_4, v_5, v_6)$
$e_5 = (1,5)$	(v_2, v_6)	$(v_0, v_1, v_3, v_4, v_5, v_7)$
$e_6 = (1,6)$	(v_2, v_3)	$(v_0, v_1, v_4, v_5, v_6, v_7)$
$e_7 = (1,7)$	(v_1, v_5)	$(v_0, v_2, v_3, v_4, v_6, v_7)$
$e_8 = (1,8)$	(v_1, v_2)	$(v_0, v_3, v_4, v_5, v_6, v_7)$
$e_9 = (1,9)$	(v_0, v_4)	$(v_1, v_2, v_3, v_5, v_6, v_7)$
$e_{10} = (1,10)$	(v_0, v_3)	$(v_1, v_2, v_4, v_5, v_6, v_7)$
$e_{11} = (1,11)$	(v_0, v_1)	$(v_2, v_3, v_4, v_5, v_6, v_7)$
$f_0 = (2,0)$	(v_4, v_5, v_6, v_7)	(v_0, v_1, v_2, v_3)
$f_1 = (2,1)$	(v_2, v_3, v_6, v_7)	(v_0, v_1, v_4, v_5)
$f_2 = (2,2)$	(v_1, v_2, v_5, v_6)	(v_0, v_3, v_4, v_7)
$f_3 = (2,3)$	(v_0, v_3, v_4, v_7)	(v_1, v_2, v_5, v_6)
$f_4 = (2,4)$	(v_0, v_1, v_4, v_5)	(v_2, v_3, v_6, v_7)
$f_5 = (2,5)$	(v_0, v_1, v_2, v_3)	(v_4, v_5, v_6, v_7)
$c_0 = (3,0)$	$(v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7)$	$\emptyset$

16.6 Discussion

UFC has been used for many applications, including the Poisson equation; convection–diffusion–reaction equations; continuum equations for linear elasticity, hyperelasticity and plasticity; the incompressible Navier–Stokes equations; mixed formulations for the Hodge Laplacian; and many more. The types of finite elements involved include standard continuous Lagrange elements of arbitrary order, discontinuous Galerkin formulations, Brezzi–Douglas–Marini elements, Raviart–Thomas elements, Crouzeix–Raviart elements and Nédélec elements.

The form compilers FFC and SFC described in Chapters 11 and 15 are UFC compliant, both generating efficient UFC code from an abstract problem definition. The assembler in DOLFIN uses the generated UFC code, communicates with the DOLFIN mesh data structure to extract `ufc::mesh` and `ufc::cell` data, and assembles the global tensor into a data structure implemented by one of a number of linear algebra backends supported by DOLFIN, including PETSc, Trilinos (Epetra), uBLAS and MTL4.

One of the main limitations in the current version (2.0) of the UFC interface is the assumption of a homogeneous mesh; that is, only one cell shape is allowed throughout the mesh. Thus, although mesh ordering conventions have been defined for the interval, triangle, tetrahedron, quadrilateral and hexahedron, only one type of shape can be used at any time. Another limitation is that only one fixed finite element space can be chosen for each argument of the form, which excludes p-refinement

(increasing the element order in a subset of the cells). These limitations may be addressed in future versions of the UFC interface.

16.7 Historical notes

UFC was introduced in 2007 when the first version of UFC (1.0) was released. The UFC interface has been used by DOLFIN since the release of DOLFIN 0.7.0 in 2007. The 1.0 release of UFC was followed by version 1.1 in 2008, version 1.2 in 2009, version 1.4 in 2010 and version 2.0 in 2011. The new releases have involved minor corrections to the initial UFC interface but have also introduced some new functionality, like functions for evaluating multiple degrees of freedom (`evaluate_dofs` in addition to `evaluate_dof`) and multiple basis functions (`evaluate_basis_all` in addition to `evaluate_basis`). In contrast to other FEniCS components, few changes are made to the UFC interface in order to maintain a stable interface for both form compilers (FFC and SFC) and assemblers (DOLFIN).

Acknowledgments

The authors would like to thank Johan Hake, Ola Skavhaug, Garth Wells, Kristian Ølgaard and Hans Petter Langtangen for their contributions to UFC.

17 *UFL: a finite element form language*

By Martin Sandve Alnæs

The Unified Form Language – UFL (Alnæs and Logg, 2009) – is a domain specific language for the declaration of finite element discretizations of variational forms and functionals. More precisely, the language defines a flexible user interface for defining finite element spaces and expressions for weak forms in a notation close to mathematical notation.

The FEniCS project provides a framework for building applications for solving partial differential equations (PDEs). UFL is one of the core components of this framework. It defines the language you *express* your PDEs in. It is the input language and front-end of the form compilers FFC and SFC, which are covered in Chapter 11 and Chapter 15. The UFL implementation also provides algorithms that the form compilers can use to simplify the compilation process. The output from these form compilers is C++ (Stroustrup, 1997) code that conforms to the UFC specification, which is explained in Chapter 16. This code can be used with the C++/Python library DOLFIN, which is covered in Chapter 10, to efficiently assemble linear systems and compute solutions to PDEs. Note that this chapter does not cover how to actually solve equations defined in UFL. See Chapter 1 for a tutorial on how to use the complete FEniCS framework to solve equations.

This chapter is intended both for the FEniCS user who wants to learn how to express her equations, and for other FEniCS developers and technical users who want to know how UFL works on the inside. Therefore, the sections of this chapter are organized with an increasing amount of technical details. Sections 17.1–17.5 give an overview of the language as seen by the end-user and is intended for all audiences. Sections 17.6–17.9 explain the design of the implementation and dive into some implementation details. Many details of the language have to be omitted in a text such as this, and we refer to the UFL manual (Alnæs and Logg, 2009) for a more thorough description. Note that this chapter refers to UFL version 1.0.0, and both the user interface and the implementation may change in future versions.

Starting with a brief overview, we mention the main design goals for UFL and show an example implementation of a non-trivial PDE in Section 17.1. Next, we look at how to define finite element spaces in Section 17.2, followed by the overall structure of forms and their declaration in Section 17.3. The main part of the language is concerned with defining expressions from a set of data types and operators, which are discussed in Section 17.4. Operators applying to entire forms are the topic of Section 17.5.

The technical part of the chapter begins with Section 17.6 which discusses the representation of expressions. Building on the notation and data structures defined there, how to compute derivatives is discussed in Section 17.7. Some central internal algorithms and key issues in their implementation are discussed in Section 17.8. Implementation details, some of which are specific to the programming language Python (van Rossum et al.), are the topic of Section 17.9. Finally, Section 17.10 discusses future prospects of the UFL project.

17.0.1 Related work

The combination of domain specific languages and symbolic computing with finite element methods has been pursued from other angles in several other projects. Sundance (Long, 2003, 2004b,a) implements a symbolic engine directly in C++ to define variational forms, and has support for automatic differentiation. The Life (Prud'homme, 2006b,a) project uses a domain specific language embedded in C++, based on expression template techniques to specify variational forms. SfePy (Cimrman et al., 2008) uses SymPy as a symbolic engine, extending it with finite element methods. GetDP (Dular and Geuzaine, 2005) is another project using a domain specific language for variational forms. The Mathematica package AceGen (Korelc, 1997, 2002) uses the symbolic capabilities of Mathematica to generate efficient code for finite element methods. All these packages have in common a focus on high level descriptions of partial differential equations to achieve higher human efficiency in the development of simulation software.

UFL almost resembles a library for symbolic computing, but its scope, goals and priorities are different from generic symbolic computing projects such as GiNaC (Bauer et al., 2002), Swiginac (Skavhaug and Čertík, 2009) and SymPy (Čertík et al., 2009). Intended as a domain specific language and form compiler frontend, UFL is not suitable for large scale symbolic computing.

17.1 Overview

17.1.1 Design goals

UFL is a unification, refinement and reimplementation of the form languages used in previous versions of FFC and SFC. The development of this language has been motivated by several factors, the most important being:

- A richer form language, especially for expressing nonlinear PDEs.
- Automatic differentiation of expressions and forms.
- Improving the performance of the form compiler technology to handle more complicated equations efficiently.

UFL fulfills all these requirements, and by this it represents a major step forward in the capabilities of the FEniCS project.

Tensor algebra and index notation support is modeled after the FFC form language and generalized further. Several nonlinear operators and functions which only SFC supported before have been included in the language. Differentiation of expressions and forms has become an integrated part of the language, and is much easier to use than the way these features were implemented in SFC before. In summary, UFL combines the best of FFC and SFC in one unified form language and adds additional capabilities.

The efficiency of code generated by the new generation of form compilers based on UFL has been verified to match previous form compiler benchmarks (Alnæs and Mardal, 2010; Ølgaard and Wells, 2010). The form compilation process is now fast enough to blend into the regular application build process. Complicated forms that previously required too much memory to compile, or took tens of minutes or even hours to compile, now compiles in seconds with both SFC and FFC.

17.1.2 Motivational example

One major motivating example during the initial development of UFL has been the equations for elasticity with large deformations. In particular, models of biological tissue use complicated

hyperelastic constitutive laws with anisotropies and strong nonlinearities. To implement these equations with FEniCS, all three design goals listed above had to be addressed. Below, one version of the hyperelasticity equations and their corresponding UFL implementation is shown. Keep in mind that this is only intended as an illustration of the close correspondence between the form language and the natural formulation of the equations. The meaning of these equations is not necessary for the reader to understand. Chapter 27 covers nonlinear elasticity in more detail. Note that many other examples are distributed together with UFL.

In the formulation of the hyperelasticity equations presented here, the unknown function is the displacement vector field u. The material coefficients c_1 and c_2 are scalar constants. The second Piola-Kirchoff stress tensor S is computed from the strain energy function $W(C)$. W defines the constitutive law, here a simple Mooney-Rivlin law. The equations relating the displacement and stresses read:

$$\begin{aligned} F &= I + \operatorname{grad} u, \\ C &= F^\top F, \\ I_C &= \operatorname{tr}(C), \\ II_C &= \frac{1}{2}(\operatorname{tr}(C)^2 - \operatorname{tr}(CC)), \\ W &= c_1(I_C - 3) + c_2(II_C - 3), \\ S &= 2\frac{\partial W}{\partial C}. \end{aligned} \tag{17.1}$$

For simplicity in this example, we ignore external body and boundary forces and assume a quasi-stationary situation, leading to the following mechanics problem. Find u such that

$$\operatorname{div}(FS) = 0, \quad \text{in d}x, \tag{17.2}$$

$$u = u_0, \quad \text{on d}s. \tag{17.3}$$

The finite element method is presented in Chapter 2, so we will only very briefly cover the steps we take here. First we multiply Equation (17.2) with a test function $\phi \in V$, then integrate over the domain Ω, and integrate by parts. The nonlinear variational problem then reads: find $u \in V$ such that

$$L(u;\phi) = \int_\Omega FS : \operatorname{grad}\phi \,\mathrm{d}x = 0 \quad \forall\, \phi \in V. \tag{17.4}$$

Here we have omitted the coefficients c_1 and c_2 for brevity. Approximating the displacement field as $u \approx u_h = \sum_k u_k \psi_k$, where $\psi_k \in V_h \approx V$ are trial functions, and using Newtons's method to solve the nonlinear equations, we end up with a system of equations to solve

$$\sum_{k=1}^{|V_h|} \frac{\partial L(u_h;\phi)}{\partial u_k} \Delta u_k = -L(u_h;\phi) \quad \forall\, \phi \in V_h. \tag{17.5}$$

A bilinear form $a(u;\psi,\phi)$ corresponding to the left-hand side of Equation (17.5) can be computed automatically by UFL, such that

$$a(u_h;\psi_k,\phi) = \frac{\partial L(u_h;\phi)}{\partial u_k} \quad k = 1,\ldots,|V_h|. \tag{17.6}$$

Figure 17.1 shows an implementation of equations (17.1), (17.4) and (17.6) in UFL. Notice the

UFL code

```
# Finite element spaces
cell = tetrahedron
element = VectorElement("Lagrange", cell, 1)

# Form arguments
phi0 = TestFunction(element)
phi1 = TrialFunction(element)
u = Coefficient(element)
c1 = Constant(cell)
c2 = Constant(cell)

# Deformation gradient Fij = dXi/dxj
I = Identity(cell.d)
F = I + grad(u)

# Right Cauchy-Green strain tensor C with invariants
C = variable(F.T*F)
I_C = tr(C)
II_C = (I_C**2 - tr(C*C))/2

# Mooney-Rivlin constitutive law
W = c1*(I_C-3) + c2*(II_C-3)

# Second Piola-Kirchoff stress tensor
S = 2*diff(W, C)

# Weak forms
L = inner(F*S, grad(phi0))*dx
a = derivative(L, u, phi1)
```

Figure 17.1: UFL implementation of hyperelasticity equations with a Mooney-Rivlin material law.

close relation between the mathematical notation and the UFL source code. In particular, note the automated differentiation of both the constitutive law and the residual equation. The operator `diff` can be applied to expressions to differentiate w.r.t designated variables such as `C` here, while the operator `derivative` can be applied to entire forms to differentiate w.r.t. each coefficient of a discrete function such as `u`. The combination of these features allows a new material law to be implemented by simply changing W, the rest is automatic. In the following sections, the notation, definitions and operators used in this implementation will be explained.

17.2 Defining finite element spaces

A polygonal cell is defined in UFL by a basic shape, and is declared by

UFL code

```
cell = Cell(shapestring)
```

UFL defines a set of valid polygonal cell shapes: “interval”, “triangle”, “tetrahedron”, “quadrilateral”, and “hexahedron”. `Cell` objects of all shapes are predefined and can be used instead by writing

UFL code

```
cell = tetrahedron
```

In the rest of this chapter, a variable name `cell` will be used where any cell is a valid argument, to make the examples dimension independent wherever possible.

UFL defines syntax for *declaring* finite element spaces, but does not know anything about the actual polynomial basis or degrees of freedom. The polynomial basis is selected implicitly by choosing among predefined basic element families and providing a polynomial degree, but UFL only assumes that there *exists* a basis with a fixed ordering for each finite element space V_h; that is,

$$V_h = \text{span}\left\{\phi_j\right\}_{j=1}^{n}. \tag{17.7}$$

Basic scalar elements can be combined to form vector elements or tensor elements, and elements can easily be combined in arbitrary mixed element hierarchies.

The set of predefined[1] element family names in UFL includes "Lagrange" (short name "CG"), representing scalar Lagrange finite elements (continuous piecewise polynomial functions), "Discontinuous Lagrange" (short name "DG"), representing scalar discontinuous Lagrange finite elements (discontinuous piecewise polynomial functions), and a range of other families that can be found in the manual. Each family name has an associated short name for convenience. To print all valid families to screen from Python, call `show_elements()`.

The syntax for declaring elements is best explained with some examples.

UFL code

```
cell = tetrahedron

P = FiniteElement("Lagrange", cell, 1)
V = VectorElement("Lagrange", cell, 2)
T = TensorElement("DG", cell, 0, symmetry=True)

TH = V*P
ME = MixedElement(T, V, P)
```

In the first line a polygonal cell is selected from the set of predefined cells. Then a scalar linear Lagrange element `P` is declared, as well as a quadratic vector Lagrange element `V`. Next a symmetric rank 2 tensor element `T` is defined, which is also piecewise constant on each cell. The code proceeds to declare a mixed element `TH`, which combines the quadratic vector element `V` and the linear scalar element `P`. This element is known as the Taylor-Hood element. Finally another mixed element with three subelements is declared. Note that writing `T*V*P` would not result in a mixed element with three direct subelements, but rather `MixedElement(MixedElement(T, V), P)`.

17.3 *Defining forms*

Consider Poisson's equation with two different boundary conditions on $\partial\Omega_0$ and $\partial\Omega_1$,

$$a(w;u,v) = \int_\Omega w \,\text{grad}\, u \cdot \text{grad}\, v \,\mathrm{d}x, \tag{17.8}$$

$$L(f,g,h;v) = \int_\Omega fv \,\mathrm{d}x + \int_{\partial\Omega_0} g^2 v \,\mathrm{d}s + \int_{\partial\Omega_1} hv \,\mathrm{d}s. \tag{17.9}$$

These forms can be expressed in UFL as

UFL code

```
a = w*dot(grad(u), grad(v))*dx
L = f*v*dx + g**2*v*ds(0) + h*v*ds(1)
```

[1] Form compilers can register additional element families.

where multiplication by the measures `dx`, `ds(0)` and `ds(1)` represent the integrals $\int_{\Omega_0}(\cdot)\,\mathrm{d}x$, $\int_{\partial\Omega_0}(\cdot)\,\mathrm{d}s$, and $\int_{\partial\Omega_1}(\cdot)\,\mathrm{d}s$ respectively.

Forms expressed in UFL are intended for finite element discretization followed by compilation to efficient code for computing the element tensor. Considering the above example, the bilinear form a with one coefficient function w is assumed to be evaluated at a later point with a range of basis functions and the coefficient function fixed, that is

$$V_h^1 = \operatorname{span}\left\{\phi_k^1\right\}, \quad V_h^2 = \operatorname{span}\left\{\phi_k^2\right\}, \quad V_h^3 = \operatorname{span}\left\{\phi_k^3\right\}, \tag{17.10}$$

$$w = \sum_{k=1}^{|V_h^3|} w_k \phi_k^3, \quad \{w_k\} \text{ given}, \tag{17.11}$$

$$A_{ij} = a(w; \phi_i^1, \phi_j^2), \quad i = 1, \ldots, |V_h^1|, \quad j = 1, \ldots, |V_h^2|. \tag{17.12}$$

In general, UFL is designed to express forms of the following generalized form:

$$a(w^1, \ldots, w^n; \phi^1, \ldots, \phi^r) = \sum_{k=1}^{n_c} \int_{\Omega_k} I_k^c \,\mathrm{d}x + \sum_{k=1}^{n_e} \int_{\partial\Omega_k} I_k^e \,\mathrm{d}s + \sum_{k=1}^{n_i} \int_{\Gamma_k} I_k^i \,\mathrm{d}S. \tag{17.13}$$

Most of this chapter deals with ways to define the integrand expressions I_k^c, I_k^e and I_k^i. The rest of the notation will be explained below.

The form arguments are divided in two groups, the basis functions $\phi^1, \ldots, \phi^r$ and the coefficient functions $w^1, \ldots, w^n$. All $\{\phi^k\}$ and $\{w^k\}$ are functions in some discrete function space with a basis. Note that the actual basis functions $\{\phi_j^k\}$ and the coefficients $\{w_k\}$ are never known to UFL, but we assume that the ordering of the basis for each finite element space is fixed. A fixed ordering only matters when differentiating forms, explained in Section 17.7.

Each term of a valid form expression must be a scalar-valued expression integrated exactly once, and they must be linear in $\{\phi^k\}$. Any term may have nonlinear dependencies on coefficient functions. A form with one or two basis function arguments ($r = 1, 2$) is called a linear or bilinear form respectively, ignoring its dependency on coefficient functions. These will be assembled to vectors and matrices when used in an application. A form depending only on coefficient functions ($r = 0$) is called a functional, since it will be assembled to a real number. Multilinear forms where $r > 2$ are supported but not as commonly used.

The entire domain is denoted Ω, the external boundary is denoted $\partial\Omega$, while the set of interior facets of the triangulation is denoted Γ. Subdomains are marked with a suffix, e.g., $\Omega_k \subset \Omega$. As mentioned above, integration is expressed by multiplication with a measure, and UFL defines the measures `dx`, `ds` and `dS`. In summary, there are three kinds of integrals with corresponding UFL representations

- $\int_{\Omega_k}(\cdot)\,\mathrm{d}x \leftrightarrow$ `(·)*dx(k)`, called a *cell integral*,
- $\int_{\partial\Omega_k}(\cdot)\,\mathrm{d}s \leftrightarrow$ `(·)*ds(k)`, called an *exterior facet integral*,
- $\int_{\Gamma_k}(\cdot)\,\mathrm{d}S \leftrightarrow$ `(·)*dS(k)`, called an *interior facet integral*,

Defining a different quadrature order for each term in a form can be achieved by attaching metadata to measure objects, e.g.,

UFL code

```
dx02 = dx(0, { "integration_order": 2 })
dx14 = dx(1, { "integration_order": 4 })
```

```
dx12 = dx(1, { "integration_order": 2 })
L = f*v*dx02 + g*v*dx14 + h*v*dx12
```

Metadata can also be used to override other form compiler specific options separately for each term. For more details on this feature see the manuals of UFL and the form compilers.

17.4 Defining expressions

Most of UFL deals with how to declare expressions such as the integrand expressions in Equation 17.13. The most basic expressions are terminal values, which do not depend on other expressions. Other expressions are called operators, which are discussed in sections 17.4.2–17.4.5.

Terminal value types in UFL include form arguments (which is the topic of Section 17.4.1), geometric quantities, and literal constants. Among the literal constants are scalar integer and floating point values, as well as the d by d identity matrix `I = Identity(d)`. To get unit vectors, simply use rows or columns of the identity matrix, e.g., `e0 = I[0,:]`. Similarly, `I[i,j]` represents the Kronecker delta function δ_{ij} (see Section 17.4.2 for details on index notation). Available geometric values are the spatial coordinates $x \leftrightarrow$ `cell.x` and the facet normal $n \leftrightarrow$ `cell.n`. The geometric dimension is available as `cell.d`.

17.4.1 Form arguments

Basis functions and coefficient functions are represented by `Argument` and `Coefficient` respectively. The ordering of the arguments to a form is decided by the order in which the form arguments were declared in the UFL code. Each basis function argument represents any function in the basis of its finite element space

$$\phi^j \in \{\phi_k^j\}, \quad V_h^j = \text{span}\left\{\phi_k^j\right\}. \tag{17.14}$$

with the intention that the form is later evaluated for all ϕ_k such as in Equation (17.12). Each coefficient function w represents a discrete function in some finite element space V_h; it is usually a sum of basis functions $\phi_k \in V_h$ with coefficients w_k

$$w = \sum_{k=1}^{|V_h|} w_k \phi_k. \tag{17.15}$$

The exception is coefficient functions that can only be evaluated point-wise, which are declared with a finite element with family "Quadrature". Basis functions are declared for an arbitrary element as in the following manner:

UFL code

```
phi = Argument(element)
v = TestFunction(element)
u = TrialFunction(element)
```

By using `TestFunction` and `TrialFunction` in declarations instead of `Argument` you can ignore their relative ordering. The only time `Argument` is needed is for forms of arity $r > 2$.

Coefficient functions are declared similarly for an arbitrary element, and shorthand notation exists for declaring constant coefficients:

UFL code

```
w = Coefficient(element)
```

```
c = Constant(cell)
v = VectorConstant(cell)
M = TensorConstant(cell)
```

If a form argument u in a mixed finite element space $V_h = V_h^0 \times V_h^1$ is desired, but the form is more easily expressed using subfunctions $u_0 \in V_h^0$ and $u_1 \in V_h^1$, you can split the mixed function or basis function into its subfunctions in a generic way using `split`:

UFL code

```
V = V0*V1
u = Coefficient(V)
u0, u1 = split(u)
```

The `split` function can handle arbitrary mixed elements. Alternatively, a handy shorthand notation for argument declaration followed by `split` is

UFL code

```
v0, v1 = TestFunctions(V)
u0, u1 = TrialFunctions(V)
f0, f1 = Coefficients(V)
```

17.4.2 Index notation

UFL allows working with tensor expressions of arbitrary rank, using both tensor algebra and index notation. A basic familiarity with tensor algebra and index notation is assumed. The focus here is on how index notation is expressed in UFL.

Assuming a standard orthonormal Euclidean basis $\langle e_k \rangle_{k=1}^d$ for $\mathbb{R}^d$, a vector can be expressed with its scalar components in this basis. Tensors of rank two can be expressed using their scalar components in a dyadic basis $\{e_i \otimes e_j\}_{i,j=1}^d$. Arbitrary rank tensors can be expressed the same way, as illustrated here.

$$v = \sum_{k=1}^{d} v_k e_k, \tag{17.16}$$

$$A = \sum_{i=1}^{d} \sum_{j=1}^{d} A_{ij} e_i \otimes e_j, \tag{17.17}$$

$$\mathcal{C} = \sum_{i=1}^{d} \sum_{j=1}^{d} \sum_{k} C_{ijk} e_i \otimes e_j \otimes e_k. \tag{17.18}$$

Here, v, A and $\mathcal{C}$ are rank 1, 2 and 3 tensors respectively. Indices are called *free* if they have no assigned value, such as i in v_i, and *fixed* if they have a fixed value such as 1 in v_1. An expression with free indices represents any expression you can get by assigning fixed values to the indices. The expression A_{ij} is scalar valued, and represents any component (i, j) of the tensor A in the Euclidean basis. When working on paper, it is easy to switch between tensor notation (A) and index notation (A_{ij}) with the knowledge that the tensor and its components are different representations of the same physical quantity. In a programming language, we must express the operations mapping from tensor to scalar components and back explicitly. Mapping from a tensor to its components, for a rank 2 tensor defined as

$$A_{ij} = A : (e_i \otimes e_j) \tag{17.19}$$

is accomplished using indexing with the notation `A[i,j]`. Defining a tensor A from component values A_{ij} is defined as

$$A = A_{ij} e_i \otimes e_j, \tag{17.20}$$

and is accomplished using the function `as_tensor(Aij, (i,j))`. To illustrate, consider the outer product of two vectors $A = u \otimes v = u_i v_j e_i \otimes e_j$, and the corresponding scalar components A_{ij}. One way to implement this is

UFL code

```
A = outer(u, v)
Aij = A[i, j]
```

Alternatively, the components of A can be expressed directly using index notation, such as $A_{ij} = u_i v_j$. A_{ij} can then be mapped to A in the following manner:

UFL code

```
Aij = v[j]*u[i]
A = as_tensor(Aij, (i, j))
```

These two pairs of lines are mathematically equivalent, and the result of either pair is that the variable `A` represents the tensor A and the variable `Aij` represents the tensor A_{ij}. Note that free indices have no ordering, so their order of appearance in the expression `v[j]*u[i]` is insignificant. Instead of `as_tensor`, the specialized functions `as_vector` and `as_matrix` can be used. Although a rank two tensor was used for the examples above, the mappings generalize to arbitrary rank tensors.

When indexing expressions, fixed indices can also be used such as in `A[0,1]` which represents a single scalar component. Fixed indices can also be mixed with free indices such as in `A[0,i]`. In addition, slices can be used in place of an index. An example of using slices is `A[0,:]` which is a vector expression that represents row 0 of `A`. To create new indices, you can either make a single one or make several at once:

UFL code

```
i = Index()
j, k, l = indices(3)
```

A set of indices `i`, `j`, `k`, `l` and `p`, `q`, `r`, `s` are predefined, and these should suffice for most applications.

If your components are not represented as an expression with free indices, but as separate unrelated scalar expressions, you can build a tensor from them using `as_tensor` and its peers. As an example, lets define a 2D rotation matrix and rotate a vector expression by $\frac{\pi}{2}$:

UFL code

```
th = pi/2
A = as_matrix([[ cos(th), -sin(th)],
               [ sin(th),  cos(th)]])
u = A*v
```

When indices are repeated in a term, summation over those indices is implied in accordance with the Einstein convention. In particular, indices can be repeated when indexing a tensor of rank two or higher (`A[i,i]`), when differentiating an expression with a free index (`v[i].dx(i)`), or when multiplying two expressions with shared free indices (`u[i]*v[i]`).

$$A_{ii} \equiv \sum_i A_{ii}, \qquad v_i u_i \equiv \sum_i v_i u_i, \qquad v_{i,i} \equiv \sum_i v_{i,i}. \tag{17.21}$$

An expression `Aij = A[i,j]` is represented internally using the `Indexed` class. `Aij` will reference `A`, keeping the representation of the original tensor expression `A` unchanged. Implicit summation is represented explicitly in the expression tree using the class `IndexSum`. Many algorithms become easier to implement with this explicit representation, since e.g. a `Product` instance can never implicitly represent a sum. More details on representation classes are found in Section 17.6.

17.4.3 Algebraic operators and functions

UFL defines a comprehensive set of operators that can be used for composing expressions. The elementary algebraic operators `+`, `-`, `*`, `/` can be used between most UFL expressions with a few limitations. Division requires a scalar expression with no free indices in the denominator. The operands to a sum must have the same shape and set of free indices.

The multiplication operator $*$ is valid between two scalars, a scalar and any tensor, a matrix and a vector, and two matrices. Other products could have been defined, but for clarity we use tensor algebra operators and index notation for those rare cases. A product of two expressions with shared free indices implies summation over those indices, see Section 17.4.2 for more about index notation.

Three often used operators are `dot(a, b)`, `inner(a, b)`, and `outer(a, b)`. The dot product of two tensors of arbitrary rank is the sum over the last index of the first tensor and the first index of the second tensor. Some examples are

$$v \cdot u = v_i u_i, \tag{17.22}$$
$$A \cdot u = A_{ij} u_j e_i, \tag{17.23}$$
$$A \cdot B = A_{ik} B_{kj} e_i e_j, \tag{17.24}$$
$$\mathcal{C} \cdot A = C_{ijk} A_{kl} e_i e_j e_l. \tag{17.25}$$

The inner product is the sum over all indices, for example

$$v : u = v_i u_i, \tag{17.26}$$
$$A : B = A_{ij} B_{ij}, \tag{17.27}$$
$$\mathcal{C} : \mathcal{D} = C_{ijkl} D_{ijkl}. \tag{17.28}$$

Some examples of the outer product are

$$v \otimes u = v_i u_j e_i e_j, \tag{17.29}$$
$$A \otimes u = A_{ij} u_k e_i e_j e_k, \tag{17.30}$$
$$A \otimes B = A_{ij} B_{kl} e_i e_j e_k e_l \tag{17.31}$$

Other common tensor algebra operators are `cross(u,v)`, `transpose(A)` (or `A.T`), `tr(A)`, `det(A)`, `inv(A)`, `cofac(A)`, `dev(A)`, `skew(A)`, and `sym(A)`. Most of these tensor algebra operators expect tensors without free indices. The detailed definitions of these operators are found in the manual.

A set of common elementary functions operating on scalar expressions without free indices are included, in particular `abs(f)`, `pow(f, g)`, `sqrt(f)`, `exp(f)`, `ln(f)`, `cos(f)`, `sin(f)`, `tan(f)`, `acos(f)`, `asin(f)`, `atan(f)`, and `sign(f)`. Any operator taking scalar arguments can be applied element-wise to tensors using e.g. `elem_op(sin, A)`.

17.4.4 Differential operators

UFL implements derivatives w.r.t. three different kinds of variables. The most used kind is spatial

derivatives. Expressions can also be differentiated w.r.t. arbitrary user defined variables. And the final kind of derivatives are derivatives of a form or functional w.r.t. the coefficients of a discrete function; that is, a `Coefficient` or `Constant`. Form derivatives are explained in Section 17.5.1.

Note that derivatives are not computed immediately when declared. A discussion of how derivatives are computed is found in Section 17.7.

Spatial derivatives Basic spatial derivatives $\frac{\partial f}{\partial x_i}$ can be expressed in two equivalent ways:

UFL code

```
df = Dx(f, i)
df = f.dx(i)
```

Here, `df` represents the derivative of `f` in the spatial direction x_i. The index `i` can either be an integer, representing differentiation in one fixed spatial direction x_i, or an `Index`, representing differentiation in the direction of a free index. The notation `f.dx(i)` is intended to mirror the index notation $f_{,i}$, which is shorthand for $\frac{\partial f}{\partial x_i}$. Repeated indices imply summation, such that the divergence of a vector valued expression `v` can be written $v_{i,i}$, or `v[i].dx(i)`.

Several common compound spatial derivative operators are defined, namely the gradient, divergence, and curl (rot) operators. These operators are named `grad`, `div`, `nabla_grad`, `nabla_div`, `curl` and `rot` (rot is a synonym for curl). Be aware that there are two common ways to define the gradient and divergence, and UFL supports both.

Let s be a scalar expression, v be a vector expression, and M be a tensor expression of rank r. In UFL, the operator `grad` is then defined explicitly as

$$(\text{grad}(s))_i = s_{,i}, \tag{17.32}$$
$$(\text{grad}(v))_{ij} = v_{i,j}, \tag{17.33}$$
$$(\text{grad}(M))_{i_1 \ldots i_r k} = M_{i_1 \ldots i_r, k}, \tag{17.34}$$

and the operator `div` is correspondingly defined as

$$\text{div}(v) = v_{i,i}, \tag{17.35}$$
$$(\text{div}(M))_{i_1 \ldots i_{r-1}} = M_{i_1 \ldots i_r, i_r}. \tag{17.36}$$

In contrast, the `nabla_*` operators are defined in terms of the ∇ operator

$$\nabla \equiv e_k \frac{\partial}{\partial x_k}. \tag{17.37}$$

The operator `nabla_grad` is the outer product of ∇ with its operand:

$$(\nabla s)_i = s_{,i}, \tag{17.38}$$
$$(\nabla v)_{ij} = v_{j,i}, \tag{17.39}$$
$$(\nabla M)_{k,i_1 \ldots i_r} = M_{i_1 \ldots i_r, k}. \tag{17.40}$$

Similarly, the operator `nabla_div` is the dot product of ∇ with its operand:

$$\nabla \cdot v = v_{i,i}, \tag{17.41}$$
$$(\nabla \cdot M)_{i_2 \ldots i_r} = M_{i_1 \ldots i_r, i_1}. \tag{17.42}$$

Thinking in terms of value shape, the `grad` operator appends an axis to the end of the shape of its operand, while the `nabla_grad` operator prepends an axis. For gradients of scalars, the result is the same. Correspondingly, the `div` operator sums over the last index of its operand, while the `nabla_div` operator sums over the first index of its operand. For the divergence of vectors, the result is the same.

For the operators `curl` and `rot` there is no difference between the two traditions. For 3D vector expressions, the curl can be defined in terms of the nabla operator and the cross product:

$$\text{curl}(v) \equiv \nabla \times v = e_0(v_{2,1} - v_{1,2}) - e_1(v_{2,0} - v_{0,2}) + e_2(v_{1,0} - v_{0,1}) \tag{17.43}$$

For 2D vector and scalar expressions the definitions are:

$$\text{curl}(v) \equiv v_{1,0} - v_{0,1}, \tag{17.44}$$
$$\text{curl}(f) \equiv f_{,1}e_0 - f_{,0}e_1. \tag{17.45}$$

User defined variables The second kind of differentiation variables are user-defined variables, which can represent arbitrary expressions. Automating derivatives w.r.t. arbitrary quantities is useful for several tasks, from differentiation of material laws to computing sensitivities. An arbitrary expression g can be assigned to a variable v. An expression f defined as a function of v can be differentiated f w.r.t. v:

$$v = g, \tag{17.46}$$
$$f = f(v), \tag{17.47}$$
$$h(v) = \frac{\partial f(v)}{\partial v}. \tag{17.48}$$

Setting $g = sin(x_0)$ and $f = e^{v^2}$, gives $h = 2ve^{v^2} = 2\sin(x_0)e^{\sin^2(x_0)}$, which can be implemented as follows:

UFL code

```
g = sin(cell.x[0])
v = variable(g)
f = exp(v**2)
h = diff(f, v)
```

Try running this code in a Python session and print the expressions. The result is

Python code

```
>> print v
var0(sin((x)[0]))
>> print h
d/d[var0(sin((x)[0]))] (exp((var0(sin((x)[0]))) ** 2))
```

Note that the variable has a label "`var0`", and that `h` still represents the abstract derivative. Section 17.7 explains how derivatives are computed.

17.4.5 Other operators

A few operators are provided for the implementation of discontinuous Galerkin methods. The basic concept is restricting an expression to the positive or negative side of an interior facet, which is expressed simply as `v("+")` or `v("-")` respectively. On top of this, the operators `avg` and `jump` are

implemented, defined as

$$\text{avg}(v) = \frac{1}{2}(v^+ + v^-), \tag{17.49}$$

$$\text{jump}(v) = v^+ - v^-. \tag{17.50}$$

These operators can only be used when integrating over the interior facets (`*dS`).

The only control flow construct included in UFL is conditional expressions. A conditional expression takes on one of two values depending on the result of a boolean logic expression. The syntax for this is

UFL code

```
f = conditional(condition, true_value, false_value)
```

which is interpreted as

$$f = \begin{cases} t, & \text{if condition is true,} \\ f, & \text{otherwise.} \end{cases} \tag{17.51}$$

The condition can be one of

- `lt(a, b)` $\leftrightarrow (a < b)$
- `le(a, b)` $\leftrightarrow (a \leq b)$
- `eq(a, b)` $\leftrightarrow (a = b)$
- `And(P, Q)` $\leftrightarrow (P \wedge Q)$
- `Not(P)` $\leftrightarrow (\neg P)$
- `gt(a, b)` $\leftrightarrow (a > b)$
- `ge(a, b)` $\leftrightarrow (a \geq b)$
- `ne(a, b)` $\leftrightarrow (a \neq b)$
- `Or(P, Q)` $\leftrightarrow (P \vee Q)$

17.5 Form operators

Once you have defined some forms, there are several ways to compute related forms from them. While operators in the previous section are used to define expressions, the operators discussed in this section are applied to forms, producing new forms. Form operators can both make form definitions more compact and reduce the chances of bugs since changes in the original form will propagate to forms computed from it automatically. These form operators can be combined arbitrarily; given a semi-linear form only a few lines are needed to compute the action of the adjoint of the Jacobi. Since these computations are done prior to processing by the form compilers, there is no overhead at run-time.

17.5.1 Differentiating forms

The form operator `derivative` declares the derivative of a form w.r.t. coefficients of a discrete function (`Coefficient`). This functionality can be used for example to linearize your nonlinear residual equation (linear form) automatically for use with the Newton-Raphson method. It can also be applied multiple times, which is useful to derive a linear system from a convex functional, in order to find the function that minimizes the functional. For non-trivial equations such expressions can be tedious to calculate by hand. Other areas in which this feature can be useful include optimal control and inverse methods, as well as sensitivity analysis.

In its simplest form, the declaration of the derivative of a form `L` w.r.t. the coefficients of a function `w` reads

UFL code

```
a = derivative(L, w, u)
```

The form `a` depends on an additional basis function argument `u`, which must be in the same finite element space as the function `w`. If the last argument is omitted, a new basis function argument is created.

Let us step through an example of how to apply `derivative` twice to a functional to derive a linear system. In the following, V_h is a finite element space with some basis , w is a function in V_h, and $f = f(w)$ is a functional we want to minimize. Derived from $f(w)$ is a linear form $F(w;v)$, and a bilinear form $J(w;u,v)$.

$$V_h = \text{span}\left\{\phi_k\right\}, \tag{17.52}$$

$$w(x) = \sum_{k=1}^{|V_h|} w_k \phi_k(x), \tag{17.53}$$

$$f : V_h \to \mathbb{R}, \tag{17.54}$$

$$F(w;\phi_i) = \frac{\partial f(w)}{\partial w_i}, \quad i = 1, \ldots, |V_h|, \tag{17.55}$$

$$J(w;\phi_j,\phi) = \frac{\partial F(w;\phi)}{\partial w_j}, \quad j = 1, \ldots, |V_h|, \quad \phi \in V_h. \tag{17.56}$$

For a concrete functional $f(w) = \int_\Omega \frac{1}{2} w^2 \, \mathrm{d}x$, we can implement this as

UFL code

```
v = TestFunction(element)
u = TrialFunction(element)
w = Coefficient(element)
f = 0.5*w**2*dx
F = derivative(f, w, v)
J = derivative(F, w, u)
```

This code declares two forms `F` and `J`. The linear form `F` represents the standard load vector `w*v*dx` and the bilinear form `J` represents the mass matrix `u*v*dx`.

Derivatives can also be defined w.r.t. coefficients of a function in a mixed finite element space. Consider the Harmonic map equations derived from the functional

$$f(x,\lambda) = \int_\Omega \text{grad}\, x : \text{grad}\, x + \lambda x \cdot x \, \mathrm{d}x, \tag{17.57}$$

where x is a function in a vector finite element space V_h^d and λ is a function in a scalar finite element space V_h. The linear and bilinear forms derived from the functional in Equation 17.57 have basis function arguments in the mixed space $V_h^d \times V_h$. The implementation of these forms with automatic linearization reads

UFL code

```
Vx = VectorElement("Lagrange", triangle, 1)
Vy = FiniteElement("Lagrange", triangle, 1)
u = Coefficient(Vx*Vy)
x, y = split(u)
```

```
f = inner(grad(x), grad(x))*dx + y*dot(x,x)*dx
F = derivative(f, u)
J = derivative(F, u)
```

Note that the functional is expressed in terms of the subfunctions `x` and `y`, while the argument to `derivative` must be the single mixed function `u`. In this example the basis function arguments to `derivative` are omitted and thus provided automatically in the right function spaces.

Note that in computing derivatives of forms, we have assumed that

$$\frac{\partial}{\partial w_k}\int_\Omega I\,\mathrm{d}x = \int_\Omega \frac{\partial}{\partial w_k} I\,\mathrm{d}x, \tag{17.58}$$

or in particular that the domain Ω is independent of w. Also, any coefficients other than w are assumed independent of w. Furthermore, note that there is no restriction on the choice of element in this framework, in particular arbitrary mixed elements are supported.

17.5.2 Adjoint

Another form operator is the adjoint a^* of a bilinear form a, defined as $a^*(v,u) = a(u,v)$, which is equivalent to taking the transpose of the assembled sparse matrix. In UFL this is implemented simply by swapping the order of the test and trial functions, and can be written using the `adjoint` form operator. (Note that this is not the most generic definition of the adjoint of an operator). An example of its use on an anisotropic diffusion term looks like

UFL code

```
V = VectorElement("Lagrange", cell, 1)
T = TensorElement("Lagrange", cell, 1)
u = TrialFunction(V)
v = TestFunction(V)
M = Coefficient(T)
a = M[i,j]*u[k].dx(j)*v[k].dx(i)*dx
astar = adjoint(a)
```

which corresponds to (with $u \in U$ and $v \in V$)

$$a(M;u,v) = \int_\Omega M_{ij}u_{k,j}v_{k,i}\,\mathrm{d}x, \tag{17.59}$$

$$a^*(M;v,u) = \int_\Omega M_{ij}u_{k,j}v_{k,i}\,\mathrm{d}x = a(M;u,v). \tag{17.60}$$

This automatic transformation is particularly useful if we need the adjoint of nonsymmetric bilinear forms computed using `derivative`, since the explicit expressions for a are not at hand. Several of the form operators below are most useful when used in conjunction with `derivative`.

17.5.3 Replacing functions

Evaluating a form with new definitions of form arguments can be done by replacing terminal objects with other values. Lets say you have defined a form `L` that depends on some functions `f` and `g`. You can then specialize the form by replacing these functions with other functions or fixed values, such as

$$L(f,g;v) = \int_\Omega (f^2/(2g))v\,\mathrm{d}x, \tag{17.61}$$

$$L_2(f,g;v) = L(g,3;v) = \int_\Omega (g^2/6)v\,\mathrm{d}x. \tag{17.62}$$

This feature is implemented with `replace`, as illustrated in this case:

UFL code

```
V = FiniteElement("Lagrange", cell, 1)
v = TestFunction(V)
f = Coefficient(V)
g = Coefficient(V)
L = f**2 / (2*g)*v*dx
L2 = replace(L, { f: g, g: 3})
L3 = g**2 / 6*v*dx
```

Here `L2` and `L3` represent exactly the same form. Since they depend only on `g`, the code generated for these forms can be more efficient.

17.5.4 Action

In some applications the matrix is not needed explicitly, only the action of the matrix on a vector. Assembling the resulting vector directly can be much more efficient than assembling the sparse matrix and then performing the matrix-vector multiplication. Assume `a` is a bilinear form and `w` is a `Coefficient` defined on the same finite element as the trial function in `a`. Let A denote the sparse matrix that can be assembled from `a`. Then you can assemble the action of A on a vector directly by defining a linear form L representing the action of a bilinear form a on a function w. The notation for this is simply `L = action(a, w)`, or even shorter `L = a*w`.

17.5.5 Splitting a system

If you prefer to write your PDEs with all terms on one side such as

$$a(u, v) - L(v) = 0, \tag{17.63}$$

you can declare forms with both linear and bilinear terms and split the equations into a and L afterwards. A simple example is

UFL code

```
V = FiniteElement("Lagrange", cell, 1)
u = TrialFunction(V)
v = TestFunction(V)
f = Coefficient(V)
pde = u*v*dx - f*v*dx
a, L = system(pde)
```

Here `system` is used to split the PDE into its bilinear and linear parts. Alternatively, `lhs` and `rhs` can be used to obtain the two parts separately. Make note of the resulting sign of the linear part, which corresponds to moving L to the right-hand side in Equation (17.63).

17.5.6 Computing the sensitivity of a function

If you have found the solution u to Equation (17.63), and u depends on some constant scalar value c, you can compute the sensitivity of u w.r.t. changes in c. If u is represented by a coefficient vector x that is the solution to the algebraic linear system $Ax = b$, the coefficients of $\frac{\partial u}{\partial c}$ are $\frac{\partial x}{\partial c}$. Applying $\frac{\partial}{\partial c}$ to $Ax = b$ and using the chain rule, we can write

$$A\frac{\partial x}{\partial c} = \frac{\partial b}{\partial c} - \frac{\partial A}{\partial c}x, \tag{17.64}$$

and thus $\frac{\partial x}{\partial c}$ can be found by solving the same algebraic linear system used to compute x, only with a different right-hand side. The linear form corresponding to the right-hand side of Equation (17.64) can be written

UFL code

```
u = Coefficient(element)
sL = diff(L, c) - action(diff(a, c), u)
```

or you can use the equivalent form transformation

UFL code

```
sL = sensitivity_rhs(a, u, L, c)
```

Note that the solution `u` must be represented by a `Coefficient`, while u in $a(u, v)$ is represented by a `Argument`.

17.6 Expression representation

From a high level view, UFL is all about defining forms. Each form contains one or more scalar integrand expressions, but the form representation is largely disconnected from the representation of the integrand expressions. Indeed, most of the complexity of the UFL implementation is related to expressing, representing, and manipulating expressions. The rest of this chapter will focus on expression representations and algorithms operating on them. These topics will be of little interest to the average user of UFL, and more directed towards developers and curious technically oriented users.

To reason about expression algorithms without the burden of implementation details, we need an abstract notation for the structure of an expression. UFL expressions are representations of programs, and the notation should allow us to see this connection. Below we will discuss the properties of expressions both in terms of this abstract notation, and related to specific implementation details.

17.6.1 The structure of an expression

The most basic expressions, which have no dependencies on other expressions, are called *terminal expressions*. Other expressions result from applying some operator to one or more existing expressions. Consider an arbitrary (non-terminal) expression z. This expression depends on a set of terminal expressions $\{t_i\}$, and is computed using a set of operators $\{f_i\}$. If each subexpression of z is labeled with an integer, an abstract program can be written to compute z by computing a sequence of subexpressions $\langle y_i \rangle_{i=1}^{n}$ and setting $z = y_n$. Algorithm 5 shows such a program.

Algorithm 5 Program to compute an expression z.

1: **for** $i \leftarrow 1, \ldots, m$ **do**
2: $\quad y_i := t_i =$ terminal expression
3: **end for**
4: **for** $i \leftarrow m+1, \ldots, n$ **do**
5: $\quad y_i := f_i(\langle y_j \rangle_{j \in \mathfrak{I}_i})$
6: **end for**
7: $z := y_n$

Each terminal expression t_i is a literal constant or input argument to the program. This includes coefficients, basis functions, and geometric quantities. A non-terminal subexpression y_i is the result of

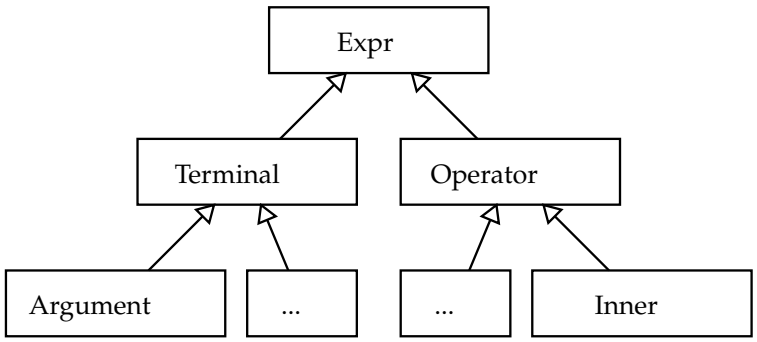


Figure 17.2: Expression class hierarchy.

applying an operator f_i to a sequence of previously computed expressions $\langle y_j \rangle_{j \in \mathfrak{I}_i}$, where $\mathfrak{I}_i$ is an ordered sequence of expression labels. Note that the order in which subexpressions must be computed to produce the same value of z is not unique. For correctness we only require $j < i \; \forall \, j \in \mathfrak{I}_i$, such that all dependencies of a subexpression y_i has been computed before y_i. In particular, all terminals are numbered first in this abstract algorithm for notational convenience only.

The program to compute z can be represented as a graph, where each expression y_i corresponds to a graph vertex. There is a directed graph edge $e = (i, j)$ from y_i to y_j if $j \in \mathfrak{I}_i$, that is if y_i depends on the value of y_j. More formally, the graph G representing the computation of z consists of a set of vertices V and a set of edges E defined by:

$$G = (V, E), \tag{17.65}$$

$$V = \langle v_i \rangle_{i=1}^{n} = \langle y_i \rangle_{i=1}^{n}, \tag{17.66}$$

$$E = \{e_k\} = \bigcup_{i=1}^{n} \{(i, j) \; \forall \, j \in \mathfrak{I}_i\}. \tag{17.67}$$

This graph is clearly directed, since dependencies have a direction. It is acyclic, since an expression can only be constructed from existing expressions. Thus a UFL expression can be represented by a directed acyclic graph (DAG). There are two ways this DAG can be represented in UFL. While defining expressions, a linked representation called the expression tree is built. Technically this is still a DAG since vertices can be reused in multiple subexpressions, but the representation emphasizes the tree like structure of the DAG. The other representation is called the computational graph, which closely mirrors the definition of G above. This representation is mostly useful for form compilers. The details of these two DAG representations will be explained below. They both share the representation of a vertex in the graph as an expression object, which will be explained next.

17.6.2 Expression objects

Recall from Algorithm 5 that non-terminals are expressions $y_i = f_i(\langle y_j \rangle_{j \in \mathfrak{I}_i})$. The operator f_i is represented by the class of the expression object, while the expression y_i is represented by the instance of this class. In the UFL implementation, each expression object is an instance of some subclass of `Expr`. The class `Expr` is the superclass of a hierarchy containing all terminal expression types and operator types supported by UFL. `Expr` has two direct subclasses, `Terminal` and `Operator`, which divides the expression type hierarchy in two, as illustrated in Figure 17.2.

All expression objects are considered immutable; once constructed an expression object will never be modified. Manipulating an expression should always result in a new object being created. The immutable property ensures that expression objects can be reused and shared between expressions without side effects in other parts of a program. This both reduces memory usage, avoids needless

copying of objects, and simplifies recognition of common subexpressions.

Calling `e.operands()` on an `Expr` object `e` representing y_i returns a tuple with expression objects representing $\langle y_j \rangle_{j \in \mathfrak{I}_i}$. Note that this also applies to terminals where there are no outgoing edges and `t.operands()` returns an empty tuple. Instead of modifying the operands of an expression object, a new expression object of the same type can be constructed with modified operands using `e.reconstruct(operands)`, where `operands` is a tuple of expression objects. If the operands are the same this function returns the original object, allowing many algorithms to save memory without additional complications. The invariant `e.reconstruct(e.operands()) == e` should always hold.

17.6.3 Expression properties

In Section 17.4.2 the tensor algebra and index notation capabilities of UFL was discussed. Expressions can be scalar or tensor-valued, with arbitrary rank and shape. Therefore, each expression object `e` has a value shape `e.shape()`, which is a tuple of integers with the dimensions in each tensor axis. Scalar expressions have shape `()`. Another important property is the set of free indices in an expression, obtained as a tuple using `e.free_indices()`. Although the free indices have no ordering, they are represented with a tuple of `Index` instances for simplicity. Thus the ordering within the tuple carries no meaning.

UFL expressions are referentially transparent with some exceptions. Referential transparency means that a subexpression can be replaced by another representation of its value without changing the meaning of the expression. A key point here is that the value of an expression in this context includes the tensor shape and set of free indices. Another important point is that the derivative of a function $f(v)$ in a point, $f'(v)|_{v=g}$, depends on function values in the vicinity of $v = g$. The effect of this dependency is that operator types matter when differentiating, not only the current value of the differentiation variable. In particular, a `Variable` cannot be replaced by the expression it represents, because `diff` depends on the `Variable` instance and not the expression it has the value of. Similarly, replacing a `Coefficient` with some value will change the meaning of an expression that contains derivatives w.r.t. function coefficients.

The following example illustrate the issue with `Variable` and `diff`.

UFL code

```
e = 0
v = variable(e)
f = sin(v)
g = diff(f, v)
```

Here `v` is a variable that takes on the value 0, but `sin(v)` cannot be simplified to 0 since the derivative of `f` then would be 0. The correct result here is `g = cos(v)`. Printing `f` and `g` gives the strings `sin(var1(0))` and `d/d[var1(0)] (sin(var1(0)))`. Try just setting `v = e` and see how `f` and `g` becomes zero.

17.6.4 Tree representation

The expression tree does not have a separate data structure. It is merely a way of viewing the structure of an expression. Any expression object `e` can be seen as the root of a tree, where `e.operands()` returns its children. If some of the children are equal, they will appear as many times as they appear in the expression. Thus it is easy to traverse the tree nodes; that is, v_i in the DAG, but eventual reuse of subexpressions is not directly visible. Edges in the DAG does not appear explicitly, and the list of vertices can only be obtained by traversing the tree recursively and selecting unique objects.

An expression tree for the stiffness term grad u : grad v is illustrated in Figure 17.3. The terminals u and v have no children, and the term grad u is itself represented by a tree with two nodes. Each time

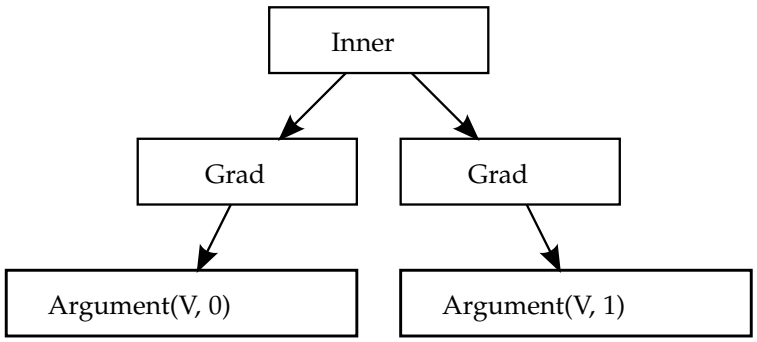


Figure 17.3: Expression tree for grad u : grad v.

an operator is applied to some expressions, it will return a new tree root that references its operands. Note that the user will apply the functions `grad` and `inner` in her use of the language, while the names `Grad`, `Inner` and `Argument` in this figure are the names of the `Expr` subclasses used in UFL to represent the expression objects. In other words, taking the gradient of an expression with `grad(u)` gives an expression representation `Grad(u)`, and `inner(a, b)` gives an expression representation `Inner(a, b)`. This separation of language and representation is merely a design choice in the implementation of UFL.

17.6.5 Graph representation

When viewing an expression as a tree, the lists of all unique vertices and edges are not directly available. Representing the DAG more directly allows many algorithms to be simplified or optimized. UFL includes tools to build an array based representation of the DAG, the *computational graph*, from any expression. The computational graph $G = V, E$ is a data structure based on flat arrays, directly mirroring the definition of the graph in equations (17.65)–(17.67). This representation gives direct access to dependencies between subexpressions, and allows easy iteration over unique vertices. The graph is constructed easily with the lines:

Python code

```
from ufl.algorithms import Graph
G = Graph(expression)
V, E = G
```

One array (Python list) `V` is used to store the unique vertices $\langle v_i \rangle_{i=1}^{n}$ of the DAG. For each vertex v_i an expression node y_i is stored to represent it. Thus the expression tree for each vertex is also directly available, since each expression node is the root of its own expression tree. The edges are stored in an array `E` with integer tuples `(i,j)` representing an edge from v_i to v_j; that is, v_j is an operand of v_i. The vertex list in the graph is built using a postordering from a depth first traversal, which guarantees that the vertices are topologically sorted such that $j < i \; \forall \, j \in \mathfrak{I}_i$.

Let us look at an example of a computational graph. The following code defines a simple expression and then prints the vertices and edges of its graph.

Python code

```
from ufl import *
cell = triangle
V = FiniteElement("Lagrange", cell, 1)
u = TrialFunction(V)
v = TestFunction(V)
c = Constant(cell)
f = Coefficient(V)
```

```
e = c*f**2*u*v

from ufl.algorithms import Graph, partition
G = Graph(e)
V, E, = G

print "str(e) = %s\n" % str(e)
print "\n".join("V[%d] = %s" % (i, v) for (i, v) in enumerate(V)), "\n"
print "\n".join("E[%d] = %s" % (i, e) for (i, e) in enumerate(E)), "\n"
```

An excerpt of the program output is shown here:

Generated code

```
V[0] = v_{-2}
...
V[7] = v_{-1} * c_0 * w_1 ** 2
V[8] = v_{-2} * v_{-1} * c_0 * w_1 ** 2
...
E[6] = (8, 0)
E[7] = (8, 7)
```

The two last edges shown here represent the dependencies of vertex 8 on vertex 7 and 0, since $v_8 = v_0 v_7$. Run the code to see the full output of this code. Try changing the expression and see what the graph looks like.

From the edges E, related arrays can be computed efficiently; in particular the vertex indices of dependencies of a vertex v_i in both directions are useful:

$$\begin{aligned} V_{out} &= \langle \mathfrak{I}_i \rangle_{i=1}^{n}, \\ V_{in} &= \left\langle \{j | i \in \mathfrak{I}_j\} \right\rangle_{i=1}^{n} \end{aligned} \tag{17.68}$$

These arrays can be easily constructed for any expression:

Python code

```
Vin = G.Vin()
Vout = G.Vout()
```

Similar functions exist for obtaining indices into E for all incoming and outgoing edges. A nice property of the computational graph built by UFL is that no two vertices will represent the same identical expression. During graph building, subexpressions are inserted in a hash map (Python dictionary) to achieve this. Some expression classes sort their arguments uniquely such that e.g. `a*b` and `b*a` will become the same vertex in the graph.

Free indices in expression nodes can complicate the interpretation of the linearized graph when implementing some algorithms, because an expression object with free indices represents not one value but a set of values, one for each permutation of the values its free indices can have. One solution to this can be to apply `expand_indices` before constructing the graph, which will replace all expressions with free indices with equivalent expressions with explicit fixed indices. Note however that free indices cannot be regained after expansion. See Section 17.8.3 for more about this transformation.

17.6.6 Partitioning

UFL is intended as a front-end for form compilers. Since the end goal is generation of code from expressions, some utilities are provided for the code generation process. In principle, correct code can be generated for an expression from its computational graph simply by iterating over the vertices and generating code for each operation separately, basically mirroring Algorithm 5. However, a good

form compiler should be able to produce better code. UFL provides utilities for partitioning the computational graph into subgraphs (partitions) based on dependencies of subexpressions, which enables quadrature based form compilers to easily place subexpressions inside the right sets of loops. The function `partition` implements this feature. Each partition is represented by a simple array of vertex indices, and each partition is labeled with a set of dependencies. By default, this set of dependencies use the strings `x`, `c`, and `v%d` to denote dependencies on spatial coordinates, cell specific quantities, and form arguments (not coefficients) respectively.

The following example code partitions the graph built above, and prints vertices in groups based on their dependencies.

Python code

```
partitions, keys = partition(G)
for deps in sorted(partitions.keys()):
    P = partitions[deps]
    print "The following depends on", tuple(deps)
    for i in sorted(P):
        print "V[%d] = %s" % (i, V[i])
```

The output text from the program is included below. Notice that the literal constant 2 has no dependencies. Expressions in this partition can always be precomputed at compile-time. The `Constant c_0` depends on data which varies for each cell, represented by `c` in the dependency set, but not on spatial coordinates, so it can be placed outside the quadrature loop. The `Function w_1` and expressions depending on it depends in addition on the spatial coordinates, represented by `x`, and therefore needs to be computed for each quadrature point. Expressions depending on only the test or trial function are marked with `v%d` where the number is the internal counter used by UFL to distinguish between arguments. Note that test and trial functions are here marked as depending on the spatial coordinates, but not on cell dependent quantities. This is only true for finite elements defined on a local reference element, in which case the basis functions can be precomputed in each quadrature point. The actual run-time dependencies of a basis function in a finite element space is unknown to UFL, which is why the partition function takes an optional multifunction argument such that the form compiler writer can provide more accurate dependencies. We refer to the implementation of `partition` for such implementation details.

Generated code

```
The following depends on ()
V[4] = 2
The following depends on ("c",)
V[2] = c_0
The following depends on ("x", "c")
V[3] = w_1
V[5] = w_1 ** 2
V[6] = c_0 * w_1 ** 2
The following depends on ("x", "v-1")
V[1] = v_{-1}
The following depends on ("x", "c", "v-1")
V[7] = v_{-1} * c_0 * w_1 ** 2
The following depends on ("x", "v-2")
V[0] = v_{-2}
The following depends on ("x", "c", "v-2", "v-1")
V[8] = v_{-2} * v_{-1} * c_0 * w_1 ** 2
```

17.7 Computing derivatives

When any kind of derivative expression is declared by the end-user of the form language, an expression object is constructed to represent it, but nothing is computed. The type of this expression object is a subclass of `Derivative`. Before low level code can be generated from the derivative expression, some kind of algorithm to evaluate derivatives must be applied, since differential operators are not available natively in low level languages such as C++. Computing exact derivatives is important, which rules out approximations by divided differences. Several alternative algorithms exist for computing exact derivatives. All relevant algorithms are based on the chain rule combined with differentiation rules for each expression object type. The main differences between the algorithms are in the extent of which subexpressions are reused, and in the way subexpressions are accumulated.

Mixing derivative computation into the code generation strategy of each form compiler would lead to a significant duplication of implementation effort. To separate concerns and keep the code manageable, differentiation is implemented as part of UFL in such a way that the form compilers are independent of the differentiation strategy chosen in UFL. Therefore, it is advantageous to use the same representation for the evaluated derivative expressions as for any other expression. Before expressions are interpreted by a form compiler, differential operators should be evaluated such that the only operators left are non-differential operators. An exception is made for spatial derivatives of terminals which are unknown to UFL because they are provided by the form compilers.

Below, the differences and similarities between some of the simplest algorithms are discussed. After the algorithm currently implemented in UFL has been explained, extensions to tensor and index notation and higher order derivatives are discussed. Finally, the section is closed with some remarks about the differentiation rules for terminal expressions.

17.7.1 Approaches to computing derivatives

Algorithms for computing derivatives are designed with different end goals in mind. Symbolic Differentiation (SD) takes as input a single symbolic expression and produces a new symbolic expression for its derivative. Automatic Differentiation (AD) takes as input a program to compute a function and produces a new program to compute the derivative of the function. Several variants of AD algorithms exist, the two most common being Forward Mode AD and Reverse Mode AD (Griewank, 1989). More advanced algorithms exist, and is an active research topic. A UFL expression is a symbolic expression, represented by an expression tree. But the expression tree is a directed acyclic graph that represents a program to evaluate said expression. Thus it seems the line between SD and AD becomes less distinct in this context.

Naively applied, SD can result in huge expressions, which can both require a lot of memory during the computation and be highly inefficient if written to code directly. However, some illustrations of the inefficiency of symbolic differentiation, such as in Griewank (1989), are based on computing closed form expressions of derivatives in some stand-alone computer algebra system (CAS). Copying the resulting large expressions directly into a computer code can lead to very inefficient code. The compiler may not be able to detect common subexpressions, in particular if simplification and rewriting rules in the CAS has changed the structure of subexpressions with a potential for reuse.

In general, AD is capable of handling algorithms that SD can not. A tool for applying AD to a generic source code must handle many complications such as subroutines, global variables, arbitrary loops and branches (Bischof et al., 1992, 2002; Giering and Kaminski, 1998). Since the support for program flow constructs in UFL is very limited, the AD implementation in UFL will not run into such complications. In Section 17.7.2 the similarity between SD and forward mode AD in the context of UFL is explained in more detail.

17.7.2 Forward mode automatic differentiation

Recall Algorithm 5, which represents a program for computing an expression z from a set of terminal values $\{t_i\}$ and a set of elementary operations $\{f_i\}$. Assume for a moment that there are no differential operators among $\{f_i\}$. The algorithm can then be extended to compute the derivative $\frac{dz}{dv}$, where v represents a differentiation variable of any kind. This extension gives Algorithm 6.

Algorithm 6 Forward mode AD on Algorithm 5.

1: **for** $i \leftarrow 1, \ldots, m$ **do**
2: $\quad y_i := t_i$
3: $\quad \frac{dy_i}{dv} := \frac{dt_i}{dv}$
4: **end for**
5: **for** $i \leftarrow m+1, \ldots, n$ **do**
6: $\quad y_i := f_i(\langle y_j \rangle_{j \in \mathfrak{I}_i})$
7: $\quad \frac{dy_i}{dv} := \sum_{k \in \mathfrak{I}_i} \frac{\partial f_i}{\partial y_k} \frac{dy_k}{dv}$
8: **end for**
9: $z := y_n$
10: $\frac{dz}{dv} := \frac{dy_n}{dv}$

This way of extending a program to simultaneously compute the expression z and its derivative $\frac{dz}{dv}$ is called forward mode automatic differentiation (AD). By renaming y_i and $\frac{dy_i}{dv}$ to a new sequence of values $\langle \hat{y}_j \rangle_{j=1}^{\hat{n}}$, Algorithm 6 can be rewritten as shown in Algorithm 7, which is isomorphic to Algorithm 5 (they have exactly the same structure).

Algorithm 7 Program to compute $\frac{dz}{dv}$ produced by forward mode AD

1: **for** $i \leftarrow 1, \ldots, \hat{m}$ **do**
2: $\quad \hat{y}_i := \hat{t}_i$
3: **end for**
4: **for** $i \leftarrow \hat{m}+1, \ldots, \hat{n}$ **do**
5: $\quad \hat{y}_i := \hat{f}_i(\langle \hat{y}_j \rangle_{j \in \hat{\mathfrak{I}}_i})$
6: **end for**
7: $\frac{dz}{dv} := \hat{y}_{\hat{n}}$

Since the program in Algorithm 5 can be represented as a DAG, and Algorithm 7 is isomorphic to Algorithm 5, the program in Algorithm 7 can also be represented as a DAG. Thus a program to compute $\frac{dz}{dv}$ can be represented by an expression tree built from terminal values and non-differential operators.

The currently implemented algorithm for computing derivatives in UFL follows forward mode AD closely. Since the result is a new expression tree, the algorithm can also be called symbolic differentiation. In this context, the differences between the two are implementation details. To ensure that we can reuse expressions properly, simplification rules in UFL avoids modifying the operands of an operator. Naturally repeated patterns in the expression can therefore be detected easily by the form compilers. Efficient common subexpression elimination can then be implemented by placing subexpressions in a hash map. However, there are simplifications such as $0 * f \to 0$ and $1 * f \to f$, called constant folding, which simplify the result of the differentiation algorithm automatically as it is being constructed. These simplifications are crucial for the memory use during derivative computations, and the performance of the resulting program.

17.7.3 Extensions to tensors and indexed expressions

So far we have not considered derivatives of non-scalar expression and expressions with free indices. This issue does not affect the overall algorithms, but it does affect the local derivative rules for each expression type.

Consider the expression `diff(A, B)` with `A` and `B` matrix expressions. The meaning of derivatives of tensors w.r.t. to tensors is easily defined via index notation, which is heavily used within the differentiation rules:

$$\frac{dA}{dB} = \frac{dA_{ij}}{dB_{kl}} e_i \otimes e_j \otimes e_k \otimes e_l \tag{17.69}$$

Derivatives of subexpressions are frequently evaluated to literal constants. For indexed expressions, it is important that free indices are propagated correctly with the derivatives. Therefore, differentiated expressions will some times include literal constants annotated with free indices.

There is one rare and tricky corner case when an index sum binds an index i such as in $(v_i v_i)$ and the derivative w.r.t. x_i is attempted. The simplest example of this is the expression $(v_i v_i)_{,j}$, which has one free index j. If j is replaced by i, the expression can still be well defined, but you would never write $(v_i v_i)_{,i}$ manually. If the expression in the parenthesis is defined in a variable `e = v[i]*v[i]`, the expression `e.dx(i)` looks innocent. However, this will cause problems as derivatives (including the index i) are propagated up to terminals. If this case is encountered in the current implementation of UFL, it will be detected and an error message will be triggered. To work around the problem, simply use different index instances. In a future version of UFL, this case may be handled by relabeling indices to change any expression $(\sum_i e_i)_{,i}$ into $(\sum_j e_j)_{,i}$.

17.7.4 Higher order derivatives

A simple forward mode AD implementation such as Algorithm 6 only considers one differentiation variable. Higher order or nested differential operators must also be supported, with any combination of differentiation variables. A simple example illustrating such an expression can be

$$a = \frac{d}{dx}\left(\frac{d}{dx} f(x) + 2\frac{d}{dy} g(x,y)\right). \tag{17.70}$$

Considerations for implementations of nested derivatives in a functional[2] framework have been explored in several papers (Karczmarczuk, 2001; Pearlmutter and Siskind, 2007; Siskind and Pearlmutter, 2008).

In the current UFL implementation this is solved in a different fashion. Considering Equation (17.70), the approach is simply to compute the innermost derivatives $\frac{d}{dx} f(x)$ and $\frac{d}{dy} g(x,y)$ first, and then computing the outer derivatives. This approach is possible because the result of a derivative computation is represented as an expression tree just as any other expression. Mainly this approach was chosen because it is simple to implement and easy to verify. Whether other approaches are faster has not been investigated. Furthermore, alternative AD algorithms such as reverse mode can be experimented with in the future without concern for nested derivatives in the first implementations.

An outer controller function `apply_ad` handles the application of a single variable AD routine to an expression with possibly nested derivatives. The AD routine is a function accepting a derivative expression node and returning an expression where the single variable derivative has been computed.

[2]Functional as in functional languages.

Python code

```
def apply_ad(e, ad_routine):
    if isinstance(e, Terminal):
        return e
    ops = [apply_ad(o, ad_routine) for o in e.operands()]
    e = e.reconstruct(*ops)
    if isinstance(e, Derivative):
        e = ad_routine(e)
    return e
```

Figure 17.4: Simple implementation of recursive `apply_ad` procedure.

This routine can be an implementation of Algorithm 7. The result of `apply_ad` is mathematically equivalent to the input, but with no derivative expression nodes left[3].

The function `apply_ad` works by traversing the tree recursively in post-order, discovering subtrees where the root represents a derivative, and applying the provided AD routine to the derivative subtree. Since the children of the derivative node has already been visited by `apply_ad`, they are guaranteed to be free of derivative expression nodes and the AD routine only needs to handle the case discussed above with algorithms 6 and 7.

The complexity of the `ad_routine` should be $O(n)$, with n being the size of the expression tree. The size of the derivative expression is proportional to the original expression. If there are d derivative expression nodes in the expression tree, the complexity of this algorithm is $O(dn)$, since `ad_routine` is applied to subexpressions d times. As a result the worst case complexity of `apply_ad` is $O(n^2)$, but in practice $d \ll n$. A recursive implementation of this algorithm is shown in Figure 17.4.

17.7.5 Basic differentiation rules

To implement the algorithm descriptions above, we must implement differentiation rules for all expression node types. Derivatives of operators can be implemented as generic rules independent of the differentiation variable, and these are well known and not mentioned here. Derivatives of terminals depend on the differentiation variable type. Derivatives of literal constants are of course always zero, and only spatial derivatives of geometric quantities are nonzero. Since form arguments are unknown to UFL (they are provided externally by the form compilers), their spatial derivatives ($\frac{\partial \phi^k}{\partial x_i}$ and $\frac{\partial w^k}{\partial x_i}$) are considered input arguments as well. In all derivative computations, the assumption is made that form coefficients have no dependencies on the differentiation variable. Two more cases needs explaining, the user defined variables and derivatives w.r.t. the coefficients of a `Coefficient`.

If v is a `Variable`, then we define $\frac{dt}{dv} \equiv 0$ for any terminal t. If v is scalar valued then $\frac{dv}{dv} \equiv 1$. Furthermore, if V is a tensor valued `Variable`, its derivative w.r.t. itself is

$$\frac{dV}{dV} = \frac{dV_{ij}}{dV_{kl}} e_i \otimes e_j \otimes e_k \otimes e_l = \delta_{ik}\delta_{jl} e_i \otimes e_j \otimes e_k \otimes e_l. \tag{17.71}$$

In addition, the derivative of a variable w.r.t. something else than itself equals the derivative of the expression it represents:

$$v = g, \tag{17.72}$$

$$\frac{dv}{dz} = \frac{dg}{dz}. \tag{17.73}$$

Finally, we consider the operator `derivative`, which represents differentiation w.r.t. all coefficients

[3] Except direct spatial derivatives of form arguments, but that is an implementation detail.

$\{w_k\}$ of a function w. Consider an object `element` which represents a finite element space V_h with a basis $\{\phi_k\}$. Next consider form arguments defined in this space:

UFL code

```
v = Argument(element)
w = Coefficient(element)
```

The Argument instance `v` represents any $v \in \{\phi_k\}$, while the `Coefficient` instance `w` represents the sum

$$w = \sum_k w_k \phi_k(x). \tag{17.74}$$

The derivative of `w` w.r.t. any w_k is the corresponding basis function in V_h,

$$\frac{\partial w}{\partial w_k} = \phi_k, \quad k = 1, \ldots, |V_h|, \tag{17.75}$$

$$\tag{17.76}$$

which can be represented by `v`, since

$$v \in \langle \phi_k \rangle_{k=1}^{|V_h|} = \left\langle \frac{\partial w}{\partial w_k} \right\rangle_{k=1}^{|V_h|}. \tag{17.77}$$

Note that `v` should be a basis function instance that has not already been used in the form.

17.8 Algorithms

In this section, some central algorithms and key implementation issues are discussed, much of which relates to the Python programming language. Thus, this section is mainly intended for developers and others who need to relate to UFL on a technical level. Python users may also find some of the techniques here interesting.

17.8.1 Effective tree traversal in Python

Applying some action to all nodes in a tree is naturally expressed using recursion:

Python code

```
def walk(expression, pre_action, post_action):
    pre_action(expression)
    for o in expression.operands():
        walk(o)
    post_action(expression)
```

This implementation simultaneously covers pre-order traversal, where each node is visited before its children, and post-order traversal, where each node is visited after its children.

A more "pythonic" way to implement iteration over a collection of nodes is using generators. A minimal implementation of this could be

Python code

```
def post_traversal(root):
    for o in root.operands():
        yield post_traversal(o)
    yield root
```

which then enables the natural Python syntax for iteration over expression nodes:

Python code

```
for e in post_traversal(expression):
    post_action(e)
```

For efficiency, the actual implementation of `post_traversal` in UFL is not using recursion. Function calls are very expensive in Python, which makes the non-recursive implementation an order of magnitude faster than the above.

17.8.2 Type based function dispatch in Python

A common task in both symbolic computing and compiler implementation is the selection of some operation based on the type of an expression node. For a selected few operations, this is done using overloading of functions in the subclasses of `Expr`, but this is not suitable for all operations. In many cases type-specific operations are better implemented together in the algorithm instead of distributed across class definitions. This implementation pattern is called the Visitor pattern (Gamma et al., 1995). The implementation in UFL is somewhat different from the patterns used in a statically typed language such as C++.

One way to implement type based operation selection is to use a *type switch*, which is a sequence of if-tests as shown here:

Python code

```
def operation(expression):
    if isinstance(expression, IntValue):
        return int_operation(expression)
    elif isinstance(expression, Sum):
        return sum_operation(expression)
    # etc.
```

There are several problems with this approach, one of which is efficiency when there are many types to check. A type based function dispatch mechanism with efficiency independent of the number of types is implemented as an alternative through the class `MultiFunction`. The underlying mechanism is a dictionary lookup (which is $O(1)$) based on the type of the input argument, followed by a call to the function found in the dictionary. The lookup table is built in the `MultiFunction` constructor only once. Functions to insert in the table are discovered automatically using the introspection capabilities of Python.

A multifunction is declared as a subclass of `MultiFunction`. For each type that should be handled particularly, a member function is declared in the subclass. The `Expr` classes use the `CamelCaps` naming convention, which is automatically converted to `underscore_notation` for corresponding function names, such as `IndexSum` and `index_sum`. If a handler function is not declared for a type, the closest superclass handler function is used instead. Note that the `MultiFunction` implementation is specialized to types in the `Expr` class hierarchy. The declaration and use of a multifunction is illustrated in this example code:

Python code

```
class ExampleFunction(MultiFunction):
    def __init__(self):
        MultiFunction.__init__(self)

    def terminal(self, expression):
        return "Got a Terminal subtype %s." % type(expression)
```

```
    def operator(self, expression):
        return "Got an Operator subtype %s." % type(expression)

    def argument(self, expression):
        return "Got an Argument."

    def sum(self, expression):
        return "Got a Sum."

m = ExampleFunction()

cell = triangle
element = FiniteElement("Lagrange", cell, 1)
x = cell.x
print m(Argument(element))
print m(x)
print m(x[0] + x[1])
print m(x[0] * x[1])
```

Note that `argument` and `sum` will handle instances of the exact types `Argument` and `Sum`, while `terminal` and `operator` will handle the types `SpatialCoordinate` and `Product` since they have no specific handlers.

17.8.3 *Implementing expression transformations*

Many transformations of expressions can be implemented recursively with some type-specific operation applied to each expression node. Examples of operations are converting an expression node to a string representation, to an expression representation using an symbolic external library, or to a UFL representation with some different properties. A simple variant of this pattern can be implemented using a multifunction to represent the type-specific operation:

Python code

```
def apply(e, multifunction):
    ops = [apply(o, multifunction) for o in e.operands()]
    return multifunction(e, *ops)
```

The basic idea is as follows. Given an expression node `e`, begin with applying the transformation to each child node. Then return the result of some operation specialized according to the type of `e`, using the already transformed children as input.

The `Transformer` class implements this pattern. Defining a new algorithm using this pattern involves declaring a `Transformer` subclass, and implementing the type specific operations as member functions of this class just as with `MultiFunction`. The difference is that member functions take one additional argument for each operand of the expression node. The transformed child nodes are supplied as these additional arguments. The following code replaces terminal objects with objects found in a dictionary `mapping`, and reconstructs operators with the transformed expression trees. The algorithm is applied to an expression by calling the function `visit`, named after the similar Visitor pattern.

Python code

```
class Replacer(Transformer):
    def __init__(self, mapping):
        Transformer.__init__(self)
        self.mapping = mapping

    def operator(self, e, *ops):
        return e.reconstruct(*ops)
```

```
    def terminal(self, e):
        return self.mapping.get(e, e)

f = Constant(triangle)
r = Replacer({f: f**2})
g = r.visit(2*f)
```

After running this code the result is $g = 2f^2$. The actual implementation of the `replace` function is similar to this code.

In some cases, child nodes should not be visited before their parent node. This distinction is easily expressed using `Transformer`, simply by omitting the member function arguments for the transformed operands. See the source code for many examples of algorithms using this pattern.

17.8.4 Important transformations

There are many ways in which expression representations can be manipulated. Here, we describe three particularly important transformations. Note that each of these algorithms removes some abstractions, and hence may remove some opportunities for analysis or optimization. To demonstrate their effect, each transformation will be applied below to the expression

$$a = \operatorname{grad}(fu) \cdot \operatorname{grad} v. \tag{17.78}$$

At the end of the section, some example code is given to demonstrate more representation details.

Some operators in UFL are termed "compound" operators, meaning they can be represented by other more elementary operators. Try defining an expression `a = dot(grad(f*u), grad(v))`, and print `repr(a)`. As you will see, the representation of `a` is `Dot(Grad(Product(f, u)), Grad(v))`, with some more details in place of `f`, `u` and `v`. By representing the gradient directly with a high level type `Grad` instead of more low level types, the input expressions are easier to recognize in the representation, and rendering of expressions to for example LaTeX format can show the original compound operators as written by the end-user. However, since many algorithms must implement actions for each operator type, the function `expand_compounds` is used to replace all expression nodes of "compound" types with equivalent expressions using basic types. When this operation is applied to the input forms from the user, algorithms in both UFL and the form compilers can still be written purely in terms of more basic operators. Expanding the compound expressions from Equation (17.78) results in the expression

$$a_c = \sum_i \frac{\partial v}{\partial x_i} \frac{\partial (uf)}{\partial x_i}. \tag{17.79}$$

Another important transformation is `expand_derivatives`, which applies automatic differentiation to expressions, recursively and for all kinds of derivatives. The end result is that most derivatives are evaluated, and the only derivative operator types left in the expression tree applies to terminals. The precondition for this algorithm is that `expand_compounds` has been applied. Expanding the derivatives in a_c from Equation (17.79) gives us

$$a_d = \sum_i \frac{\partial v}{\partial x_i} (u \frac{\partial f}{\partial x_i} + f \frac{\partial u}{\partial x_i}). \tag{17.80}$$

Index notation and the `IndexSum` expression node type complicate interpretation of an expression tree somewhat, in particular in expressions with nested index sums. Since expressions with free indices will take on multiple values, each expression object represents not only one value but a

set of values. The transformation `expand_indices` then comes in handy. The precondition for this algorithm is that `expand_compounds` and `expand_derivatives` have been applied. The postcondition of this algorithm is that there are no free indices left in the expression. Expanding the indices in Equation (17.80) finally gives

$$a_i = \frac{\partial v}{\partial x_0}(u\frac{\partial f}{\partial x_0} + f\frac{\partial u}{\partial x_0}) + \frac{\partial v}{\partial x_1}(u\frac{\partial f}{\partial x_1} + f\frac{\partial u}{\partial x_1}). \tag{17.81}$$

We started with the higher level concepts gradient and dot product in Equation (17.78), and ended with only scalar addition, multiplication, and partial derivatives of the form arguments. A form compiler will typically start with a_d or a_i, insert values for the argument derivatives, apply some other transformations, before finally generating code.

Some example code to play around with should help in understanding what these algorithms do at the expression representation level. Since the printed output from this code is a bit lengthy, only key aspects of the output is repeated below. Copy this code to a python file or run it in a python interpreter to see the full output.

Python code

```
from ufl import *
V = FiniteElement("Lagrange", triangle, 1)
u = TestFunction(V)
v = TrialFunction(V)
f = Coefficient(V)

# Note no *dx! This is an expression, not a form.
a = dot(grad(f*u), grad(v))

from ufl.algorithms import *
ac = expand_compounds(a)
ad = expand_derivatives(ac)
ai = expand_indices(ad)
print "\na: ", str(a),  "\n", tree_format(a)
print "\nac:", str(ac), "\n", tree_format(ac)
print "\nad:", str(ad), "\n", tree_format(ad)
print "\nai:", str(ai), "\n", tree_format(ai)
```

The print output showing `a` is (with the details of the finite element object cut away for shorter lines):

Output

```
a:  (grad(v_{-2} * w_0)) . (grad(v_{-1}))
Dot
(
    Grad
        Product
        (
            Argument(FiniteElement(...), -2)
            Coefficient(FiniteElement(...), 0)
        )
    Grad
        Argument(FiniteElement(...), -1)
)
```

The arguments labeled -1 and -2 refer to v and u respectively.

In `ac`, the `Dot` product has been expanded to an `IndexSum` of a `Product` with two `Indexed` operands:

Output

```
IndexSum
(
    Product
    (
        Indexed
        (
            ...
            MultiIndex((Index(10),), {Index(10): 2})
        )
        Indexed
        (
            ...
            MultiIndex((Index(10),), {Index(10): 2})
        )
    )
    MultiIndex((Index(10),), {Index(10): 2})
)
```

The somewhat complex looking expression `MultiIndex((Index(10),), {Index(10): 2})` can be read simply as "index named i_{10}, bound to an axis with dimension 2".

Zooming in to one of the `...` lines above, the representation of $\text{grad}(fu)$ must still keep the vector shape after being transformed to more basic expressions, which is why the `SpatialDerivative` object is wrapped in a `ComponentTensor` object:

Output

```
ComponentTensor
(
    SpatialDerivative
    (
        Product
        (
            u
            f
        )
        MultiIndex((Index(8),), {Index(8): 2})
    )
    MultiIndex((Index(8),), {Index(8): 2})
)
```

A common pattern occurs in the algorithmically expanded expressions:

Output

```
Indexed
(
    ComponentTensor
    (
        ...
        MultiIndex((Index(8),), {Index(8): 2})
    )
    MultiIndex((Index(10),), {Index(10): 2})
)
```

This pattern acts as a relabeling of the index objects, renaming i_8 from inside ... to i_{10} on the outside. When looking at the print of `ad`, the result of the chain rule ($(fu)' = uf' + fu'$) can be seen as the `Sum` of two `Product` objects.

Output

```
Sum
(
    Product
    (
        u
        SpatialDerivative
        (
            f
            MultiIndex((Index(8),), {Index(8): 2})
        )
    )
    Product
    (
        f
        SpatialDerivative
        (
            u
            MultiIndex((Index(8),), {Index(8): 2})
        )
    )
)
```

Finally after index expansion in `ai` (not shown here), no free `Index` objects are left, but instead a lot of `FixedIndex` objects can be seen in the print of `ai`. Looking through the full output from the example code above is strongly encouraged if you want a good understanding of the three transformations shown here.

17.8.5 *Evaluating expressions*

Even though UFL expressions are intended to be compiled by form compilers, it can be useful to evaluate them to floating point values directly. In particular, this makes testing and debugging of UFL much easier, and is used extensively in the unit tests. To evaluate an UFL expression, values of form arguments and geometric quantities must be specified. Expressions depending only on spatial coordinates can be evaluated by passing a tuple with the coordinates to the call operator. The following code which can be copied directly into an interactive Python session shows the syntax:

Python code

```
from ufl import *
cell = triangle
x = cell.x
e = x[0] + x[1]
print e((0.5, 0.7)) # prints 1.2
```

Other terminals can be specified using a dictionary that maps from terminal instances to values. This code extends the above code with a mapping:

Python code

```
c = Constant(cell)
e = c*(x[0] + x[1])
print e((0.5, 0.7), { c: 10 }) # prints 12.0
```

If functions and basis functions depend on the spatial coordinates, the mapping can specify a Python callable instead of a literal constant. The callable must take the spatial coordinates as input and return a floating point value. If the function being mapped is a vector function, the callable must return a tuple of values instead. These extensions can be seen in the following code:

Python code

```
element = VectorElement("Lagrange", triangle, 1)
c = Constant(triangle)
f = Coefficient(element)
e = c*(f[0] + f[1])
def fh(x):
    return (x[0], x[1])
print e((0.5, 0.7), { c: 10, f: fh }) # prints 12.0
```

To use expression evaluation for validating that the derivative computations are correct, spatial derivatives of form arguments can also be specified. The callable must then take a second argument which is called with a tuple of integers specifying the spatial directions in which to differentiate. A final example code computing $g^2 + g_{,0}^2 + g_{,1}^2$ for $g = x_0 x_1$ is shown below.

Python code

```
element = FiniteElement("Lagrange", triangle, 1)
g = Coefficient(element)
e = g**2 + g.dx(0)**2 + g.dx(1)**2
def gh(x, der=()):
    if der == ():   return x[0]*x[1]
    if der == (0,): return x[1]
    if der == (1,): return x[0]
print e((2, 3), { g: gh }) # prints 49
```

17.8.6 *Viewing expressions*

Expressions can be formatted in various ways for inspection, which is particularly useful while debugging. The Python built in string conversion operator `str(e)` provides a compact human readable string. If you type `print e` in an interactive Python session, `str(e)` is shown. Another Python built in string operator is `repr(e)`. UFL implements `repr` correctly such that `e == eval(repr(e))` for any expression `e`. The string `repr(e)` reflects all the exact representation types used in an expression, and can therefore be useful for debugging. Another formatting function is `tree_format(e)`, which produces an indented multi-line string that shows the tree structure of an expression clearly, as opposed to `repr` which can return quite long and hard to read strings. Information about formatting of expressions as LaTeX and the dot graph visualization format can be found in the manual.

17.9 *Implementation issues*

17.9.1 *Python as a basis for a domain specific language*

Many of the implementation details detailed in this section are influenced by the initial choice of implementing UFL as an embedded language in Python. Therefore some words about why Python is suitable for this, and why not, are appropriate here.

Python provides a simple syntax that is often said to be close to pseudo-code. This is a good starting point for a domain specific language. Object orientation and operator overloading is well supported, and this is fundamental to the design of UFL. The functional programming features of Python (such as generator expressions) are useful in the implementation of algorithms and form compilers. The built-in data structures `list`, `dict` and `set` play a central role in fast implementations of scalable algorithms.

There is one problem with operator overloading in Python, and that is the comparison operators. The problem stems from the fact that `__eq__` or `__cmp__` are used by the built-in data structures dictionary and set to compare keys, meaning that `a == b` must return a boolean value for `Expr` to be

used as keys. The result is that `__eq__` can not be overloaded to return some `Expr` type representation such as `Equals(a, b)` for later processing by form compilers. The other problem is that `and` and `or` cannot be overloaded, and therefore cannot be used in `conditional` expressions. There are good reasons for these design choices in Python. This conflict is the reason for the somewhat non-intuitive design of the comparison operators in UFL.

17.9.2 Ensuring unique form signatures

The form compilers need to compute a unique signature of each form for use in a cache system to avoid recompilations. A convenient way to define a signature is using `repr(form)`, since the definition of this in Python is `eval(repr(form)) == form`. Therefore `__repr__` is implemented for all `Expr` subclasses.

Some forms are mathematically equivalent even though their representation is not exactly the same. UFL does not use a truly canonical form for its expressions, but takes some measures to ensure that trivially equivalent forms are recognized as such.

Some of the types in the `Expr` class hierarchy (subclasses of `Counted`), has a global counter to identify the order in which they were created. This counter is used by form arguments (both `Argument` and `Coefficient`) to identify their relative ordering in the argument list of the form. Other counted types are `Index` and `Label`, which only use the counter as a unique identifier. Algorithms are implemented for renumbering of all `Counted` types such that all counts start from 0.

In addition, some operator types such as `Sum` and `Product` maintains a sorted list of operands such that `a+b` and `b+a` are both represented as `Sum(a, b)`. This operand sorting is intentionally independent of the numbering of indices because that would not be stable. The reason for this instability is that the result of algorithms for renumbering indices depends on the order of operands. The operand sorting and renumberings combined ensure that the signature of equal forms will stay the same. Note that the representation, and thus the signature, of a form may change with versions of UFL. The following line prints the signature of a form with `expand_derivatives` and renumbering applied.

Python code

```
print repr(preprocess(myform).preprocessed_form)
```

17.9.3 Efficiency considerations

By writing UFL in Python, we clearly do not put peak performance as a first priority. If the form compilation process can blend into the application build process, the performance is sufficient. We do, however, care about scaling performance to handle complicated equations efficiently, and therefore about the asymptotic complexity of the algorithms we use.

To write clear and efficient algorithms in Python, it is important to use the built in data structures correctly. These data structures include in particular `list`, `dict` and `set`. CPython (van Rossum et al.), the reference implementation of Python, implements the data structure `list` as an array, which means append, and pop, and random read or write access are all O(1) operations. Random insertion, however, is O(n). Both `dict` and `set` are implemented as hash maps, the latter simply with no value associated with the keys. In a hash map, random read, write, insertion and deletion of items are all $O(1)$ operations, as long as the key types implement `__hash__` and `__eq__` efficiently. The dictionary data structure is used extensively by the Python language, and therefore particular attention has been given to make it efficient (Kuchling, 2007). Thus to enjoy efficient use of these containers, all `Expr` subclasses must implement these two special functions efficiently. Such considerations have been important for making the UFL implementation perform efficiently.

17.10 Conclusions and future directions

Many additional features can be introduced to UFL. Which features are added will depend on the needs of FEniCS users and developers. Some features can be implemented in UFL alone, but most features will require updates to other parts of the FEniCS project. Thus the future directions for UFL is closely linked to the development of the FEniCS project as a whole.

Improvements to finite element declarations is likely easy to do in UFL. The added complexity will mostly be in the form compilers. Among the current suggestions are space-time elements and time derivatives. Additional geometry mappings and finite element spaces with non-uniform cell types are also possible extensions.

Additional operators can be added to make the language more expressive. Some operators are easy to add because their implementation only affects a small part of the code. More compound operators that can be expressed using elementary operations is easy to add. Additional special functions are easy to add as well, as long as their derivatives are known. Other features may require more thorough design considerations, such as support for complex numbers which will affect large parts of the code.

User friendly notation and support for rapid development are core values in the design of UFL. Having a notation close to the mathematical abstractions allows expression of particular ideas more easily, which can reduce the probability of bugs in user code. However, the notion of metaprogramming and code generation adds another layer of abstraction which can make understanding the framework more difficult for end-users. Good error checking everywhere is therefore very important, to detect user errors as close as possible to the user input. Improvements to the error messages, documentation, and unit test suite will always be helpful, to avoid frequently repeated errors and misunderstandings among new users.

To support the form compiler projects, algorithms and utilities for generating better code more efficiently could be included in UFL. Such algorithms should probably be limited to algorithms such as general transformations of expression graphs which can be useful independently of form compiler specific approaches. In this area, more work on alternative automatic differentiation algorithms (Forth et al., 2004; Tadjouddine, 2008) can be useful.

To summarize, UFL is a central component in the FEniCS framework, where it provides a rich form language, automatic differentiation, and a building block for efficient form compilers. These are useful features in rapid development of applications for efficiently solving partial differential equations. UFL provides the user interface to Automation of Discretization that is the core feature of FEniCS, and adds Automation of Linearization to the framework. With these features, UFL has brought FEniCS one step closer to its overall goal Automation of Mathematical Modeling.

17.11 Acknowledgements

This work has been supported by the Norwegian Research Council (grant 162730) and Simula Research Laboratory. I wish to thank everyone who has helped improving UFL with suggestions and testing, in particular Anders Logg, Kristian Ølgaard, Garth Wells, Harish Narayanan and Marie Rognes. In addition to the two anonymous referees, both Kent-André Mardal and Marie Rognes performed critical reviews which greatly improved this chapter.

18 Unicorn: a unified continuum mechanics solver

By Johan Hoffman, Johan Jansson, Cem Degirmenci, Niclas Jansson and Murtazo Nazarov

This chapter provides a description of the technology of Unicorn focusing on simple, efficient and general algorithms and software for the Unified Continuum (UC) concept and the adaptive General Galerkin (G2) discretization as a unified approach to continuum mechanics. We describe how Unicorn fits into the FEniCS framework, how it interfaces to other FEniCS components, what interfaces and functionality Unicorn provides itself and how the implementation is designed. We also present some examples in fluid–structure interaction and adaptivity computed with Unicorn.

18.1 Background

Unicorn is solver technology (models, methods, algorithms and software) with the goal of automated simulation of realistic continuum mechanics applications, such as drag or lift computation for fixed or flexible objects (fluid–structure interaction) in turbulent incompressible or compressible flow. The basis for Unicorn is Unified Continuum (UC) modeling (Hoffman et al., 2011) formulated in Euler (laboratory) coordinates, together with a G2 (General Galerkin) adaptive stabilized finite element discretization with a moving mesh for tracking the phase interfaces. The UC model consists of canonical conservation equations for mass, momentum, energy and phase over the whole domain as one continuum, together with a Cauchy stress and phase variable as data for defining material properties and constitutive equations. Unicorn formulates and implements the adaptive G2 method applied to the UC model, and interfaces to other components in the FEniCS chain (FIAT, FFC, DOLFIN) providing representation of finite element function spaces, weak forms and mesh, and algorithms such as automated parallel assembly and linear algebra.

The Unicorn software is organized into three parts:

Library The Unicorn library provides common solver technology such as automated time-stepping, error estimation, adaptivity, mesh smoothing and slip/friction boundary conditions.

Solver The Unicorn solver implements the G2 adaptive discretization method for the UC model by formulating the relevant weak forms. There are currently two primary solvers: incompressible fluids and solids (including fluid–structure interaction) and compressible Euler (only fluid). The long-term goal is a unification of the incompressible and compressible formulations.

Applications Associated to the solver(s) are applications such as computational experiments and benchmarks with certain geometries, coefficients and parameters. These are represented as stand-alone programs built on top of the Unicorn solver/library, running in either serial or parallel (currently restricted to incompressible flow).

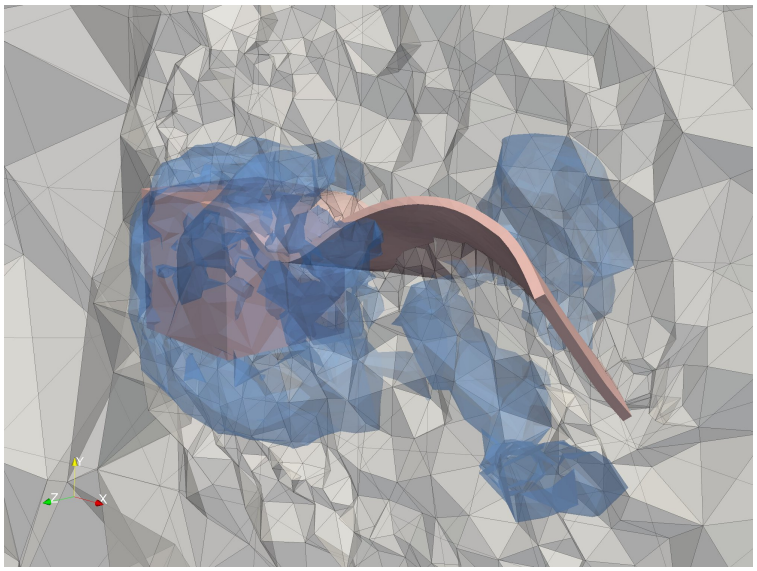


Figure 18.1: A fluid–structure interaction problem consisting of a flag mounted behind a cube in turbulent flow. The plot shows the fluid–structure interface, an isosurface of the pressure and a cut of the mesh.

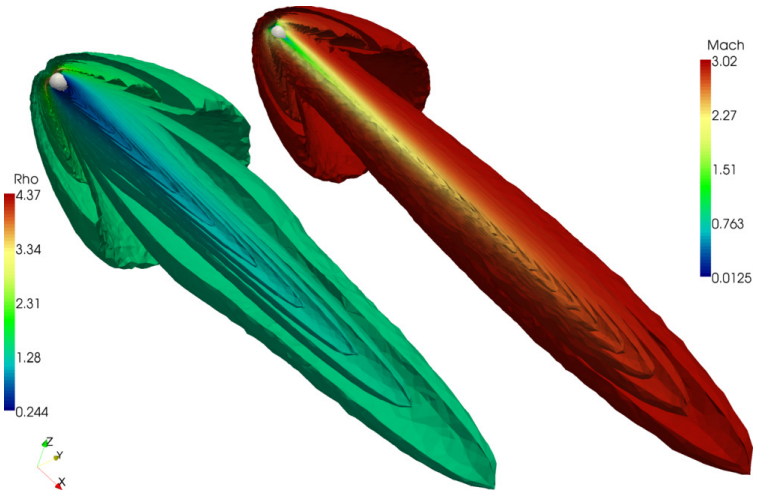


Figure 18.2: Example application of adaptive computation of 3D compressible flow around a sphere.

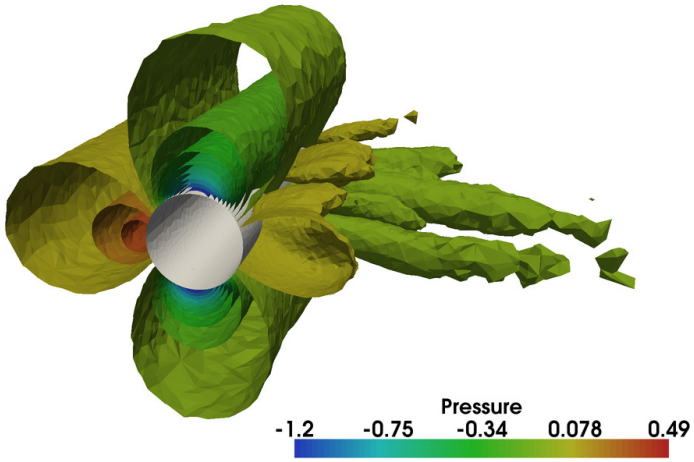


Figure 18.3: Example application of 3D turbulent incompressible flow around a cylinder with parallel adaptive computation.

18.2 Unified continuum modeling

We define, following classical continuum mechanics (Gurtin, 1981), a unified continuum model in a fixed Euler coordinate system consisting of:

- conservation of mass,
- conservation of momentum,
- conservation of energy,
- phase convection equation,
- constitutive equations for stress as data,

where the stress is the Cauchy (laboratory) stress and the phase is an indicator function used to determine which constitutive equation and material parameters to use. Note that in this continuum description the coordinate system is fixed (Euler), and a phase function (indicator) is convected according to the phase convection equation. The mesh is moved with the continuum velocity in the case of a solid phase to eliminate diffusion of the phase interface. We elaborate on this below in Section 18.3.2.

We define two variants of this model, incompressible and compressible, where a future aim is to construct a unified incompressible/compressible model and solver. We focus here the presentation on the incompressible model.

We start with a model for conservation of mass and momentum, together with a convection equation for a phase function θ over a space-time domain $Q = \Omega \times [0, T]$ with Ω a bounded open domain in $\mathbb{R}^3$ with boundary Γ:

$$\frac{\partial \rho}{\partial t} + \frac{\partial}{\partial x_j}(u_j \rho) = 0, \qquad \text{(mass conservation)} \tag{18.1}$$

$$\frac{\partial m_i}{\partial t} + \frac{\partial}{\partial x_j}(u_j m_i) = \frac{\partial}{\partial x_j}\sigma_{ij} + f_i, \qquad \text{(momentum conservation)} \tag{18.2}$$

$$\frac{\partial \theta}{\partial t} + \frac{\partial}{\partial x_j}(u_j \theta) = 0, \qquad \text{(phase convection equation)} \tag{18.3}$$

together with initial and boundary conditions, where ρ is density, $m_i = \rho u_i$ is momentum and u_i is velocity. If we make the assumption that the continuum is incompressible; that is, $0 = D_t\rho = \frac{\partial}{\partial t}\rho + u_j \frac{\partial}{\partial x_j}\rho$, it follows that we may express the incompressible UC equations as

$$\rho\left(\frac{\partial u_i}{\partial t} + u_j \frac{\partial u_i}{\partial x_j}\right) = \frac{\partial}{\partial x_j}\sigma_{ij} + f_i, \tag{18.4}$$

$$\frac{\partial u_j}{\partial x_j} = 0, \tag{18.5}$$

$$\frac{\partial \theta}{\partial t} + u_j \frac{\partial \theta}{\partial x_j} = 0. \tag{18.6}$$

We note that to model a fluid–structure interaction problem, we can simply write the total stress σ as a linear combination of fluid and solid stress using the θ phase function: $\sigma = \theta\sigma^f + (1-\theta)\sigma^s$. We can also decompose the stress into a pressure $p\delta_{ij}$ and deviatoric part: $\sigma_{ij} = \sigma^D_{ij} - p\delta_{ij}$.

The UC modeling framework is simple and compact, close to the formulation of the original conservation equations, and does not require mappings between different coordinate systems. This allows simple manipulation and processing for error estimation and implementation.

One key design choice of UC modeling is to define the Cauchy stress σ as data, which means the conservation equations are fixed regardless of the choice of constitutive equation. This gives a generality in method and software design, where a modification of the constitutive equation impacts the formulation and implementation of the constitutive equation, but not the formulation and implementation of the conservation equations.

In Unicorn we have currently implemented two constitutive laws, see Hoffman et al. (2011) for details:

$$\sigma_{ij}^f = 2\mu_f \epsilon(u)_{ij} \qquad \text{(Newtonian fluid)} \tag{18.7}$$

$$\frac{\partial \sigma_{ij}^s}{\partial t} = 2\mu_s \epsilon(u)_{ij} + \frac{\partial u_i}{\partial x_k}\sigma_{kj}^s + \sigma_{ik}^s \frac{\partial u_k}{\partial x_j} \qquad \text{(incompressible Neo-Hookean solid)} \tag{18.8}$$

18.3 Space-time general Galerkin discretization

Adaptive G2 methods (also referred to as Adaptive DNS/LES) have been used in a number of turbulent flow computations to a very low computational cost (Hoffman, 2005; Hoffman and Johnson, 2006; Hoffman, 2006; Hoffman and Johnson, 2007; Hoffman, 2009; Hoffman and Jansson, 2009; de Abreu et al., 2010), where convergence is obtained in output quantities such as drag, lift and pressure coefficients and Strouhal numbers, using orders of magnitude fewer mesh points than with standard LES methods based on *ad hoc* refined computational meshes.

18.3.1 Standard Galerkin

We begin by formulating the standard cG(1)cG(1) FEM (Eriksson et al., 1996) with continuous piecewise linear solution in time and space for (18.9). We let $w = (u, p, \theta)$ denote the exact solution, $W = (U, P, \Theta)$ the discrete solution, $v = (v^u, v^p, v^\theta)$ the test function and $R(W) = (R_u(W), R_p(W), R_\theta(W))$ the residual. The residual is defined by

$$\begin{aligned} R_u(W) &= \rho\left(\frac{\partial U_i}{\partial t} + U_j\frac{\partial U_i}{\partial x_j}\right) - \frac{\partial}{\partial x_j}(\Sigma_{ij}^D - P\delta_{ij}) - f_i, \\ R_p(W) &= \frac{\partial U_j}{\partial x_j}, \\ R_\theta(W) &= \frac{\partial \Theta}{\partial t} + U_j\frac{\partial \Theta}{\partial x_j}, \end{aligned} \tag{18.9}$$

where Σ denotes a discrete piecewise constant stress.

To compute the solution, we enforce the Galerkin orthogonality

$$\langle R(W), v\rangle = 0 \tag{18.10}$$

for all functions v in the test space $\hat{V}_{hk}$ consisting of piecewise linear continuous functions in space and piecewise constant discontinuous functions in time. Here $\langle\cdot,\cdot\rangle$ denotes the L^2-inner product in space and time.

This standard finite element formulation is unstable for convection-dominated problems and also suffers from instabilities as a result of equal order elements for the pressure and velocity. We therefore

add streamline–diffusion stabilization as described below.

The cG(1)cG(1) formulation with trapezoidal quadrature in time is equivalent to Crank–Nicolson time-stepping with piecewise linear elements in space. This has the advantage of being a very simple, standard, and familiar discrete formulation.

18.3.2 Local ALE

If the phase function Θ has different values on the same cell, it would lead to an undesirable diffusion of the phase interface. By introducing a moving space-time finite element space and mesh, oriented along the characteristics of the convection of the phase interface (Eriksson et al., 1996, section concerning "The characteristic Galerkin method"), we can define the phase interface at cell facets, allowing the interface to stay discontinuous.

We thus define a local ALE coordinate map as part of the discretization on each space-time slab, where it is used to introduce a mesh velocity. Note that we still compute with global Euler coordinates, but with a moving mesh.

To be able to define and compensate for an arbitrary mesh velocity β_h, we define a local coordinate map ϕ on each space-time slab:

$$\begin{aligned} \frac{\partial}{\partial t}\phi(t,\bar{x}) &= \beta_h(t,\bar{x}), \\ (x,t) &= \phi(\bar{x},t). \end{aligned} \tag{18.11}$$

Application of the chain rule gives the relation

$$\frac{\partial}{\partial t}U(x,t) + U(x,t)\cdot\nabla U(x,t) = \frac{\partial}{\partial t}\bar{U}(\bar{x},t) + (\bar{U}(\bar{x},t) - \beta_h))\cdot\nabla\bar{U}(\bar{x},t). \tag{18.12}$$

Choosing $\beta_h = U$ in the solid part of the mesh gives a trivial solution of the phase convection equation, and we can remove it from the system. The resulting discrete UC equations are then defined by the residuals

$$\begin{aligned} R_u(W) &= \rho\left(\frac{\partial U_i}{\partial t} + (U_j - \beta_j^h)\frac{\partial U_i}{\partial x_j}\right) - \frac{\partial}{\partial x_j}(\Sigma_{ij}^D - P\delta_{ij}) - f_i, \\ R_p(W) &= \frac{\partial U_j}{\partial x_j}. \end{aligned} \tag{18.13}$$

We thus choose the mesh velocity β_h to be the discrete material velocity U in the structure part of the mesh (vertices touching structure cells) and in the rest of the mesh we use mesh smoothing to determine β_h to maximize the mesh quality. Alternatively, one may use local mesh modification operations (refinement, coarsening, swapping) on the mesh to maintain the quality (Compère et al., 2009).

18.3.3 Weighted least–squares stabilization

The standard FEM formulation is unstable. We therefore consider a weighted standard least–squares method of the form $\langle R(W), v + \delta R(v)\rangle = 0$ for all $v \in \hat{V}_{hk}$ (see Eriksson et al. (1996); Hoffman and Johnson (2007)) with $\delta > 0$ a stabilization parameter. We make simplifications by dropping the stabilization terms including the time derivative and Σ^D. Although not fully consistent, this avoids unnecessary smearing of shear layers. For the UC model, the stabilized method thus takes the

following form:

$$\langle R^u(W), v^u \rangle = \langle \rho(\frac{\partial U_i}{\partial t} + U_j \frac{\partial U_i}{\partial x_j}) - f_i, v_i^u \rangle + \langle \Sigma_{ij}^D - P\delta_{ij}, \frac{\partial}{\partial x_j} v_i^u \rangle + SD^u(W, v^u) = 0, \tag{18.14}$$

$$\langle R^p(W), v^p \rangle = \langle \frac{\partial U_j}{\partial x_j}, v^p \rangle + SD^p(W, v^p) = 0, \tag{18.15}$$

for all $v \in \hat{V}_{hk}$, where

$$SD^u(W, v^u) = \delta_1 \langle U_j \frac{\partial U_i}{\partial x_j} + \frac{\partial P\delta_{ij}}{\partial x_i}, U_j \frac{\partial v_i^u}{\partial x_j} \rangle + \delta_2 \langle \frac{\partial U_j}{\partial x_j}, \frac{\partial v_j^u}{\partial x_j} \rangle, \tag{18.16}$$

$$SD^p(W, v^p) = \delta_1 \langle \frac{\partial P\delta_{ij}}{\partial x_i} + U_j \frac{\partial U_i}{\partial x_j}, \frac{\partial v^p}{\partial x_i} \rangle. \tag{18.17}$$

18.4 Implementation

We here present an overview of the design of Unicorn. The Unicorn solver class `UCSolver` ties together the technology in the Unicorn library with other parts of FEniCS to expose an interface (see listing 18.6) for simulating applications in continuum mechanics. The main part of the solver implementation is the weak forms for the G2 discretization of the UC model (see Figure 18.4), together with forms for the stress and residuals for the error estimation. Coefficients from the application are connected to the form, and then time-stepping is carried out by the class `TimeDependentPDE`. Certain coefficients, such as the δ stabilization coefficients are also computed as part of the solver (not as forms). The solver computes one iteration of the adaptive algorithm (primal solve, dual solve and mesh refinement), where the adaptive loop is implemented by iteratively running the solver for a sequence of meshes.

The `UCSolver` implementation is parallelized for distributed memory architectures using MPI, and we can show strong scaling for hundreds of cores on several platforms (see Figure 18.5). The entire adaptive algorithm is parallel, including Rivara mesh refinement and *a priori* predictive load balancing. Unicorn can efficiently simulate massively parallel applications for turbulent incompressible flow (Jansson et al., 2010; Jansson, 2011). An example of a parallel adaptive cylinder simulation is presented in Figure 18.3.

A compressible variant of the `UCSolver` exists as the `CNSSolver` for adaptive G2 for compressible Euler flow. The general method and algorithm is very close to that of the `UCSolver`, aside from the incompressibility. The long-term goal is a unification of the incompressible/compressible formulations as well. We refer to Nazarov (2009) for implementation details of the compressible `CNSSolver`. See Figure 18.2 for an example plot of compressible flow around a sphere.

18.4.1 Unicorn classes/interfaces

Key concepts are abstracted in the following classes/interfaces:

`TimeDependentPDE`: time-stepping
In each time-step a nonlinear algebraic system is solved by fixed-point iteration.

`ErrorEstimate`: adaptive error control
Adaptivity is based on computing local *error indicators* of the form $\eta_K = \|hR(U)\|_T \|DZ\|_T$, where Z is the so-called dual solution.

`SpaceTimeFunction`: space-time coefficient
Storage and evaluation of a space-time function/coefficient.

Figure 18.4: Source code for the bilinear and linear forms for a Newton iteration (with approximate Jacobian) for solving the incompressible UC model.

UFL code

```
...

v = TestFunction(V)
U1 = TrialFunction(V) # Unknown velocity U
UP = TrialFunction(V) # U from previous iteration
U0 = Function(V)      # U from previous timestep
P = Function(VS)      # Pressure

UC = 0.5*(U0 + UP)

def epsilon(u):
    return sym(grad(u))

def S(u, P):
    return P*Identity(d) - nu*grad(u)

def f(u, v):
    return -inner(grad(u)*u, v) + \
        inner(S(u, P), grad(v)) + \
        -d1*inner(grad(u)*u + grad(P), grad(v)*u) + \
        -d2*inner(div(u), div(v)) + \
        dot(fsource, v)

def dfdu(u, k, v):
    return -inner(grad(u)*UC, v) + \
        -inner(nu*grad(u), grad(v)) + \
        -d1*inner(grad(u)*UC, grad(v)*UC) + \
        -d2*inner(div(u), div(v))

# cG(1) in time
def F(u, u0, k, v):
    uc = 0.5 * (u + u0)
    return (-dot(u, v) + dot(u0, v) + k*f(u, v))

def dFdu(u, u0, k, v):
    uc = 0.5 * u
    return (-dot(u, v) + k*dfdu(uc, k, v))

a = (dFdu(U1, U0, k, v)) * dx
L = (dFdu(UP, U0, k, v) - F(UP, U0, k, v)) * dx
```

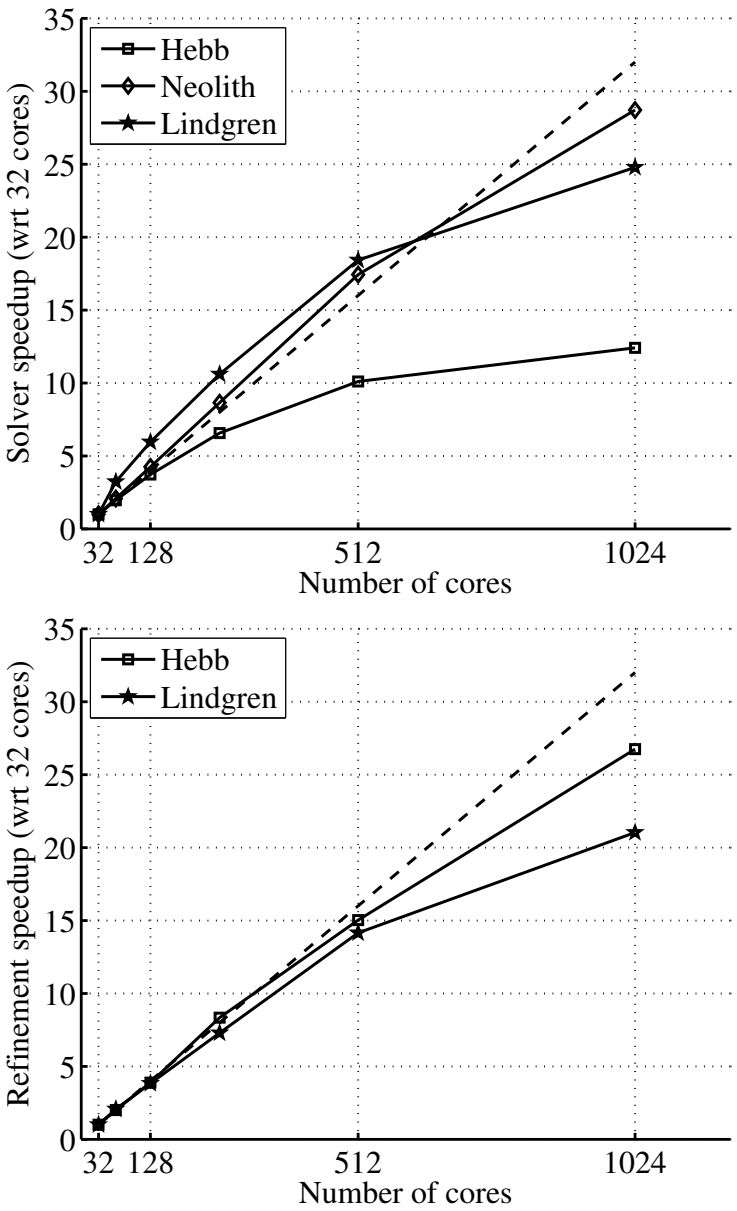


Figure 18.5: Strong scaling results for mesh refinement and entire solver on several different architectures: *Lindgren* (Cray XT6m), *Hebb* (BlueGene/L) and *Neolith* (regular Linux cluster with InfiniBand). The dashed lines refer to ideal speedup.

Figure 18.6: C++ class interface for the Unicorn class `UCSolver`.

C++ code

```
class UCSolver :
  public TimeDependentPDE, public MeshAdaptInterface
{
public:
  /// Constructor: give boundary conditions,
  /// coefficients
  UCSolver(Function& U, Function& U0,
           Function** bisect, Mesh& mesh,
           Array <BoundaryCondition*>& bc_mom,
           Array <BoundaryCondition*>& bc_con,
           Function** f, real T, real nu,
           real mu, real rho_f, real rho_s,
           real u_bar, TimeDependent& t,
           PDEData* pdedata);

  /// Prescribe mesh size for MeshAdaptInterface
  virtual void updateSizeField();

  /// Allocate/deallocate PDE data for dynamic mesh
  /// adaptivity
  virtual void allocateAndComputeData();
  virtual void deallocateData();

  /// Compute mesh vertex coordinates and velocity
  void computeX();
  void computeW();

  /// Compute density, pressure, stress
  void computeRho();
  void computeP();
  void computeStress();

  /// Compute initial theta
  void computeTheta0();

  /// From TimeDependentPDE: time-stepping control
  void shift();
  bool update(real t, bool end);
  void preparestep();
  void prepareiteration();

  /// Assemble time step residual (L) right-hand
  /// side of Newton
  void rhs(const Vector& x, Vector& dotx, real T);

  /// Compute initial value
  void u0(Vector& x);

  /// Save solution/output quantities
  void save(Function& U, real t);

  /// Compute least-squares stabilization parameters
  /// (delta)
  void computeStabilization(Mesh& mesh, Function& w,
                            real nu, real k, real t,
                            Vector& d1vector,
                            Vector& d2vector);

  /// Deform/move mesh
  void deform(Mesh& mesh, Function& W, Function& W0);

  /// Smooth/optimize quality of all or part of the
  /// mesh
  void smoothMesh(bool bAdaptive);
}
```

C++ code

```
/// Represent and solve time dependent PDE.
class TimeDependentPDE
{
  /// Public interface
public:
  TimeDependentPDE(
   // Computational mesh
   Mesh& mesh,
   // Bilinear form for Jacobian approx.
   Form& a,
   // Linear form for time-step residual
   Form& L,
   // List of boundary conditions
   Array <BoundaryCondition*>& bcs,
   // End time
   real T);

  /// Solve PDE
  virtual uint solve();

protected:
  /// Compute initial value
  virtual void u0(Vector& u);
  /// Called before each time step
  virtual void preparestep();
  /// Called before each fixed-point iteration
  virtual void prepareiteration();
  /// Return the bilinear form a
  Form& a();
  /// Return the linear form L
  Form& L();
  /// Return the mesh
  Mesh& mesh();
};
```

Figure 18.7: C++ class interface for `TimeDependentPDE`.

`SlipBC`: friction boundary condition
Efficient computation of turbulent flow in Unicorn is based on modeling of turbulent boundary layers by a friction model, where the slip boundary condition $u \cdot n = 0$ is implemented strongly as part of the algebraic system.

`ElasticSmoother`: elastic mesh smoothing/optimization
Optimization of cell quality according to an elastic analogy.

`MeshAdaptInterface`: mesh adaptation interface
Abstraction of the interface to the MAdLib package for mesh adaptation using local mesh operations.

18.4.2 `TimeDependentPDE`

We consider general time-dependent equations of the type $\frac{\partial}{\partial t}u + A(u) = 0$, where A denotes a possibly nonlinear differential operator in space. We want to define a class (data structures and algorithms) abstracting the time-stepping of the G2 method. The equation is given as input and the time-stepping should be generated automatically. We do this for the cG(1)cG(1) method by applying a simplified Newton's method. This is encapsulated in a C++ class interface in Figure 18.7 called `TimeDependentPDE`.

The skeleton of the time-stepping with fixed-point iteration is implemented in listing 18.8.

We use a block-diagonal quasi-Newton method, where we start by formulating the full Newton method and then drop terms off the diagonal blocks. We also use the constitutive law as an identity to express Σ in terms of U, allowing larger time steps than would be possibly otherwise by iterating between Σ and U. See Jansson (2009); Hoffman et al. (2011) for more details and discussion about the efficiency of the fixed-point iteration and its implementation.

18.4.3 `ErrorEstimate`

The duality-based adaptive error control algorithm requires the following components:

Residual computation We compute the mean-value in each cell of the residual $R(U)$ by an L^2-projection into the space of piecewise constants.

Dual solution We compute the solution of the dual problem using the same technology as the primal problem. The dual problem is solved backward in time, but using the time coordinate transform $s = T - t$ we can use the standard `TimeDependentPDE` interface.

Space-time function storage/evaluation We compute error indicators while solving the dual problem as space-time integrals over cells: $\eta_T = \langle R(U), \frac{\partial}{\partial x} Z \rangle$, where we need to evaluate both the primal solution U and the dual solution Z. In addition, U is a coefficient in the dual equation. This requires storage and evaluation of a space-time function, which is encapsulated in the `SpaceTimeFunction` class.

Mesh adaptation After the computation of the error indicators, we select the largest p% of the indicators for refinement. The refinement is then performed by recursive Rivara cell bisection. Alternatively, one may use MAdLib (Compère et al., 2009) for more general mesh adaptation based on edge split, collapse and swap operations.

Using these components, we can construct an adaptive algorithm. The adaptive algorithm is encapsulated in the C++ class interface in Figure 18.9 which we call `ErrorEstimate`.

18.4.4 `SpaceTimeFunction`

The error estimation algorithm requires, as part of solving the dual problem, the evaluation of space-time coefficients appearing in the definition of the dual problem. In particular, we must evaluate the primal solution U at time $s = T - t$. This requires storage and evaluation of a space-time function, which is encapsulated in the `SpaceTimeFunction` class (see listing 18.10).

The space-time functionality is implemented as a list of space functions at regular sample times, where evaluation is piecewise linear interpolation in time of the degrees of freedom.

18.4.5 `SlipBC`

For high Reynolds number problems such as car aerodynamics or airplane flight, it is not possible to resolve the turbulent boundary layer. One possibility is then to model turbulent boundary layers by a friction model:

$$u \cdot n = 0, \tag{18.18}$$

$$\beta u \cdot \tau_k + (\sigma n) \cdot \tau_k = 0, k = 1, 2. \tag{18.19}$$

We implement the normal component condition (slip) boundary condition strongly. By "strongly" we here mean an implementation of the boundary condition after assembling the left-hand side matrix

C++ code

```
void TimeDependentPDE::solve()
{
  // Time-stepping
  while (t < T)
  {
    U = U0;
    preparestep();
    step();
  }
}

void TimeDependentPDE::step()
{
  // Fixed-point iteration
  for(int iter = 0; iter < maxiter; iter++)
  {
    prepareiteration();
    step_residual = iter();

    if (step_residual < tol)
    {
      // Iteration converged
      break;
    }
  }
}

void TimeDependentPDE::iter()
{
  // Compute one fixed-point iteration
  assemble(J, a());
  assemble(b, L());
  for (uint i = 0; i < bc().size(); i++)
    bc()[i]->apply(J, b, a());
  solve(J, x, b);

  // Compute residual for the time-step/fixed-point
  // equation
  J.mult(x, residual);
  residual -= b;

  return residual.norm(linf);
}
```

Figure 18.8: Skeleton implementation in Unicorn of time-stepping with fixed-point iteration.

Figure 18.9: C++ class interface for `ErrorEstimate`.

C++ code

```
/// Estimate error as local error indicators based
/// on duality
class ErrorEstimate
{
public:

  /// Constructor (give components of UC residual
  /// and dual solution)
  ErrorEstimate(Mesh& mesh,
    Form* Lres_1,
    Form* Lres_2,
    Form* Lres_3,
    Form* LDphi_1,
    Form* LDphi_2,
    Form* LDphi_3);

  // Compute error (norm estimate)
  void ComputeError(real& error);

  // Compute error indicator
  void ComputeErrorIndicator(real t, real k,
                             real T);

  // Compute largest indicators
  void ComputeLargestIndicators(
    std::vector<int>& cells,
    real percentage);

  // Refine based on indicators
  void AdaptiveRefinement(real percentage);
}
```

Figure 18.10: C++ class interface for `SpaceTimeFunction`.

C++ code

```
/// Representation of space-time function (storage
/// and evaluation)
class SpaceTimeFunction
{
public:

  /// Create space-time function
  SpaceTimeFunction(Mesh& mesh, Function& Ut);

  /// Evaluate function at time t, giving result in
  /// Ut
  void eval(real t);

  // Add a space function at time t
  void addPoint(std::string Uname, real t);

  /// Return mesh associated with function
  Mesh& mesh();

  /// Return interpolant function
  Function& evaluant();
}
```

Python code

```
V = VectorFunctionSpace(mesh, "Lagrange", 1)
VT = TensorFunctionSpace(mesh, "Lagrange", 1)

v = TestFunction(VV)
F = Function(VV)
U = Function(V)

def f(U, F, v):
    return -inner(F*grad(U), v) - inner(grad(U).T*F, v)

# icv is inverse cell volume
L = (icv*(dot(F0m, vm) + k*f(U, Fm, vm)))*dx
```

Figure 18.11: Source code for the form representing one time step for the deformation gradient (F) evolution in the elastic smoother variant of the UC model.

and the right-hand side vector in the algebraic system, whereas the tangential components (friction) are implemented "weakly" by adding boundary integrals in the variational formulation. The row of the matrix and load vector corresponding to a degree of freedom is found and replaced by a new row according to the boundary condition.

The idea is as follows: Initially, the test function v is expressed in the Cartesian standard basis (e_1, e_2, e_3). Now, the test function is mapped locally to normal-tangent coordinates with the basis (n, τ_1, τ_2), where $n = (n_1, n_2, n_3)$ is the normal, and $\tau_1 = (\tau_{11}, \tau_{12}, \tau_{13})$, $\tau_2 = (\tau_{21}, \tau_{22}, \tau_{23})$ are tangents to each node on the boundary. This allows us to let the normal direction be constrained and the tangent directions be free:

$$v = (v \cdot n)n + (v \cdot \tau_1)\tau_1 + (v \cdot \tau_2)\tau_2. \tag{18.20}$$

For the matrix and vector this means that the rows corresponding to the boundary need to be multiplied with n, τ_1, τ_2, respectively, and then the normal component of the velocity should be set to zero.

This concept is encapsulated in the class `SlipBC` which is a subclass of `dolfin::BoundaryCondition` for representing strong boundary conditions. For more details about the implementation of slip boundary conditions, we refer to Nazarov (2011).

18.4.6 `ElasticSmoother`

To maintain a discontinuous phase interface in the UC model, we define the mesh velocity β_h as the discrete velocity U in the solid phase (specifically on the interface). The mesh velocity in the fluid can be chosen more arbitrarily, but has to satisfy mesh quality and size criteria. We construct a cell quality optimization/smoothing method based on a pure elastic variant of the UC (see the form in Figure 18.11). We define the following requirements for the mesh velocity β_h:

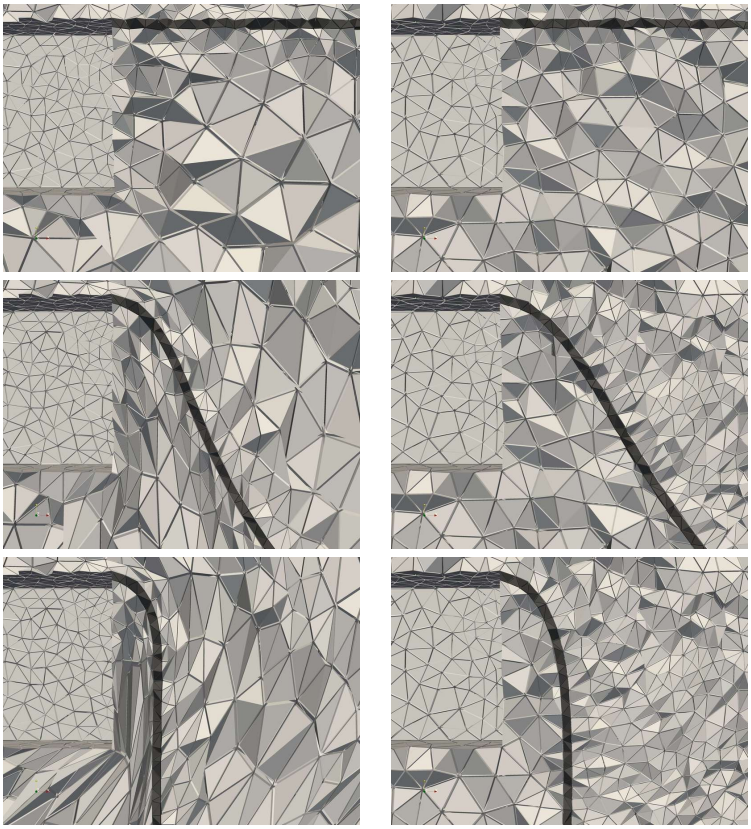
1. $\beta_h = U$ in the solid phase part of the mesh.

2. Bounded mesh quality Q defined by

$$Q = \frac{d\|F\|_F^2}{\det(F)^{\frac{2}{d}}},$$

   where d is the spatial dimension, in the fluid part of the mesh. Preferably the mesh smoothing should improve Q if possible.

3. Maintain mesh size $h(x)$ close to a desired $\hat{h}(x)$ given by *a posteriori* error estimation in an adaptive algorithm.



(a) (b)

Figure 18.12: Robustness test with (a) elastic smoothing and (b) mesh adaptation. Note the badly shaped cells squeezed between the cube and flag.

Figure 18.13: C++ class interface for `ElasticSmoother`.

C++ code

```
/// Optimize cell quality according to elastic
/// variant of UC model
class ElasticSmoother
{
public:

  ElasticSmoother(Mesh& mesh);

  /// Smooth smoothed_cells giving mesh velocity W
  /// over time step k with h0 the prescribed cell
  /// size
  void smooth(MeshFunction<bool>& smoothed_cells,
              MeshFunction<bool>& masked_cells,
              MeshFunction<real>& h0,
              Function& W, real k);

  /// Extract submesh (for smoothing only marked cells)
  static void
  submesh(Mesh& mesh, Mesh& sub,
    MeshFunction<bool>& smoothed_cells,
    MeshFunction<int>& old2new_vertex,
    MeshFunction<int>& old2new_cell);

}
```

Mesh smoothing is handled in Unicorn by an elastic model using the constitutive law $\sigma = \mu(I - (FF^\top)^{-1})$ where we recall F as the deformation gradient. We use the update law: $\frac{\partial}{\partial t}F^{-1} = -F^{-1}\nabla u$ where we thus need an initial condition for F. We set the initial condition $F_0 = \bar{F}$ where $\bar{F}$ is the deformation gradient with regard to a scaled equilateral reference cell, representing the optimal shape with quality $Q = 1$.

Solving the elastic model can thus be seen as optimizing for the highest global quality Q in the mesh. We also introduce a weight on the Young's modulus μ for cells with low quality, penalizing high average, but low local quality over mediocre global quality. We refer to the source code for more details.

Unicorn provides the `ElasticSmoother` class (see listing 18.13, which can be used to smooth/optimize for quality in all or part of the mesh.

We perform a robustness test of the elastic smoothing and the mesh adaptivity shown in 18.12 where we use the same geometry as the turbulent 3D flag problem, but define a zero inflow velocity and instead add a gravity body force to the flag to create a very large deformation with the flag pointing straight down. Both the elastic smoothing and the mesh adaptivity compute solutions, but as expected, the elastic mesh smoothing eventually cannot control the cell quality; there does not exist a mesh motion which can handle large rigid body rotations while bounding the cell quality.

18.4.7 `MeshAdaptInterface`

A critical component in the adaptive algorithm as described above is *mesh adaptivity*, which we define as constructing a mesh satisfying a given mesh size function $h(x)$.

We start by presenting the Rivara recursive bisection algorithm (Rivara, 1992) as a basic choice for mesh adaptivity (currently the only available choice for parallel mesh adaptivity), but which can only refine and not coarsen. Then the more general MAdLib is presented, which enables full mesh adaptation to the prescribed $h(x)$ through local mesh operations: edge split, edge collapse and edge swap.

Rivara recursive bisection The Rivara algorithm bisects (splits) the longest edge of a cell, thus replacing the cell with two new cells, and uses recursive bisection to eliminate non-conforming cells with hanging nodes. The same algorithm holds in both 2D/3D (triangles/tetrahedra). In 2D, it can be shown

Algorithm 8 The Rivara recursive bisection algorithm

```
procedure BISECT(T)
    Split longest edge e
    while T_i(e) is non-conforming do
        BISECT(T_i)
    end while
end procedure
```

(Rivara, 1992) that the algorithm terminates in a finite number of steps, and that the minimum angle of the refined mesh is at least half the minimum angle of the starting mesh. In practice, the algorithm produces excellent quality refined meshes both in 2D and 3D.

Local mesh operations: MAdLib MAdLib incorporates an algorithm and implementation of mesh adaptation in which a small set of local mesh modification operators are defined such as edge split, edge collapse and edge swap (see Figure 18.14 for an illustration of the edge swap operator). A mesh adaptation algorithm is defined which uses this set of local operators in a control loop to satisfy a

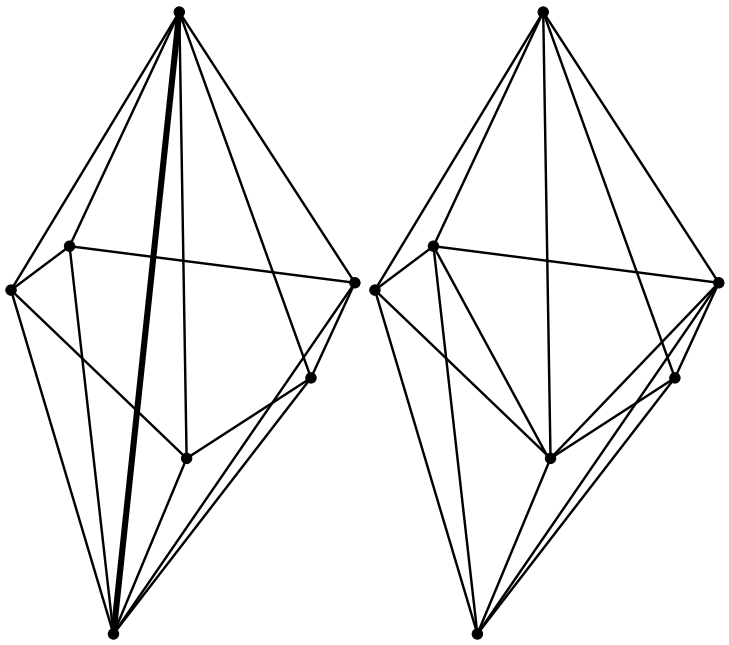


Figure 18.14: Edge swap operation: (top) initial cavity with swap edge highlighted and (bottom) possible configuration after the swap.

prescribed size field $h(x)$ and quality tolerance. Edge swapping is the key operator for improving the quality of cells, for example around a vertex with a large number of connected edges.

In the formulation of finite element methods, it is typically assumed that the cell size of a computational mesh can be freely modified to satisfy a desired size field $h(x)$ or to allow mesh motion. In state-of-the-art finite element software implementations, this is seldom the case (Bangerth et al., 2007; COMSOL, 2009).

The mesh adaptation algorithm in MAdLib gives the freedom to adapt to a specified size field using local mesh operations. The implementation is published as free/open-source software.

Unicorn provides the `MeshAdaptInterface` class (see listing 18.15), where one can subclass and implement virtual functions to control the mesh adaptation using MAdLib.

18.5 Solving continuum mechanics problems

In this section, we present some examples computed using Unicorn. The first example is a fluid–structure interaction problem without adaptivity, where we cover modeling of geometry and subdomains, coefficients (functions used in the form), parameters and specification of the main program (interface to running the solver). Next, we present an example of solving a turbulent pure fluid problem with adaptivity, where we cover modeling of data for the dual problem, the adaptive loop, and specifying slip/friction boundary conditions for modeling turbulent boundary layers.

We present here illustrative pieces of the problem implementations. For the full details, see the Unicorn source code distribution.

18.5.1 Fluid–structure interaction

We here give a use case of solving a fluid–structure continuum mechanics problem, where the user specifies data for modeling the problem, and illustrates interfaces and expected outcomes. The problem is the FLUSTRUK-A variant 3 benchmark given in (Hron and Turek, 2005) and used as verification of Unicorn in (Hoffman et al., 2011) and also in Chapter 28. The velocity field and phase

C++ code

```
/// Interface to MAdLib for mesh adaptation using
/// local operations Subclass and implement the
/// virtual functions
class MeshAdaptInterface
{
public:
  MeshAdaptInterface(Mesh *);

protected:
  /// Start mesh adaptation algorithm
  void adaptMesh();

  /// Give cell size field
  virtual void updateSizeField() = 0;

  /// Allocate and deallocate solver data
  virtual void deallocateData() = 0;
  virtual void allocateAndComputeData() = 0;

  /// Constrain entities not to be adapted
  void constrainExternalBoundaries();
  void constrainInternalBoundaries();

  /// Add functions to be automatically interpolated
  void addFunction(string name, Function** f);
  void clearFunctions();
};
```

Figure 18.15: C++ class interface for `MeshAdaptInterface`.

function θ are plotted in Figure 18.16 for times t = 1 and t = 5.5. The full example can be found in the Unicorn distribution under `unicorn-0.1.3-hpc/ucsolver/fsi/cylinder2D/`.

We divide the use case into four parts:

Geometry and subdomains
: The user specifies possible geometrical parameters and defines subdomains. We note that for complex geometries the user may omit geometry information and specify subdomain markers as data files.

Coefficients
: Known coefficients (functions appearing in the form) such as a force function and boundary conditions are declared.

Parameters
: Simple parameters specifying material properties and numerical tolerances are given in a parameter file.

main program
: The user implements the main program and declares and passes data to to the solver.

These four parts are demonstrated by the code examples shown in Figures 18.18–18.21.

18.5.2 Adaptivity

We continue with a use case for adaptive solution of a pure fluid turbulent flow problem: flow around a 3D cube corresponding to the cube problem description given in Chapter 28. A snapshot of the

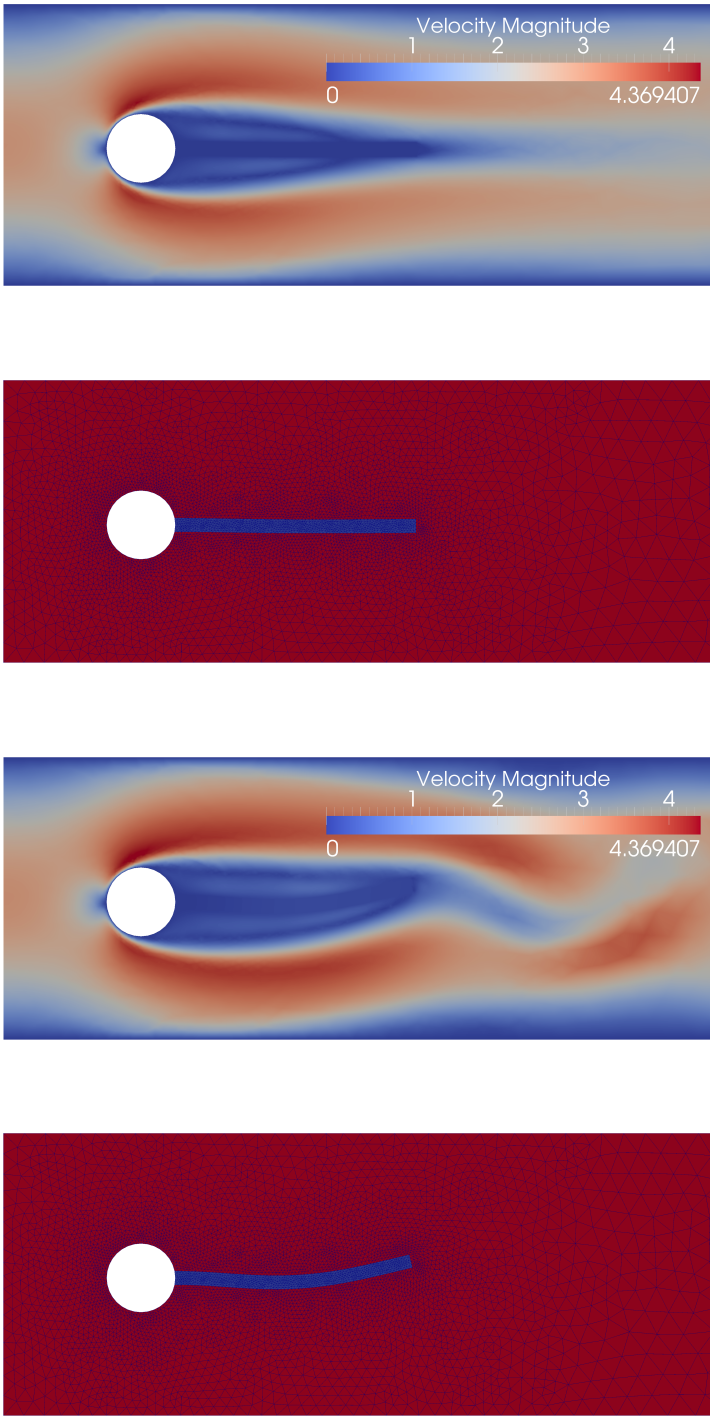


Figure 18.16: Velocity and phase function (with mesh) for times t = 1 and t = 5.5 for FSI benchmark and use case. Solution computed in parallel on a distributed mesh with 12 cores (5415 mesh points).

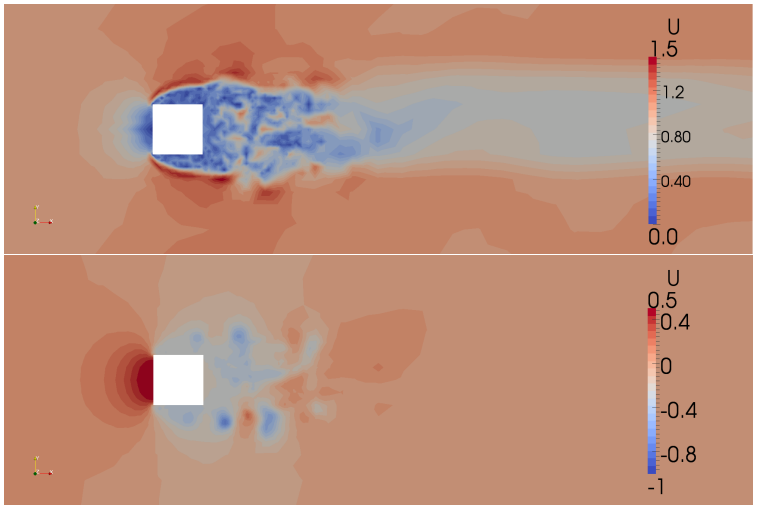


Figure 18.17: Flow around a cube: snapshots of velocity (upper) and pressure (lower) for the finest mesh.

solution can be seen in Figure 18.17. The full example can be found in the Unicorn distribution under `unicorn-0.1.3-hpc/ucsolver/icns/cube/`.

The implementation of the problem is very similar to the fluid–structure case (just with pure fluid data), but with 3 important additions:

Dual problem

To compute the error estimate required by the adaptive algorithm, we must solve a dual problem generated by the primal problem and an output quantity ψ. Since the dual problem is similar in form to the primal problem, we implement both as variants of the same solver.

In this case we are interested in computing drag, which gives ψ as a boundary condition for the dual problem:

Adaptive loop

We specify a `solve()` function which computes one iteration of the adaptive loop: solve primal problem, solve dual problem, compute error estimate and check if tolerance is satisfied and compute adapted mesh. This function is then called in an adaptive loop by calling the `unicorn_solve()` function.

C++ code

```
class Dual_BC_Momentum : public Function
{
public:
  Dual_BC_Momentum(Mesh& mesh) : Function(mesh) {}
  void eval(real* values, const real* x) const
  {
    values[0] = -1.0;
    values[1] = 0.0;
    values[2] = 0.0;
  }
};

solve()
{
  // Boundary condition for dual problem
  CubeBoundary cb;
  Dual_BC_Momentum dual_bcf_mom(mesh);
```

```
  DirichletBC dual_bc0(dual_bcf_mom, mesh, cb);

  Array <BoundaryCondition*> dual_bc_mom;
  dual_bc_mom.push_back(&dual_bc0);
  ...


  // Solve primal problem
  NSESolver primal_solver(mesh, node_normal, f, beta, aero_f, primal_bc_mom, primal_bc_con,
    chkp, w_limit, td, "primal");
  primal_solver.solve();

  // Solve dual problem (gives weight for error estimate)
  NSESolver dual_solver(mesh, node_normal, f, beta, aero_f, dual_bc_mom, dual_bc_con,
    chkp, w_limit, td, "dual");
  dual_solver.solve();
}
```

Slip boundary condition

For turbulent flow, we model the boundary layer as a friction boundary condition. We specify the normal component as a strong slip boundary condition used just as a regular Dirichlet boundary condition.

C++ code

```
NodeNormal node_normal(mesh);
SlipBoundary sb;

SlipBC slip_bc(mesh, sb, node_normal);

Array <BoundaryCondition*> primal_bc_mom;
primal_bc_mom.push_back(&slip_bc);
```

18.5.3 Unicorn-HPC installation and basic test

Unicorn-HPC is the high-performance computing branch of Unicorn, and is the actively developed branch. The version corresponding to the chapters in this book is "Unicorn 0.1.3-hpc". For simulations and plots referenced from previously published articles, we refer to that specific article for software details. The MAdLib mesh adaptation interface of Unicorn is currently only available in the no longer maintained serial branch of Unicorn, which we make available on the Live DVD `unicorn_fenics-book.iso` available through either `http://www.csc.kth.se/ctl` or `http://fenicsproject.org`.

To verify the correct installation and functionality of Unicorn-HPC, follow the steps in the README file in the Unicorn-HPC distribution, under "Testing". The test represents the turbulent flow past a cube simulation described in Chapter 28 and above.

18.6 Acknowledgments

We acknowledge contributions to Unicorn, both software development as well as ideas and scientific support from: Mattias Aechtner, Peter Brune, Zilan Ciftci, Gäetan Compere, Claes Johnson, Ashraful Kadir, Kaspar Müller, Jeannette Spühler, Michael Stöckli and Rodrigo Vilela de Abreu.

The authors would like to acknowledge the financial support from the European Research Council, Swedish Foundation for Strategic Research, the Swedish Research Council and the Swedish Energy

C++ code

```
#include <dolfin.h>
#include <unicorn/UCSolver.h>

using namespace dolfin;
using namespace dolfin::unicorn;

real bmarg = 1.0e-6 + DOLFIN_EPS;

namespace Geo
{
  // Geometry details
  real box_L = 2.5;
  real box_H = 0.41;

  real xmin = 0.0; real xmax = box_L;
  real ymin = 0.0; real ymax = box_H;
}

// Sub domain for inflow
class InflowBoundary : public SubDomain
{
public:
  bool inside(const real* p, bool on_boundary) const
  {
    return on_boundary && (p[0] < Geo::xmax - bmarg ||
                           p[1] < Geo::ymin + bmarg ||
                           p[1] > Geo::ymax - bmarg);
  }
};

// Sub domain for outflow
class OutflowBoundary : public SubDomain
{
public:
  bool inside(const real* p, bool on_boundary) const
  {
    InflowBoundary ib;

    return on_boundary && !ib.inside(p, on_boundary);
  }
};
```

Figure 18.18: Part 1 of Unicorn solver FSI use case: geometry and subdomains.

Figure 18.19: Part 2 of Unicorn solver FSI use case: coefficients.

C++ code

```
// Force term
class ForceFunction : public Function
{
public:
  ForceFunction(Mesh& mesh, TimeDependent& td) :
    Function(mesh), td(td) {}
  void eval(real* values, const real* x) const
  {
    int d = cell().dim();

    for (int i = 0; i < d; i++)
    {
      values[i] = 0.0;
    }
  }

  TimeDependent& td;
};

// Boundary condition for momentum equation
class BC_Momentum : public Function
{
public:
  BC_Momentum(Mesh& mesh) :
    Function(mesh) {}
  void eval(real* values, const real* x) const
  {
    int d = cell().dim();

    for (int i = 0; i < d; i++)
    {
      values[i] = 0.0;
    }
    if (x[0] < (Geo::xmin + bmarg))
    {
      // Define a parabolic inflow profile
      real Um = 2.0;
      values[0] =
        1.5*Um*( (x[1]*(Geo::cont_H - x[1])) /
          ((0.5*Geo::cont_H)*(0.5*Geo::cont_H)) );
  }
};

// Initial condition for phase variable
class BisectionFunction : public Function
{
public:
  BisectionFunction(Mesh& mesh) : Function(mesh) {}
  void eval(real* values, const real* p) const
  {
    bool condition =
      (((flag_start_x - bmarg) < p[0] &&
         p[0] < (flag_end_x + bmarg)) &&
       ((flag_start_y - bmarg) < p[1] &&
         p[1] < (flag_end_y + bmarg)));

    if(condition)
      values[0] = 0.0;
    else
      values[0] = 1.0;
  }
};
```

C++ code

```
Parameters for Unicorn solver:

0.0       beta - friction coefficient
0.001     nu - viscosity
2.0e3     mu - solid stiffness
0.0e0     nu_s - solid damping
1.0e0     rho_s - solid density

10        n_samples - number of samples
1         adapt_iter - number of adaptive iterations
0.001     adapt_tol - adaptive tolerance
10        adapt_percentage - adaptive percentage
rivara    adapt_algorithm - adaptive refinement type

10.0       T - final time, primal solver
0.25      dual_T - final time, dual solver
8.0       Ubar - free stream velocity

gmres     krylov_method - krylov method
amg       krylov_pc - preconditioner
```

Figure 18.20: Part 3 of FSI use case: parameter file

C++ code

```
void solve(Mesh& mesh, Checkpoint& chkp,
  long& w_limit, timeval& s_time,
  Mesh* structure_mesh)
{
  InflowBoundary iboundary;
  BC_Momentum bcf_mom(mesh);
  ForceFunction f(mesh);

  DirichletBC bc_mom0(bcf_mom, mesh, iboundary);

  Array <BoundaryCondition*> bc_mom;
  bc_mom.push_back(&bc_mom0);

  Array<Function*> aero_f;

  ...

 UCSolver psolver(mesh, U, U0, f, f, phi, beta, p_bc_momentum,
                  p_bc_pressure, p_bc_density, &density,
                  solid_cells, solid_vertices, T, nu, mu,
                  nu_s, rho_s, ubar, td, ''primal'');
  psolver.solve();
}

int main()
{
  Mesh mesh;
  ...

  unicorn_init(argc, argv, mesh, chkp, w_limit, iter,
       structure_mesh);

  unicorn_solve(mesh, chkp, w_limit, s_time, iter, 0, 0,
                &solve, structure_mesh);
  return 0;
}
```

Figure 18.21: Part 4 of FSI use case: structure of main program, passing data to solver.

Agency. The simulations were performed on resources provided by the Swedish National Infrastructure for Computing (SNIC) at High Performance Computing Center North (HPC2N), National Supercomputer Centre in Sweden (NSC) and PDC – Center for High-Performance Computing.

19 Lessons learned in mixed language programming

By Johan Hake and Kent-Andre Mardal

This chapter describes decisions made and lessons learned in the implementation of the Python interface of DOLFIN. The chapter is quite technical, since we aim at giving the reader a thorough understanding of the implementation of the DOLFIN Python interface.

19.1 Background

Python has over the last decade become an established platform for scientific computing. Widely used scientific software such as, e.g., PETSc, Hypre, Trilinos, VTK, VMTK, GiNaC (Bauer et al., 2002) have all been equipped with Python interfaces. The FEniCS packages FErari, FIAT, FFC, UFL, Viper, as well as other packages such as SymPy (Čertík et al., 2009), SciPy (Jones et al., 2009) are pure Python packages. The DOLFIN library has both a C++ and a Python user-interface. Python makes application building on top of DOLFIN more user friendly, but the Python interface also introduces additional complexity and new problems. We assume that the reader has basic knowledge of both C++ and Python. A suitable textbook on Python for scientific computing is Langtangen (2008), which cover both Python and its C interface. SWIG, which is the software we use to wrap DOLFIN, is well documented and we refer to the user manual that can be found on its web page (SWIG). Finally, we refer to Langtangen and Mardal (2003) and Sala et al. (2008) for a description of how SWIG can be used to generate Python interfaces for other packages such as Diffpack and Trilinos.

19.2 Using SWIG

Python and C++ are two very different languages, while Python is user–friendly and flexible, C++ is very efficient. To combine the strengths of the two languages, it has become common to equip C++ (or FORTRAN/C) libraries with Python interfaces. Such interfaces must comply with the Python C-API. Writing such interfaces, often called wrapper code, is quite involved. Therefore, a number of wrapper code generators have been developed in the recent years, some examples are F2PY, SIP, SILOON, and SWIG. SWIG has been used to create the DOLFIN Python interface, and will therefore be the focus in this chapter. SWIG is a mature wrapper code generator that supports many languages and is extensively documented.

19.2.1 Basic SWIG

To get a basic understanding of SWIG, we consider an implementation of an array class. Let the array class be defined in `Array.h` as follows:

C++ code

```
#include <iostream>

class Array {
public:
  // Constructors and destructors
  Array(int n_=0);
  Array(int n_, double* a_);
  Array(const Array& a_);
  ~Array();

  // Operators
  Array& operator=(const Array& a_);
  const double& operator [] (int i) const;
  double& operator [] (int i);
  const Array& operator+= (const Array& b);

  // Methods
  int dim() const;
  double norm() const;

private:
  int n;
  double *a;

};

std::ostream & operator<< ( std::ostream& os, const Array& a);
```

A first attempt to make the Array accessible in Python using SWIG, is to write a SWIG interface file `Array_1.i`.

SWIG code

```
%module Array
%{
#include "Array.h"
%}
%include "Array.h"
```

Here, we specify the name of the Python module: `Array`; the code that should be inlined in the wrapper code directly (the declarations): `#include "Array.h"`; and the code SWIG should parse to create the wrapper code: `%include "Array.h"` (definitions). The following command shows how to run SWIG to produce the wrapper code:

Bash code

```
swig -python -c++ -I. -O Array_1.i
```

The command generates two files: `Array.py` and `Array_wrap.cxx`. The file `Array_wrap.cxx` contains C code that defines the Python interface of Array. After `Array_wrap.cxx` is compiled into a shared library, it can be imported into Python. The file `Array.py` is written in pure Python. It imports the shared library and also adds some functionality to the wrapped module. The reader should be able to recognize the Python class `Array` at the end of the `Array.py` file.

The following Distutils file (`setup.py`) executes the SWIG command above and compiles and links the source code and the generated wrapper code into a shared library.

Python code

```
import os
```

```
import numpy
from distutils.core import setup, Extension
swig_cmd ='swig -o Array_wrap.cxx -python -c++ -O -I. Array_1.i'
os.system(swig_cmd)
sources = ['Array.cpp','Array_wrap.cxx']
setup(name = 'Array',
      py_modules = ["Array"],
      ext_modules = [Extension('_' + 'Array', sources, \
                    include_dirs=['.', numpy.get_include() + "/numpy"])])
```

Build and install the module in the current working directory with the command:

Bash code

```
python setup.py install --install-lib=.
```

The Python proxy class resembles the C++ class in many ways. Simple methods such as `dim()` and `norm()` will be wrapped correctly to Python, since SWIG maps `int` and `double` arguments to the corresponding Python types through built-in typemaps. However, a number of issues appear:

1. the `operator[]` does not work;
2. the `operator+=` returns a new Python object (with different `id`);
3. printing does not use the `std::ostream & operator<<`;
4. the `Array(int n_, double* a_);` constructor is not working properly.

We see that a number of different problems arise even in such a simple example. Fortunately, these problems are fairly common, and general solutions can be implemented quite easily. In the following, we will go through each of the above issues. The example code with the solutions proposed in the following can be found in `Array_2.i`.

19.2.2 *The `operator[]`*

In C++, the subscripting `operator[]` is used to implement both set and get operators. It is possible to distinguish the set operator from the get operator using `const`, but this is not required. In Python, subscripting is performed with the two special methods: `__setitem__` and `__getitem__`. Since, the mapping between the Python operators (`__setitem__` and `__getitem__`) and the C++ operators `operator[]` may be ambiguous, SWIG currently ignores these operators. To implement the operators properly, also in future versions of SWIG, we ignore both version of the `operator[]` with

SWIG code

```
%ignore Array::operator[];
```

and extend the interface of the generated C++ code with the auxiliary `__setitem__` and `__getitem__` methods:

SWIG code

```
%extend Array {
double __getitem__(int i) {
  return (*self)[i];
}

void __setitem__(int i, double v) {
  (*self)[i] = v;
}
```

```
...
};
```

Note that all SWIG directives start with `"%"`. Furthermore, the access to the actual instance is provided by the `self` pointer, which in this case is a C++ pointer that points to an `Array` instance. The pointer is comparable to the `this` pointer in a C++ class, but only the public attributes are available.

19.2.3 `operator +=`

The second problem is related to SWIG and garbage collection in Python. Python features garbage collection, which means that a user should not be concerned with the destruction of objects. The mechanism is based on reference counting; that is, when no more references are pointing to an object, the object is destroyed. The SWIG generated Python module consists of a small Python layer that defines the interface to the underlying C++ object. An instance of a SWIG generated class therefore keeps a reference to the underlying C++ object. The default behavior is that the C++ object is destroyed together with the Python object. This behavior is not consistent with the `operator +=` returning a new object, which is illustrated by the generated segmentation fault in the following example (see `segfault_test.py`):

Python code

```
from Array import Array
def add(b):
    print "id(b):",id(b)
    b+=b
    print "id(b):",id(b)

a = Array(10)
print "id(a):",id(a)
add(a)
a+=a
```

This script produces the following output:

Python code

```
id(a): 3085535980
id(b): 3085535980
id(b): 3085536492
Segmentation fault
```

The script causes a segmentation fault because the underlying C++ object is destroyed after the call to `add()`. When the last `a+=a` is performed the underlying C++ object is already destroyed. This happens because the SWIG generated `__iadd__` method returns a new Python object. This is illustrated by the different values obtained from the `id()` function[1]. The last two calls to `id(b)` return different numbers, which means that a new Python object is returned by the SWIG generated `__iadd__` method. The second `b` object is local in the `add` function and is therefore deleted together with the underlying C++ object when `add` has finished.

The memory problem can be solved by extending the interface with an `_add` method and implementing our own `__iadd__` method in terms of `_add`, using the `%extend` directive:

SWIG code

```
%extend Array {
...
  void _add(const Array& a){
```

[1]The `id` function returns a unique integer identifying the object.

```
    (*self) += a;
  }

  %pythoncode %{
    def __iadd__(self,a):
      self._add(a)
      return self
  %}
...
};
```

The above script will now report the same `id` for all objects. No objects are created or deleted, and segmentation fault is avoided.

19.2.4 *std::ostream & operator<<*

In C++, shift operators such as `operator «` are typically used to implement I/O, while in Python the `__str__` method is used. Therefore, SWIG ignores the shift operator, as it is likely not to perform as intended. However, we can again use the `%extend` directive to make this operator available from Python by adding a `__str__` method.

SWIG code

```
%include <std_string.i>

%extend Array {
...
  std::string __str__() {
    std::ostringstream s;
    s << (*self);
    return s.str();
  }
};
```

This method uses the `operator<<` representation of the array to a `std::ostringstream` and then returns a `std::string` representation of the stream. Note that we need to include `std_string.i` in the `Array_2.i`. In Python, we can then call `print` on an instance of `Array`.

19.2.5 *The constructor: `Array(int n_, double* a_);`*

The fourth problem is related to pointer handling in C/C++ and SWIG. From the constructor signature alone, it is not clear whether `double* a_` points to a single value or to the first element of an array. Therefore, SWIG takes a conservative approach and handles pointers as pointers to single values. In our example `double* a_` points to the first element of an array of length `n`, and SWIG erroneously generates code for passing an `int` and a `double` to the method.

As a remedy, SWIG provides the `typemap` concept to enable mappings between C/C++ and Python types. The following code, explained in detail below, demonstrates how to map a NumPy array to the `(int n_, double* a_)` arguments in the constructor.

SWIG code

```
%typemap(in) (int n_, double* a_){
  if (!PyArray_Check($input)) {
    PyErr_SetString(PyExc_TypeError, "Not a NumPy array");
    return NULL; ;
  }
  PyArrayObject* pyarray = reinterpret_cast<PyArrayObject*>($input);
  if (!(PyArray_TYPE(pyarray) == NPY_DOUBLE)) {
```

```
    PyErr_SetString(PyExc_TypeError, "Not a NumPy array of doubles");
    return NULL; ;
  }
  $1 = PyArray_DIM(pyarray,0);
  $2 = static_cast<double*>(PyArray_DATA(pyarray));
}
```

The first line specifies that the typemap should be applied to the input `(in)` arguments of operators, functions, and methods with the `int n_,double* a_` arguments in the signature. The $ prefixed variables are used to map input and output variables in the typemap; that is, the variables $1 and $2 map to the first and second output C arguments of the typemap, `n_` and `a_`, while `$input` maps to the Python input.

In the next three lines, we check that the input Python object is a NumPy array, and raise an exception if not. Note that any Python C-API function that returns `NULL` tells the Python interpreter that an exception has occurred. Python will then raise an error, with the error message set by the `PyErr_SetString` function. Next, we cast the Python object pointer to a NumPy array pointer and check that the data type of the NumPy array is correct; that is, that it contains doubles. Then, we acquire the data from the NumPy array and assign the two input variables.

Overloading operators, functions and methods is not possible in Python. Instead, Python dynamically determines what code to call, a process which is called dynamic dispatch. To generate proper wrapper code, SWIG relies on `%typecheck` directives to resolve the overloading. A suitable type check for our example typemap is:

SWIG code

```
%typecheck(SWIG_TYPECHECK_DOUBLE_ARRAY) (int n_, double* a_) {
   $1 = PyArray_Check($input) ? 1 : 0;
 }
```

Here, `SWIG_TYPECHECK_DOUBLE_ARRAY` is a `typedef` for the priority number assigned for arrays of doubles. The type check should return 1 if the Python object `$input` has the correct type, and 0 otherwise.

19.3 SWIG and the DOLFIN Python interface

To make the DOLFIN Python interface more *Pythonic*, we have made a number of specializations, along the lines mentioned above, that we will now go through. But let us start with the overall picture. The interface files resides in the `dolfin/swig` directory, and are organized into *i)* global files, that apply to the entire DOLFIN library, and *ii)* kernel module files that apply to specific DOLFIN modules. The latter files are divided into `..._pre.i` and `..._post.i` files, which are applied before and after the inclusion of the header files of the particular kernel module, respectively. The kernel modules, as seen in `kernel_module.i`, mirror the directory structure of DOLFIN: `common`, `parameters`, `la`, `mesh` and so forth. The global interface files are all included in `dolfin.i`, the main SWIG interface file. The kernel module interface files are included, together with the C++ header files, in the automatically generated `kernel_modules.i` file.

The following sections deal with the main interface file of `dolfin.i` and address the global interface files. Then we will address some issues in the module specific interface files.

19.3.1 *dolfin.i and the cpp module*

The file `dolfin.i` starts by defining the name of the generated Python module.

SWIG code

```
%module(package="dolfin", directors="1") cpp
```

This statement tells SWIG to create a module called `cpp` that resides in the package DOLFIN. We also enable the use of directors. This is required to be able to subclass DOLFIN classes in Python, an issue that will be discussed below. By naming the generated extension module `cpp`, and including it in the DOLFIN Python package, we hide the generated interface into a submodule of DOLFIN; the `dolfin.cpp` module. The DOLFIN module then imports the needed classes and functions from `dolfin.cpp` in the `__init__.py` file along with additional pure Python classes and functions.

The next two blocks of `dolfin.i` define code that will be inserted into the SWIG generated C++ wrapper file.

SWIG code

```
%{
#include <dolfin/dolfin.h>
#define PY_ARRAY_UNIQUE_SYMBOL PyDOLFIN
#include <numpy/arrayobject.h>
%}

%init%{
import_array();
%}
```

SWIG inserts code that resides in a `%{...%}` block, verbatim at the top of the generated C++ wrapper file. Note that `%{...%}` is short for `%header%{...%}`. Hence, the first block of code is similar to the include statements you would put in a standard C++ program. The code in the second block, `%init%{...}%`, is inserted in the code where the Python module is initialized. A typical example of such a function is `import_array()`, which initializes the NumPy module. SWIG provides several such statements, each inserting code verbatim into the wrapper file at different positions.

19.3.2 Reference counting using `shared_ptr`

In the previous example dealing with `operator+=`, we saw that it is important to prevent premature destruction of underlying C++ objects. A nice feature of SWIG is that we can declare that a wrapped class shall store the underlying C++ object using a shared pointer (`shared_ptr`), instead of a raw pointer. By doing so, the underlying C++ object is not explicitly deleted when the reference count of the Python object reach zero, instead the reference count on the `shared_ptr` is decreased.

Shared pointers are provided by the `boost_shared_ptr.i` file. This file defines the directive: `%shared_ptr`. The directive must be used for each class we want shared pointers for. In DOLFIN this is done in the `shared_ptr_classes.i` file. Note that when the `%shared_ptr` directive is called, typemaps for passing a `shared_ptr` stored object to method that expects a reference or a pointer is also defined. This means that the typemap pass a de-referenced `shared_ptr` to the function. This behavior can lead to unintentional trouble because the `shared_ptr` mechanism is circumvented.

In DOLFIN, instances of some crucial classes are stored internally with `shared_ptrs`. These classes also uses `shared_ptr` in the Python interface. When objects of these classes are passed as arguments to methods or constructors in C++, two versions are needed: a `shared_ptr` and a reference version. The following code snippet illustrates two constructors of `Function`, each taking a `FunctionSpace` as an argument[2]:

[2]Instances of `FunctionSpace` are internally stored using `shared_ptr` in the DOLFIN C++ library.

C++ code

```
/// Create function on given function space
explicit Function(const FunctionSpace& V);

/// Create function on given function space (shared data)
explicit Function(boost::shared_ptr<const FunctionSpace> V);
```

Instances of `FunctionSpace` in DOLFIN are stored using `shared_ptr`. Hence, we want SWIG to use the second constructor. However, SWIG generates de-reference typemaps for the first constructor. So when a `Function` is instantiated with a `FunctionSpace`, SWIG will unfortunately pick the first constructor and the `FunctionSpace` is passed without increasing the reference count of the `shared_ptr`. This undermines the whole concept of `shared_ptr`. To prevent this faulty behavior, we ignore the reference constructor (see `function_pre.i`):

SWIG code

```
%ignore dolfin::Function::Function(const FunctionSpace&);
```

19.3.3 Typemaps

Most types in the `kernel_module.i` file are wrapped nicely with SWIG. However, as in the `Array` example above, there is need for typemaps, for instance to handle NumPy arrays. In `dolfin.i` we include different types of global typemaps. They are called global because they are defined for the whole DOLFIN library. A typemap is not global if it is included in the kernel specific interface files, see below. Here we present some of the global typemaps defined in the files included in `dolfin.i`.

In `typemaps.i` are typemaps and so called SWIG fragments (explained later), defined for the three primitive types:`int`, `dolfin::uint` and `double`.

The simplest typemap is an out-typemap for `dolfin::uint` a typedef of `unsigned int`. This typemap is needed since Python does not have an equivalent of an `unsigned int` type:

SWIG code

```
%typemap(out, fragment=SWIG_From_frag(unsigned int)) unsigned int
{
  // Typemap unsigned int
  $result = SWIG_From(unsigned int)($1);
}
```

Here we specify that a function returning an `unsigned int` will use the SWIG provided type conversion macro: `SWIG_From(unsigned int)(arg)` to convert the `unsigned int` to a Python `int`. The macro is not provided by default in SWIG. We therefore need to specify that SWIG includes the definition of the macro in the wrapper file by using the `fragment` argument to the `typemap` directive.

The next typemap is an in typemap for `unsigned int`.

SWIG code

```
%typemap(in, fragment="Py_convert_uint") unsigned int
{
  if (!Py_convert_uint($input, $1))
    SWIG_exception(SWIG_TypeError, "expected positive 'int' for argument $argnum");
}
```

This typemap is almost as simple as the corresponding out typemap. It employs the fragment provided function: `Py_convert_uint`, to do the type check and the Python to C++ conversion. If the conversion is not successful it will return `false` and a Python exception is raised. The built in SWIG function, `SWIG_exception` is used to raise the Python exception. These predefined SWIG exceptions are defined

in the `exception.i` file, included in `dolfin.i`. Note that SWIG expands the `$argnum` variable to the argument number for which the `dolfin::uint` typemap is used for. Including this number in the string creates more understandable error message. The `Py_convert_uint` fragment is defined in the same file and looks like:

SWIG code

```
%fragment("Py_convert_uint", "header", fragment="PyInteger_Check") {
  SWIGINTERNINLINE bool Py_convert_uint(PyObject* in, dolfin::uint& value)
  {
    if (!(PyInteger_Check(in) && PyInt_AS_LONG(in)>=0))
      return false;
    value = static_cast<dolfin::uint>(PyInt_AS_LONG(in));
    return true;
  }
}
```

The first string in the `fragment` declaration is the name of the `fragment` and the second string defines where the code should be inserted. Here we insert it into the `"header"` section of the generated code. This is similar to the code inserted using `%{...%}` above. Here we also rely on another fragment provided function: `PyInteger_Check`, which we provide as a substitute to the built in `PyInt_Check` function. The reason for that is that `PyInt_Check` in Python2.6 can not be combined with NumPy `Int`.

In `numpy_typemaps.i` are typemaps for C-arrays of primitive types: `double`, `int` and `dolfin::uint` defined. As in the `Array` example in the previous section, these typemaps let a user pass a NumPy array of the corresponding type as arguments to functions, methods, and operators. Instead of writing one typemap for each primitive type we define a SWIG macro, which instantiates a typemap for a particular primitive type when it is called. Some of these typemaps are used frequently and can therefore produce a lot of code. To avoid code bloat most of the typemap code is place in the function `convert_numpy_to_array_no_check(TYPE_NAME)`, which is called by the actual typemap. The code is defined within a `fragment` directive, which means that a typemap can make use of that code by adding the name of the fragment as an argument in the typemap definition. The entire macro reads:

SWIG code

```
%define UNSAFE_NUMPY_TYPEMAPS(TYPE,TYPE_UPPER,NUMPY_TYPE,TYPE_NAME,DESCR)

%fragment(convert_numpy_to_array_no_check(TYPE_NAME), "header") {
  // Typemap function (Reducing wrapper code size)
  SWIGINTERN bool convert_numpy_to_array_no_check_ ## TYPE_NAME(PyObject* input, TYPE*& ret)
  {
    if (PyArray_ISCONTIGUOUS(xa) && PyArray_TYPE(xa) == NUMPY_TYPE)
    {
      PyArrayObject *xa = reinterpret_cast<PyArrayObject*>(input);
      if ( PyArray_TYPE(xa) == NUMPY_TYPE )
      {
        ret  = static_cast<TYPE*>(PyArray_DATA(xa));
        return true;
      }
    }
    PyErr_SetString(PyExc_TypeError,"contiguous numpy array of 'TYPE_NAME' expected. Make sure that
         the numpy array is contiguous and uses dtype='DESCR'.");
    return false;
  }
}

// The typecheck
% typecheck(SWIG_TYPECHECK_ ## TYPE_UPPER ## _ARRAY) TYPE * {
    $1 = PyArray_Check($input) ? 1 : 0;
```

```
}

// The typemap
%typemap(in, fragment=convert_numpy_to_array_no_check(TYPE_NAME)) TYPE * {
if (!convert_numpy_to_array_no_check_ ## TYPE_NAME($input,$1))
    return NULL;
}
```

The first line defines the signature of the macro. The macro is called using 5 arguments:

1. `TYPE` is the name of the primitive type. Examples are `dolfin::uint` and `double`.
2. `TYPE_UPPER` is the name of the type check name that SWIG uses. Examples are `INT32` and `DOUBLE`.
3. `NUMPY_TYPE` is the name of the NumPy type. Examples are `NPY_UINT` and `NPY_DOUBLE`.
4. `TYPE_NAME` is a short type name used in exception string. Examples are `uint` and `double`.
5. `DESCR` is a description character used in NumPy to describe the type. Examples are `'I'` and `'d'`.

We can then call the macro to instantiate the typemaps and type checks.

SWIG code

```
UNSAFE_NUMPY_TYPEMAPS(dolfin::uint,INT32,NPY_UINT,uint,I)
UNSAFE_NUMPY_TYPEMAPS(double,DOUBLE,NPY_DOUBLE,double,d)
```

Here, we have instantiated the typemap for a `dolfin::uint` and a `double` array. The above typemap does not check the length of the handed NumPy array and is therefore unsafe. Corresponding safe typemaps can also be found in `numpy_typemaps.i`. The conversion function included in the fragment declaration

SWIG code

```
SWIGINTERN bool convert_numpy_to_array_no_check_ ## TYPE_NAME(PyObject* input, TYPE*& ret)
```

takes a pointer to a `PyObject` as input. This function returns `true` if the conversion is successful and `false` otherwise. The converted array will be returned by the `TYPE*& ret` argument. Finally, the `%apply TYPE* {TYPE* _array}` directive means that we want the typemap to apply to any argument of type `TYPE*` with argument name `_array`. In this way we copy the already defined typemap to an C-array argument with the name `_array`. Note that `## TYPE_NAME` is a SWIG macro directive that will be expanded to the value of the `TYPE_NAME` macro argument.

In `std_vector_typemaps.i`, several typemaps are defined which allow users to pass either `NumPy` arrays or Python lists where a `std::vector` is expected. One is an in-typemap macro for passing a `std::vector` of pointers of DOLFIN objects to a C++ function and another is an out-typemap macro for passing a `std::vector` of primitives, using NumPy arrays, to Python. It is not strictly necessary to add these typemaps as SWIG provides interface files to handle templated types from `std::vector`. However, the provided `std::vector` functionality generates a lot of code and the resulting objects are not very Pythonic. We have therefore chosen to implement our own typemaps to handle `std::vector` arguments.

The first typemap we describe enables the use of a Python list of DOLFIN objects instead of a `std:vector` of pointers to such objects. Since the handed DOLFIN objects may and may not be stored using a `shared_ptr`, we provide a typemap that works for both situations. We also create typemaps for arguments where different combinations of `const` are used. Typically a signature in the DOLFIN code can look like:

SWIG code

```
{const} std::vector<{const} dolfin::TYPE *>
```

where `const` is optional. To handle the optional `const`s we use nested macros:

SWIG code

```
%define IN_TYPEMAPS_STD_VECTOR_OF_POINTERS(TYPE)
// Make SWIG aware of the shared_ptr version of TYPE
%types(SWIG_SHARED_PTR_QNAMESPACE::shared_ptr<TYPE>*);
IN_TYPEMAP_STD_VECTOR_OF_POINTERS(TYPE,const,)
IN_TYPEMAP_STD_VECTOR_OF_POINTERS(TYPE,,const)
IN_TYPEMAP_STD_VECTOR_OF_POINTERS(TYPE,const,const)
%enddef

%define IN_TYPEMAP_STD_VECTOR_OF_POINTERS(TYPE,CONST,CONST_VECTOR)
%typecheck(SWIG_TYPECHECK_POINTER) CONST_VECTOR std::vector<CONST dolfin::TYPE *> &
{
  $1 = PyList_Check($input) ? 1 : 0;
}

%typemap (in) CONST_VECTOR std::vector<CONST dolfin::TYPE *> & (std::vector<CONST dolfin::TYPE *>
     tmp_vec, SWIG_SHARED_PTR_QNAMESPACE::shared_ptr<dolfin::TYPE> tempshared)
{
  if (!PyList_Check($input)) {
    SWIG_exception(SWIG_TypeError, "list of TYPE expected");
  int size = PyList_Size($input);
  int res = 0;
  PyObject * py_item = 0;
  void * itemp = 0;
  int newmem = 0;
  tmp_vec.reserve(size);
  for (int i = 0; i < size; i++) {
    py_item = PyList_GetItem($input,i);
    res = SWIG_ConvertPtr(py_item, &itemp, $descriptor(dolfin::TYPE *), 0);
    if (SWIG_IsOK(res)) {
      tmp_vec.push_back(reinterpret_cast<dolfin::TYPE *>(itemp));
    }
    else {
      // If failed with normal pointer conversion then
      // try with shared_ptr conversion
      newmem = 0;
      res = SWIG_ConvertPtrAndOwn(py_item, &itemp,
           $descriptor(SWIG_SHARED_PTR_QNAMESPACE::shared_ptr< dolfin::TYPE > *), 0, &newmem);
      if (SWIG_IsOK(res)) {
        if (itemp) {
          tempshared = *(reinterpret_cast< SWIG_SHARED_PTR_QNAMESPACE::shared_ptr<dolfin::TYPE> * >
               (itemp));
          tmp_vec.push_back(tempshared.get());
        }
        // If we need to release memory
        if (newmem & SWIG_CAST_NEW_MEMORY) {
          delete reinterpret_cast< SWIG_SHARED_PTR_QNAMESPACE::shared_ptr< dolfin::TYPE > * >
               (itemp);
        }
      }
      else {
        SWIG_exception(SWIG_TypeError, "list of TYPE expected (Bad conversion)");
      }
    }
  }
  $1 = &tmp_vec;
```

```
}
%enddef
```

In the typemap, we first check that we get a Python list. We then iterate over the items and try to acquire the specified C++ object by converting the Python object to the underlying C++ pointer. This is accomplished by:

SWIG code

```
res = SWIG_ConvertPtrAndOwn(py_item, &itemp, $descriptor(dolfin::TYPE *), 0, &newmem);
```

If the conversion is successful we push the C++ pointer to the `tmp_vec`. If the conversion fails we try to acquire a `shared_ptr` version of the C++ object instead. If neither of the two conversions succeed we raise an error.

The second typemap defined for `std::vector` arguments is a so called argout-typemap. This kind of typemap is used to return values from arguments. In C++ non `const` references or pointers arguments are commonly used both as input and output of functions. In Python are output arguments returned by the function. The following call to the `GenericMatrix::getrow` method illustrates the difference between C++ and Python. The C++ signature is:

SWIG code

```
GenericMatrix::getrow(dolfin::uint row, std::vector<uint>& columns, std::vector<double>& values)
```

Here, the sparsity pattern associated with row number `row` is filled into the `columns` and `values` vectors. In Python, a corresponding call should look like:

Python code

```
columns, values = A.getrow(row)
```

To obtain the desired Python behavior we employ argout-typemaps. The following macro defines such a typemap:

SWIG code

```
%define ARGOUT_TYPEMAP_STD_VECTOR_OF_PRIMITIVES(TYPE, TYPE_UPPER, ARG_NAME, NUMPY_TYPE)
// In typemap removing the argument from the expected in list
%typemap (in,numinputs=0) std::vector<TYPE>& ARG_NAME (std::vector<TYPE> vec_temp)
{
  $1 = &vec_temp;
}

%typemap(argout) std::vector<TYPE> & ARG_NAME
{
  npy_intp size = $1->size();
  PyArrayObject *ret = reinterpret_cast<PyArrayObject*>(PyArray_SimpleNew(1, &size, NUMPY_TYPE));
  TYPE* data = static_cast<TYPE*>(PyArray_DATA(ret));
  for (int i = 0; i < size; ++i)
    data[i] = (*$1)[i];

  // Append the output to $result
  %append_output(PyArray_Return(ret));
}
%enddef
```

The macro begins by defining an in-typemap that removes the output argument and instantiates the `std::vector` that will be passed as argument to the C++ function. Then we have the code for the argout-typemap, which is inserted after the C++ call. Here, the "returned" C++ arguments are transformed to Python arguments, by instantiating a NumPy array `ret` and filling it with the values

from the `std::vector`. Note that here we are forced to copy the values, or else the return argument would overwrite any previous created return argument, with memory corruption as result.

19.3.4 DOLFIN header files and Python docstrings

As mentioned earlier, the file `kernel_module.i`, generated by `generate.py`, tells SWIG what parts of DOLFIN that should be wrapped. The associated script `generate_docstrings.py` generates the Python docstrings extracted from comments in the C++ documentation. The comments are transformed into SWIG docstring directives like:

SWIG code

```
%feature("docstring")  dolfin::Data::ufc_cell "
Return current UFC cell (if available)
";
```

and saved to a SWIG interface file `docstrings.i`. The `docstrings.i` file is included from the main `dolfin.i` file. Note that the `kernel_module.i` and `docstrings.i` files are not generated automatically during the build process. This means that when a header file is added to the DOLFIN library, one must to manually run `generate.py` to update the `kernel_module.i` and `docstrings.i` files.

19.3.5 Specializations of kernel modules

The DOLFIN SWIG interface file `kernel_module.i` mirrors the directory structure of DOLFIN. As mentioned above, many directories come with specializations in `..._pre.i` and `..._post.i` files. Below, we will highlight some of these specializations.

The `mesh` module. The `mesh` module defines the Python interfaces for `Mesh`, `MeshFunction`, `MeshEntity`, and their subclasses. In `Mesh` the geometrical and topological information is stored in contiguous C-arrays. These arrays are accessible from Python using methods that return the underlying data wrapped to NumPy arrays. With this functionality, for example move a mesh 1 unit in the x-direction as follows:

Python code

```
mesh.coordinates()[:,0] += 1
```

To obtain this functionality we ignore the original C++ methods and then extend the `Mesh` class with our own versions. The code for this functionality is found in `mesh_pre.i`:

SWIG code

```
%ignore dolfin::Mesh::coordinates;
%ignore dolfin::Mesh::cells;
...
%extend dolfin::Mesh {
  PyObject* coordinates() {
    return %make_numpy_array(2, double)(self->num_vertices(),
          self->geometry().dim(),
          self->coordinates(), true);
  }

  PyObject* cells() {
    return %make_numpy_array(2, uint)(self->num_cells(), self->topology().dim()+1,
              self->cells(), false);
  }
}
```

Here, we first ignore the C++ versions of the `coordinates` and `cells` methods. Then we re-implements them using the `make_numpy_array` macro. This macro takes a pointer to a C-array, together with the dimension and size of that array. The last argument is used to set the writable flag of the NumPy array. This flag is set to `true` for `coordinates` and false for `cells`.

In a similar fashion, we use the `make_numpy_array` macro to wrap the connectivity information to Python. This is done with the following SWIG directives in the `mesh_pre.i` files.

SWIG code

```
%ignore dolfin::MeshConnectivity::operator();
%extend dolfin::MeshConnectivity {
  PyObject* __call__() {
    return %make_numpy_array(1, uint)(self->size(), (*self)(), false);
  }
  ...
}
```

Here, we extend the C++ extension layer of the `dolfin::MeshConnectivity` class with a `__call__` method. The method returns all connections between any two types of topological dimensions in the mesh.

In `mesh_pre.i` we also declare that it should be possible to subclass `SubDomain` in Python. This is done using the `%director` directive.

SWIG code

```
%feature("director") dolfin::SubDomain;
```

To avoid code bloat this feature is only included for certain classes. The following typemap enables seamless integration of NumPy array and the `Array<double>&` in the `inside` and `map` methods.

SWIG code

```
%typemap(directorin) const dolfin::Array<double>& {
  $input = %make_numpy_array(1, double)($1_name.size(), $1_name.data().get(), false);
 }
```

Even if it by concept and name is an *in*-typemap, one can look at it as an out-typemap (since it is a typemap for a callback function). SWIG needs to wrap the arguments that the implemented `inside` or `map` method in Python are called with. The above typemap is inserted in the `inside` and `map` methods of the SWIG created C++ director class, which is a subclass of `dolfin::SubDomain`.

DOLFIN comes with a `MeshEnitityIterator` class. This class lets a user iterate over a given `MeshEntity`: a `cell`, a `vertex` and so forth. The iterators are mapped to Python by the increment and de-reference operators in `MeshEnitityIterator`. This enabling is done by renaming these operators in `mesh_pre.i`:

SWIG code

```
%rename(_increment) dolfin::MeshEntityIterator::operator++;
%rename(_dereference) dolfin::MeshEntityIterator::operator*;
```

In `mesh_post.i`, the Python iterator protocol (`__iter__` and `next`) is implemented for the `MeshEnitityIterator` class as follows:

SWIG code

```
%extend dolfin::MeshEntityIterator {
%pythoncode
%{
def __iter__(self):
```

```
  self.first = True
  return self

def next(self):
  self.first = self.first if hasattr(self,"first") else True
  if not self.first:
    self._increment()
  if self.end():
    raise StopIteration
  self.first = False
  return self._dereference()
%}
}
```

We also rename the iterators to `vertices` for the `VertexIterator`, `cells` for `CellIterator`, and so forth. Iteration over a certain mesh entity in Python is then done by:

Python code

```
for cell in cells(mesh):
    ...
```

The `la` module. The Python interface of the vector and matrix classes in the `la` module is heavily specialized, because we want the interface to be intuitive and integrate nicely with NumPy. First, all of the implemented C++ operators are ignored, just like we did for the `operator+=()` in the `Array` example above. This is done in the `la_pre.i` file:

SWIG code

```
%ignore dolfin::GenericVector::operator[];
%ignore dolfin::GenericVector::operator*=;
%ignore dolfin::GenericVector::operator/=;
%ignore dolfin::GenericVector::operator+=;
%ignore dolfin::GenericVector::operator-=;

%rename(_assign) dolfin::GenericVector::operator=;
```

Here, we only ignore the virtual operators in the base class `GenericVector`, because SWIG only implements a Python version of a virtual method in the base class. Only the base class implementation is needed since a method call in a derived Python class ends up in the corresponding Python base class. The base class in Python hands the call over to the corresponding base class call in C++, which ends up in the corresponding derived C++ class. Hence, when we ignore the above mentioned operators in the base class, we also ignore the same operators in the derived classes. Finally, we rename the assignment operator to `_assign`. The `_assign` operator will be used by the `slice` operator implemented in `la_post.i`, see below.

The following code snippet from `la_post.i` shows how the special method `__mul__` in the Python interface of `GenericVector` is implemented:

SWIG code

```
%extend dolfin::GenericVector {
  void _scale(double a)
  {(*self)*=a;}

  void _vec_mul(const GenericVector& other)
  {(*self)*=other;}

  %pythoncode %{
```

```
   ...
    def __mul__(self,other):
        """x.__mul__(y) <==> x*y"""
        if isinstance(other,(int,float)):
            ret = self.copy()
            ret._scale(other)
            return ret
        if isinstance(other, GenericVector):
            ret = self.copy()
            ret._vec_mul(other)
            return ret
        return NotImplemented
    ...
%} }
```

We first expose `operator*=` to Python by implementing corresponding hidden methods, the `_scale` method for scalars and the `_vec_mul` method for other vectors. These methods are then used in the `__mul__` special method in the Python interface.

Vectors in the DOLFIN Python interface support access and assignment using slices and NumPy arrays of booleans or integers, and lists of integers. This is achieved using the `get` and `set` methods in the `GenericVector`, but is quite technical. In short, a helper class `Indices` is implemented in `Indices.i`. This class is used in the `_get_vector` and `_set_vector` helper functions defined in the `la_get_set_items.i` file.

Python code

```
%extend dolfin::GenericVector {
  %pythoncode %{
   ...
    def __getslice__(self, i, j):
        if i == 0 and (j >= len(self) or j == -1):
            return self.copy()
        return self.__getitem__(slice(i, j, 1))

    def __getitem__(self, indices):
        from numpy import ndarray, integer
        from types import SliceType
        if isinstance(indices, (int, integer)):
            return _get_vector_single_item(self, indices)
        elif isinstance(indices, (SliceType, ndarray, list) ):
            return down_cast(_get_vector_sub_vector(self, indices))
        else:
            raise TypeError, "expected an int, slice, list or numpy array of integers"
  ...
%} }
```

The above code demonstrates the implementation of the slice and index access in the Python layer of `GenericVector`. When accessing a vector using a full slice, `v[:]`, `__getslice__` is called with `i` = 0 and `j` = a-large-number (default in Python). In this case, we return a copy of the vector. Otherwise, we create a slice and pass it to `__getitem__`. In the latter method, we check whether the `indices` argument is a single item or not and calls upon the correct helper functions.

19.4 JIT compilation of UFL forms, `Expressions` and `SubDomains`

The DOLFIN Python interface makes extensive use of just in time (JIT) compilation; that is code that is compiled, linked and imported into Python using Instant, see Chapter 14. This process is facilitated

by employing the form compilers FFC or SFC that translates UFL code into UFC code. In a similar fashion, DOLFIN enables JIT compilation of `Expressions` and `SubDomains`.

We provide two ways of defining an `Expression` in DOLFIN via Python: subclassing `Expression` directly in Python, and through the compile function interface. In the first alternative, the `eval` method is defined in a subclass of Expression:

Python code

```
class MyExpression(Expression):
    def eval(self, values, x):
        values[0] = 10*exp(-((x[0] - 0.5)**2 + (x[1] - 0.5)** 2) / 0.02)
f = MyExpression()
```

Here, `f` will be a subclass of both `ufl.Function` and `cpp.Expression`. The second alternative is to instantiate the `Expression` class directly:

Python code

```
f = Expression("10*exp(-(pow(x[0] - 0.5, 2) + pow(x[1] - 0.5, 2)) / 0.02)")
```

This example creates a scalar `Expression`. Vector valued and matrix valued expressions can also be created. As in the first example, `f` will be a subclass of `ufl.Function`. But it will not inherit from `cpp.Expression`. Instead, we create a subclass in C++ that inherit from `dolfin::Expression` and implements the `eval` method. The generated code looks like:

C++ code

```
class Expression_700475d2d88a4982f3042522e5960bc2: public Expression{
public:
  Expression_700475d2d88a4982f3042522e5960bc2():Expression(2){}

  void eval(double* values, const double* x) const{
    values[0] = 10*exp(-(pow(x[0] - 0.5, 2) + pow(x[1] - 0.5, 2)) / 0.02);
  }
};
```

The name of the subclass is generated from a hash of the passed expression string. The code is inserted into `namespace dolfin` and the appropriate `#include` is also inserted in the code. Instant is used to compile and link a Python module from the generated code. The class made by Instant is imported into Python and used to dynamically construct a class that inherits the generated class together with `ufl.Function` and `Expression`. Dynamic creation of classes in Python is done using so called meta-classes. In a similar fashion, DOLFIN provides functionality to construct C++ code and JIT compile subclasses of `SubDomain` from Python.

19.5 Debugging mixed language applications

Debugging mixed language applications may be more challenging than debugging single language applications. The main reason is that debuggers are written mainly for either compiled languages or scripting languages. However, as we will show, mixed language applications can be debugged in much of the same way as compiled languages.

Before starting the debugger, you should make sure that your library, or the relevant parts of it, is compiled with the debugging on. With GCC this is done with the `-g` option. The additional debugging information in the code slows down the performance. Therefore, DOLFIN is by default not compiled with `-g`. After compiling the code with debugging information, you may start Python in GDB, the GNU Project Debugger, as follows:

Bash code

```
gdb python
(gdb) run
...
```

The problem with GDB is that only one thread is running. This means that you will not be able to set break points and so on once you have started the Python interpreter.

However, DDD, the Data Display Debugger, facilitates running the debugger and the Python interpreter in two different threads. That is, you will have two interactive threads, one debugger and the Python interpreter, during your debugging session. The DDD debugger is started as:

Bash code

```
ddd python
```

The crucial next step is to start the Python session in a separate execution window by clicking on `View->Execution Window`. Then you start the Python session:

Bash code

```
(gdb) run
```

After importing for your module you may click or search (using the `Lookup` field) through the source code to set breakpoints, print variables and so on.

Another useful application for analyzing memory management is Valgrind. To find memory leaks, do as follows:

Bash code

```
valgrind --leak-check=full python test_script.py
```

Valgrind also provides various profilers for performance testing.

Acknowledgments. The authors are very thankful to Johan Jansson who initiated the work on the DOLFIN Python interface and to Ola Skavhaug and Martin S. Alnæs who have contributed significantly to the development. Finally, Marie Rognes has improved the language in this chapter significantly.

Part III

Applications

20 Finite elements for incompressible fluids

By Andy R. Terrel, L. Ridgway Scott, Matthew Gregg Knepley, Robert C. Kirby and Garth N. Wells

The structure of the finite element method offers a user a range of choices. This is especially true for solving incompressible fluid problems, where theory points to a number of stable finite element formulations. Using automation tools, we implement and examine various stable formulations for the steady-state Stokes equations. It is demonstrated that the expressiveness of the FEniCS Project components allows solvers for the Stokes problem that use various element formulations to be created with ease.

20.1 Stokes equations

The Stokes equations describe steady, incompressible low Reynolds number flows. For a domain $\Omega \subset \mathbb{R}^d$, where $1 \leq d \leq 3$, the Stokes equations read:

$$-\Delta u + \nabla p = f \quad \text{in } \Omega, \tag{20.1}$$

$$\nabla \cdot u = 0 \quad \text{in } \Omega, \tag{20.2}$$

$$u = 0 \quad \text{on } \partial\Omega, \tag{20.3}$$

where $u : \Omega \to \mathbb{R}^d$ is the velocity field, $p : \Omega \to \mathbb{R}$ is the pressure field and $f : \Omega \to \mathbb{R}^d$ is a source term.

In developing a variational formulation for solving the Stokes equations, a possibility is to search for solutions to a variational formulation of (20.1) in a space of divergence-free functions, thereby satisfying (20.2) by construction. However, this does not translate well to finite element formulations. Alternatively, a mixed variational formulation can be considered as follows. Let $V = [H_0^1(\Omega)]^d$ and $\Pi = \{q \in L^2(\Omega) : \int_\Omega q \, \mathrm{d}x = 0\}$. Given $f \in [L^2(\Omega)]^d$, find functions $u \in V$ and $p \in \Pi$ such that

$$a(u, v) - b(v, p) = (f, v) \quad \forall \ v \in V, \tag{20.4}$$

$$b(u, q) = 0 \quad \forall \ q \in \Pi, \tag{20.5}$$

where

$$a(u, v) := \int_\Omega \nabla u \cdot \nabla v \, \mathrm{d}x, \tag{20.6}$$

$$b(v, q) := \int_\Omega (\nabla \cdot v) \, q \, \mathrm{d}x, \tag{20.7}$$

$$(f, v) := \int_\Omega f \cdot v \, \mathrm{d}x. \tag{20.8}$$

20.2 Finite element formulations for the mixed Stokes problem

We will consider in this section finite element formulations for finding approximate solutions to the mixed formulation of (20.4)–(20.5). The Stokes problem has been studied extensively in the context of finite element methods, with some key results presented in Brezzi and Fortin (1991) and Brenner and Scott (2008). It is a challenging problem on a number of fronts. Firstly, finite element subspaces of V and Π must be chosen carefully to ensure stability of the resulting finite element problem. Secondly, the mixed variational form is a saddle point problem, which leads to indefinite matrix equations. Such systems are particularly taxing on iterative linear solvers. Moreover, conservation of mass requires that the velocity field is divergence-free; very few schemes can satisfy this condition point-wise. The degree to which this condition is imposed depends on the specific scheme.

Stable mixed finite element methods for the Stokes equations must satisfy the Ladyzhenskaya–Babuška–Brezzi (LBB) (or inf–sup) compatibility condition (see Brezzi and Fortin (1991) for more details). The most straightforward scheme — equal-order continuous Lagrange finite element spaces for both pressure and velocity components — leads to an unstable problem. Additionally, mixed element formulations can exhibit a type of "locking", which in practice is sometimes remedied by using inexact quadrature for the $b(v, q)$-type terms (Engelman et al., 1982). This has been recognized as equivalent to modifying the pressure space. Here we take the modern perspective and work with velocity and pressure spaces that are known to satisfy the LBB condition.

An approach to circumventing the difficulties associated with the saddle point nature of the Stokes problem is to modify the discrete variational problem by augmenting the continuity equation with a term that involves the pressure. With appropriate modification of the discrete problem, stability can be proved for equal-order formulations (Hughes et al., 1986). Careful modifications lead to a discrete problem that does not violate consistency.

Few numerical studies of the Stokes problem address more than one finite element formulation. This can be attributed to the difficultly in implementing a number of the known stable methods. With automated code generation, solvers for a range of methods can be easily produced; it is as simple as redefining the finite element spaces or modifying the variational formulation. In the remainder of this section, we demonstrate the construction of a variety stable finite element solvers for the mixed form of the Stokes equations.

20.2.1 Formulations based on compatible function spaces

We consider a number of LBB-stable formulations that are based on the selection of compatible function spaces for the velocity and pressure fields. The generic UFL input for most of these formulations is shown in Figure 20.1. Following the UFL convention, the bilinear and linear forms are named `a` and `L`, respectively. Different finite element spaces are defined via the element type for the velocity (`U_element`), the basis function order for the velocity (`U_order`), the element type for the pressure (`P_element`) and the basis function order for the pressure (`P_order`). From the input in Figure 20.1, FFC generates the problem-specific code used in numerical simulations.

One of the most widely used family of finite elements for the Stokes equations is the Taylor–Hood family (Taylor and Hood, 1973; Boffi, 1997). It consists of a continuous P_q ($q \geqslant 2$) Lagrange element for the velocity components and a continuous P_{q-1} Lagrange element for the pressure field (see Figure 20.2 for the $q = 3$ case). The order of the pressure convergence is lower than that for the velocity. For the UFL extract in Figure 20.1, the Taylor-Hood element corresponds to `U_element = Lagrange`, `U_order = q`, `P_element = Lagrange` and `P_order = q-1`.

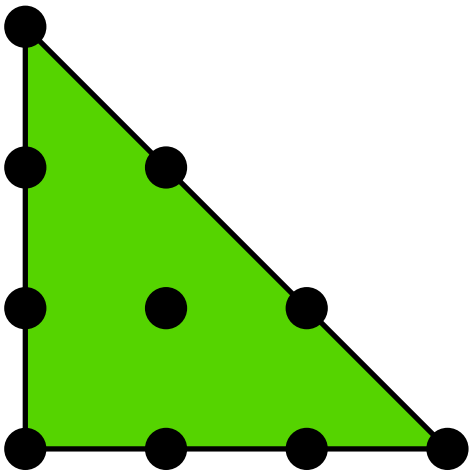
The Crouzeix–Raviart element (Crouzeix and Raviart, 1973) is a non-conforming element that uses integral moments over cell edges as a basis for the velocity, and a discontinuous pressure space that is one order lower than the velocity space. For the lowest order case, the velocity edge moments

Figure 20.1: Generic UFL input for the mixed Stokes problem.

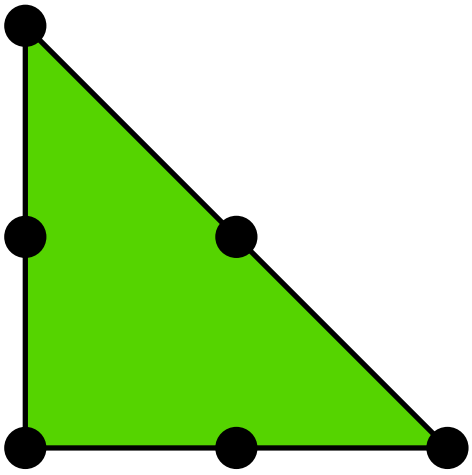
Python code

```
# Define function spaces
V = VectorFunctionSpace(mesh, U_element, U_order)
Q = FunctionSpace(mesh, P_element, P_order)
W = V * Q

# Define trial and test functions
(u, p) = TrialFunctions(W)
(v, q) = TestFunctions(W)

# Define the variational problems
a = inner(grad(u), grad(v))*dx - p*div(v)*dx + div(u)*q*dx
L = inner(f, v)*dx
```

Figure 20.2: A Taylor–Hood element with (a) cubic velocity basis and (b) quadratic pressure basis.



(a) P_3 for V



(b) P_2 for Π

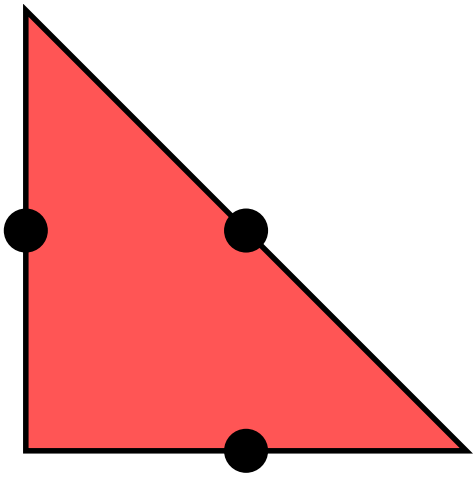


Figure 20.3: Crouzeix–Raviart element used for the velocity field.

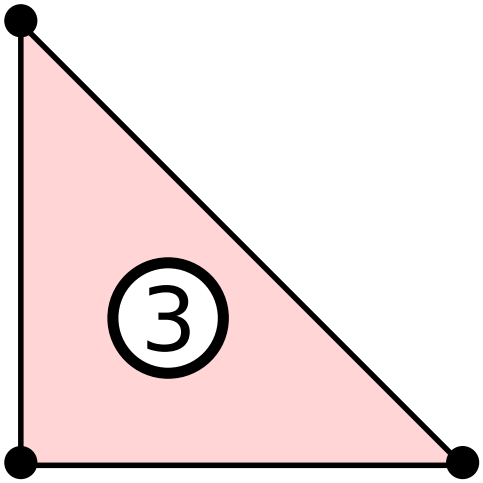


Figure 20.4: Mini element used for the velocity field. It enriches a P_q element with a $q+3$ order bubble function.

are equivalent to evaluating Lagrange basis functions at the center of each edge and the pressure uses P_0 (see Figure 20.3). For the extract in Figure 20.1, the Crouzeix–Raviart element corresponds to `U_element = CR`, `U_order = 1`, `P_element = DiscontinuousLagrange` and `P_order = 0`.

The MINI element (Arnold et al., 1984b) enriches the velocity space via bubble functions, $P_q + B_{q+3}$. The MINI element is illustrated in Figure 20.4. The pressure space uses a continuous P_q Lagrange element. The MINI element was proposed as a cheaper alternative to the Taylor–Hood element. The MINI element is implemented using the "element enrichment" concept from UFL. The UFL definition of the MINI function space is shown in Figure 20.5. At the time of writing, it is only implemented for $q = 1, 2$.

Another possibility is to use a high-degree continuous Lagrange finite element basis for the velocity

Python code

```
# Define function spaces
P = VectorFunctionSpace(mesh, "Lagrange", U_order)
B = VectorFunctionSpace(mesh, "Bubble", U_order + 2)
V = P + B
```

Figure 20.5: UFL input for defining the MINI element velocity space.

	Crouzeix–Raviart	STAB	MINI
`U_element`	`"Crouzeix-Raviart"`	`"Lagrange"`	`"MINI"`
`U_order`	1	q	q
`P_element`	`"Discontinuous Lagrange"`	`"Lagrange"`	`"Lagrange"`
`P_order`	0	q	q
`stabilized`	`False`	`True`	`False`
	CD	**Taylor–Hood**	
`U_element`	`"Lagrange"`	`"Lagrange"`	
`U_order`	q	q	
`P_element`	`"Discontinuous Lagrange"`	`"Lagrange"`	
`P_order`	$q-2$	$q-1$	
`stabilized`	`False`	`False`	

Table 20.1: Element variables defining the different mixed methods.

components and a discontinuous element that is two orders lower for the pressure field. We loosely call this element "CD", for continuous velocity/discontinuous pressure. Brezzi and Fortin (1991) discuss the $P_2 - P_0$ case, and higher order versions were analyzed in Maday et al. (1992). For the extract in Figure 20.1, the CD element corresponds to `U_element = Lagrange`, `U_order = q`, `P_element = DiscontinuousLagrange` and `P_order = q-2`.

Table 20.1 summarizes the specific variables that appear the in the UFL code in Figure 20.1 for the different presented methods.

20.2.2 Pressure stabilized method

To alleviate the difficulties of finding LBB-compatible function spaces, one may use stabilization techniques. Pressure stabilization converts the finite-dimensional formulation from a saddle point problem to a coercive problem. It is usually desirable to modify the finite-dimensional problem such that consistency is not violated. For a more complete discussion see Donea and Huerta (2003). The pressure stabilized method that we consider involves:

$$a(u,v) - b(v,p) = (f,v) \quad \forall\ v \in V_h, \tag{20.9}$$

$$b(u,q) + (\delta\nabla q, \nabla p) = (f, \delta\nabla q) \quad \forall\ q \in \Pi_h, \tag{20.10}$$

where δ is a stabilization parameter, and $V_h \subset V$ and $\Pi_h \subset \Pi$ are suitable finite element spaces. For our tests, $\delta = 0.2h_T^2$, where h_T is two times the circumference of the cell T. For the stabilized tests, continuous Lagrange elements of the same order for both the pressure and velocity spaces are used. This method will be referred to as "STAB". The stabilized method that we adopt is simple, but it does violate consistency for orders $q > 1$. Figure 20.6 illustrates the addition of the stabilization terms to the standard weak form in Figure 20.1. Table 20.1 includes the definitions for the STAB element.

20.3 A penalty approach: the Scott–Vogelius method

Other solutions to deal with the LBB condition include the Uzawa iteration method and penalty methods. A combination of these two approaches results in the iterated penalty method presented in Scott and Vogelius (1985). Let $r \in \mathbb{R}$ and $\rho \in \mathbb{R}^+$ be prescribed parameters. We wish to find $u^n \in V_h$ such that

$$a(u^n,v) + r(\nabla\cdot u^n, \nabla\cdot v) = (f,v) - (\nabla\cdot v, \nabla\cdot w^n) \quad \forall\ v \in V_h, \tag{20.11}$$

Python code

```
# Sample parameters for pressure stabilization
h = CellSize(mesh)
beta = 0.2
delta = beta*h**2

# The additional pressure stabilization terms
a += delta*inner(grad(p), grad(q))*dx
L += delta*inner(f, grad(q))*dx
```

Figure 20.6: UFL code to add stabilization to the mixed method code in Figure 20.1.

Python code

```
# Define function space
V = VectorFunctionSpace(mesh, "Lagrange", U_order)

# Define trial and test functions
u = TrialFunction(V)
v = TestFunction(V)

# Define auxiliary function and parameters
w = Function(V);
rho = 1.0e3
r = -rho

# Define the variational problem
a = inner(grad(u), grad(v))*dx + r*div(u)*div(v)*dx
L = inner(f, v)*dx + inner(div(w), div(v))*dx
U = Function(V)
pde = LinearVariationalProblem(a, L, U, bc0)
solver = LinearVariationalSolver(pde);

# Iterate to fix point
iters = 0; max_iters = 100; U_m_u = 1
while iters < max_iters and U_m_u > 1e-8:
    solver.solve()
    w.vector().axpy(rho, U.vector())
    if iters != 0:
        U_m_u = (U.vector() - u_old_vec).norm("l2")
    u_old_vec = U.vector().copy()
    iters += 1
```

Figure 20.7: DOLFIN code for defining the Scott–Vogelius method.

where

$$w^{n+1} = w^n + \rho u^n. \tag{20.12}$$

The pressure may be recovered from the auxiliary field w via $p = \nabla \cdot w - C$, where C is an arbitrary constant (since the pressure field is only determined up to an arbitrary constant). When computing the error in p, we subtract the average of $\nabla \cdot w$ to account for C. The algorithm initially assumes $w^0 = 0$, and then solves (20.11) and updates w via (20.12). The process is repeated until $\|u^{n+1} - u^n\| < \epsilon$, where ϵ is a prescribed tolerance. This method involves only one function space, but it requires a higher-order continuous element ($q \geq 4$) and it solves the divergence-free criterion exactly. The iteration count and accuracy are dependent upon the penalty parameters ρ and r. For our experiments we use $\rho = -r = 1 \times 10^3$. The implementation of this formulation is presented in Figure 20.7.

q	n	Crouzeix–Raviart	STAB	MINI
1	8	1,056	435	947
	16	4,160	1,635	3,683
	32	16,512	6,339	14,531
2	8	-	1,635	3,171
	16	-	6,339	12,483
	32	-	24,963	49,539
3	8	x	3,603	x
	16	x	14,115	x
	32	x	55,875	x
4	8	-	6,339	x
	16	-	24,963	x
	32	-	99,075	x
5	8	x	9,843	x
	16	x	38,883	x
	32	x	154,563	x
q	n	CD	Taylor–Hood	Scott–Vogelius
2	8	1,346	1,235	-
	16	5,250	4,771	-
	32	20,738	18,755	-
3	8	3,170	2,947	-
	16	12,482	11,523	-
	32	49,538	45,571	-
4	8	5,762	5,427	4,226
	16	22,786	21,347	16,642
	32	90,626	84,675	66,050
5	8	5,762	8,675	6,562
	16	36,162	34,243	25,922
	32	144,002	136,067	103,042

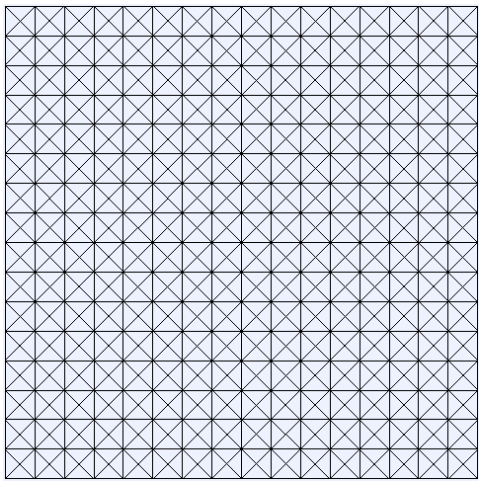
Table 20.2: A comparison of the number of degrees of freedom organized by velocity order (q) and number of mesh divisions per dimension (n). A '-' indicates that the order for that particular element is not stable; an 'x' indicates it is currently not implemented.

20.4 Numerical tests

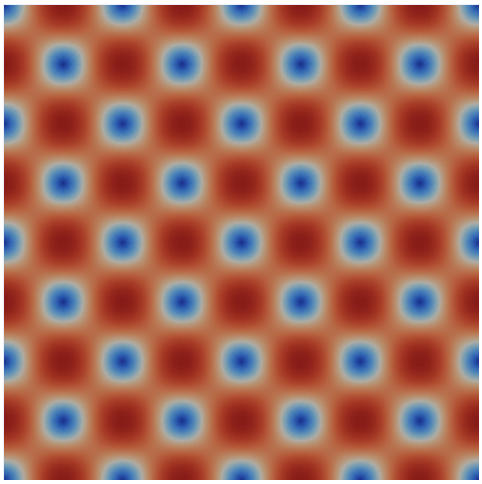
To evaluate the presented methods, we compare the computed results to a manufactured solution for different mesh refinements and element degrees. All simulations use FEniCS tools to generate the discrete problems (FFC v0.7.0, DOLFIN v0.9.4, UFL v0.4.0, UFC v1.2.0), with the UMFPACK LU solver (from the SuiteSparse package v3.4). For iterative methods applied to Stokes problems, we refer to Chapter 35 and Elman et al. (2005). The number of the degrees of freedom as a function of mesh size and element type for the considered cases are shown in Table 20.2.

20.4.1 Simulation set-up

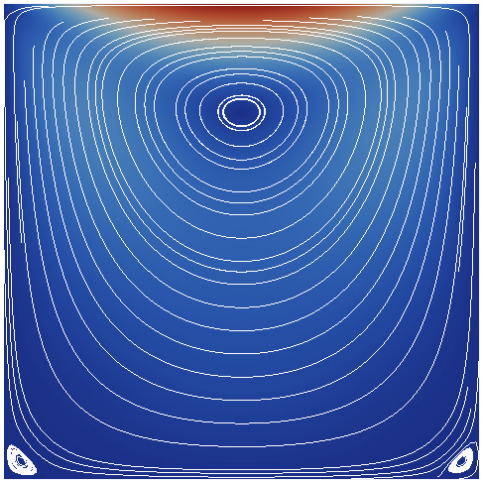
For our tests, we use an $n \times n$ unit square mesh with a crossed triangle pattern, as illustrated in Figure 20.8a. It should be noted that the choice of mesh can affect the convergence results in perhaps surprising ways, e.g., avoiding locking phenomena (Nagtegaal et al., 1974) or spurious pressure-modes (Malkus, 2000). The crossed triangle mesh was used as our test case to avoid subtle issues related to mesh construction (see Brezzi and Fortin (1991, Proposition 6.1, Section VI.6)). Furthermore, since we are using stable elements to begin with, we are not concerned with locking phenomena. For a



(a) Unit square mesh 16×16 with a crossed triangle pattern.



(b) Velocity magnitude of the analytic test problem.



(c) Velocity magnitude and streamlines for the lid-driven cavity test problem.

Figure 20.8: Test mesh and solution plots.

Figure 20.9: DOLFIN code for defining the test domain.

Python code

```
# Define the boundary domains
class NoSlipDomain(SubDomain):
    def inside(self, x, on_boundary):
        return on_boundary

class PinPoint(SubDomain):
    def inside(self, x, on_boundary):
        return x[0] < DOLFIN_EPS and x[1] < DOLFIN_EPS

# Define mesh
mesh = UnitSquare(h_num, h_num, "crossed")

# Instantiate the boundary conditions, set the
# velocity dof values to the exact solution, and
# pinpoint the pressure.
noslip_domain = NoSlipDomain()
noslip = Expression(("sin(4*pi*x[0])*cos(4*pi*x[1])",
                     "-cos(4*pi*x[0])*sin(4*pi*x[1])"))
pinpoint = PinPoint()
pin_val = Expression("pi*cos(4*pi*x[0])*cos(4*pi*x[1])")
bc0 = DirichletBC(W.sub(0), noslip, noslip_domain)
bc1 = DirichletBC(W.sub(1), pin_val, pinpoint, "pointwise")
bc = [bc0, bc1]

# Define the RHS
f = Expression(("28*pi**2*sin(4*pi*x[0])"\
                "cos(4*pi*x[1])",
                "-36*pi**2*cos(4*pi*x[0])*sin(4*pi*x[1])"))
```

comparison, using a mesh with a spurious pressure mode correction on a non-crossed mesh, see Terrel et al. (2008).

As a first test case, we use the following analytical solution of the Stokes equations:

$$f = \begin{bmatrix} 28\pi^2 \sin(4\pi x)\cos(4\pi y) \\ -36\pi^2 \cos(4\pi x)\sin(4\pi y) \end{bmatrix}, \tag{20.13}$$

$$u = \begin{bmatrix} \sin(4\pi x)\cos(4\pi y) \\ -\cos(4\pi x)\sin(4\pi y) \end{bmatrix}, \tag{20.14}$$

$$p = \pi \cos(4\pi x)\cos(4\pi y). \tag{20.15}$$

We also consider a lid-driven cavity problem with a quadratic driving function on the top (see Figures 20.8b and 20.8c).

Figures 20.9 and 20.10 show the DOLFIN Python input for the considered problems. To change from the analytical test problem to the lid-driven cavity, only a change in the boundary condition functions and the right-hand side f are required. The pressure field, which is determined only up to an arbitrary constant, is "pinned" at zero at one pressure degree of freedom. Given a mesh and one of the defined variational problems, DOLFIN will use FFC and FIAT to generate the necessary computer code automatically, allowing for one script to test all the methods.

In computing the error for the analytical test cases in terms of function norms, the exact solution is interpolated using Lagrange elements of degree 10. The code for computing the error is presented in Figure 20.11.

Python code

```
# Define the boundary domains
class NoSlipDomain(SubDomain):
    def inside(self, x, on_boundary):
        return on_boundary and x[1] < 1.0 - DOLFIN_EPS

class Top(SubDomain):
    def inside(self, x, on_boundary):
        return on_boundary and x[1] > 1.0 - DOLFIN_EPS

class PinPoint(SubDomain):
    def inside(self, x, on_boundary):
        return x[0] < DOLFIN_EPS and x[1] < DOLFIN_EPS

# Define mesh
mesh = UnitSquare(h_num, h_num, "crossed")

# Instantiate the boundary conditions
noslip_domain = NoSlipDomain()
noslip_val = Constant((0.0, 0.0))
top_domain = Top()
top_val = Expression(("x[0]*(1.0 - x[0])", "0.0"))
pinpoint = PinPoint()
pin_val = Constant(0.0)

# Define the RHS
f = Constant((0.0, 0.0))
```

Figure 20.10: DOLFIN code for defining the lid-driven cavity domain.

Python code

```
# Define a high order approximation to the exact solution
u_ex = Expression(("sin(4*pi*x[0])*cos(4*pi*x[1])",
                   "-cos(4*pi*x[0])*sin(4*pi*x[1])"),
                  element=VectorElement("Lagrange", triangle,
                       10))
p_ex = Expression("pi*cos(4*pi*x[0])*cos(4*pi*x[1])",
                  element=FiniteElement("Lagrange", triangle,
                       10))

# Define the L2 error norm
M_u = inner((u_ex - u),(u_ex - u))*dx
M_p = (p_ex - p)*(p_ex - p)*dx

# Compute the integral
u_err = assemble(M_u, mesh=mesh)
p_err = assemble(M_p, mesh=mesh)

# Compute L2 error of the divergence
M_div = div(u)*div(u)*dx
div_err = assemble(M_div, mesh=mesh)
```

Figure 20.11: DOLFIN code for computing the error in the L^2 norm. The exact solution is interpolated using 10th order Lagrange polynomials on cells.

p	Crouzeix–Raviart	STAB	MINI
1	$2.01 \pm 1 \times 10^{-2}$	$1.93 \pm 6 \times 10^{-2}$	$2.03 \pm 6 \times 10^{-2}$
2	-	$3.02 \pm 2 \times 10^{-2}$	$2.77 \pm 5 \times 10^{-2}$
3	-	$4.00 \pm 1 \times 10^{-2}$	-
4	-	$4.99 \pm 4 \times 10^{-3}$	-
5	-	$5.98 \pm 1 \times 10^{-2}$	-
p	CD	Taylor–Hood	Scott–Vogelius
2	$2.15 \pm 1 \times 10^{-1}$	$3.02 \pm 2 \times 10^{-2}$	-
3	$3.11 \pm 2 \times 10^{-2}$	$3.98 \pm 1 \times 10^{-2}$	-
4	$4.07 \pm 1 \times 10^{-2}$	$4.99 \pm 1 \times 10^{-3}$	$5.00 \pm 2 \times 10^{-3}$
5	$5.10 \pm 5 \times 10^{-2}$	$5.97 \pm 1 \times 10^{-1}$	$5.97 \pm 1 \times 10^{-1}$

Table 20.3: The computed exponential of the convergence rates of the velocity; that is, q in $O(h^q)$ where h is the width of the mesh element. This error uses L^2 for the different elements with $q = p + 1$ being the optimal error rate with p as the order of the velocity field. Notice that the CD method, as theoretically expected, loses an order of convergence and the MINI element does not do well for the second order case.

20.4.2 Results

The observed convergence rates in the L^2 norm of the velocity field for the analytic case for each method, calculated from a series of refined meshes with n in $\{8, 16, 32, 64\}$, are presented in Table 20.3. The optimal rate is $q + 1$, which is observed for all formulations except the CD element. The CD element loses one order of convergence due to poor pressure approximation and failure to satisfy the LBB condition.

To further compare the methods, a number of error and performance measures for the case of a fourth-order velocity space with a suitably chosen pressure space are presented. The analytic test case is used. The Crouzeix–Raviart and MINI elements are only implemented for low-order bases. For the sake of comparison, they are computed on a finer mesh that has a comparable number of degrees of freedom to the fourth-order Taylor–Hood element. Figure 20.12 compares the L^2 error in the velocity for the different methods. The velocity approximation appears to converge for all elements. The L^2 error in the pressure is shown in Figure 20.13. Unpredictable behavior for the pressure is observed for the CD element, whereas convergence for the pressure field is observed for the other methods. The L^2 norm of the divergence of the velocity field is presented in Figure 20.14. The divergence error for the Crouzeix–Raviart and Scott–Vogelius methods are zero to within machine precision for all meshes, as predicted by theory. The divergence error for the MINI element is considerably greater than that for the other methods. Figure 20.15 presents the run-time for the various fourth-order cases. All run-times, using a 2.6 GHz Intel Xeon, measure the assembly and linear system solve time in the Python code. The time required for the code generation is assumed to be negligible since the generated code is cached and only affects the time for the first run of a simulation; our timings always come from the second run of the simulation. Run-times for the mixed elements scale with the number of degrees of freedom. The Scott–Vogelius method has better properties for iterative solvers, hence it may be attractive for large-scale problems despite the greater run-time relative to other methods for the small problems tested.

The L^2 norm of the divergence of the velocity for the lid-driven cavity problem is shown in Figure 20.16. Unlike for the already considered smooth test case, the divergence error for the CD, MINI, stabilized and Taylor-Hood elements does not decrease with mesh refinement. A divergence error persists around the pressure singularities at the corners of the lid, as is apparent in Figure 20.17 for the Taylor-Hood case. To solve the lid-driven cavity problem with the Scott–Vogelius method, the

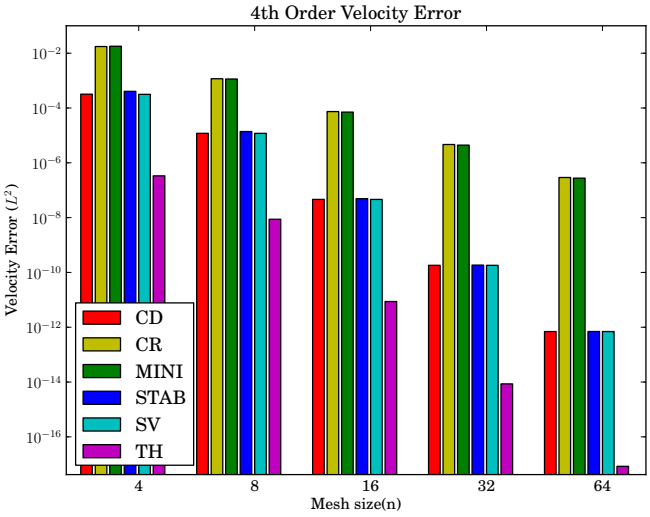


Figure 20.12: Velocity error of analytical test cases.

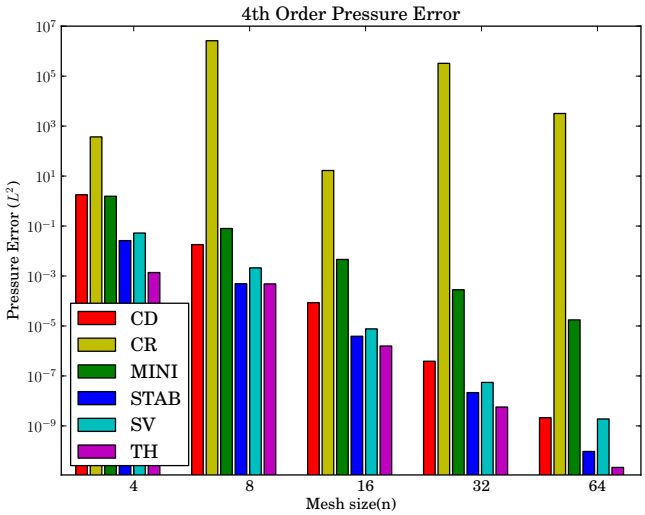


Figure 20.13: Pressure error of analytical test cases.

Figure 20.14: Divergence error of analytical test cases.

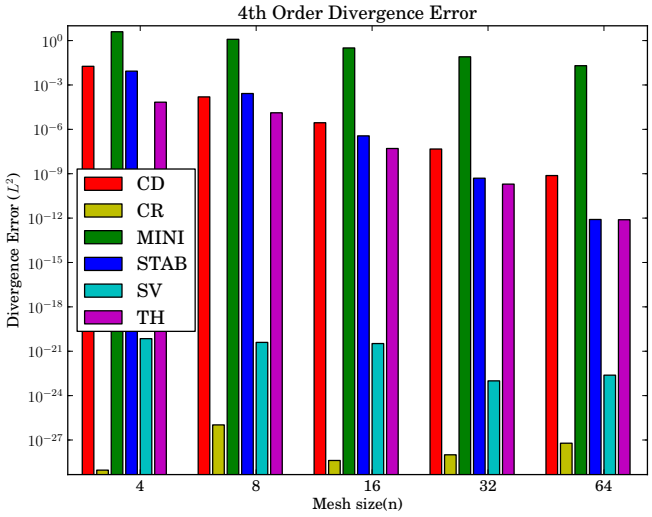
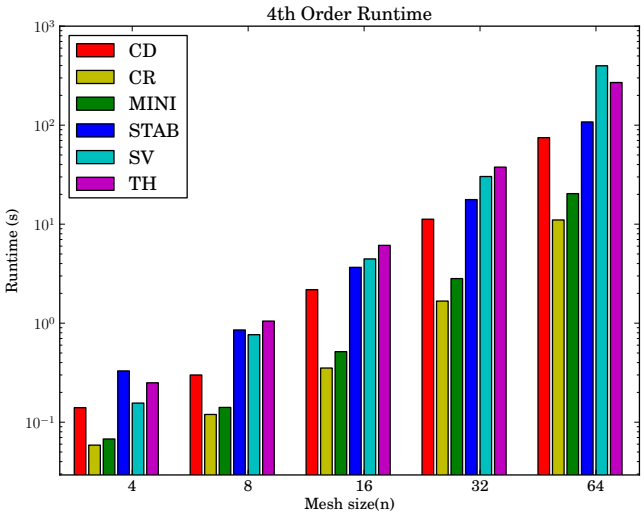


Figure 20.15: Run-times for the analytical test cases. All velocity spaces, except Crouzeix–Raviart and MINI, are fourth-order and pressure spaces are determined by the method. Crouzeix–Raviart and MINI are computed on a finer mesh with a similar number of degrees of freedom to the fourth-order Taylor–Hood method. CR, TH, and SV refer to Crouzeix–Raviart, Taylor–Hood and Scott–Vogelius, respectively.



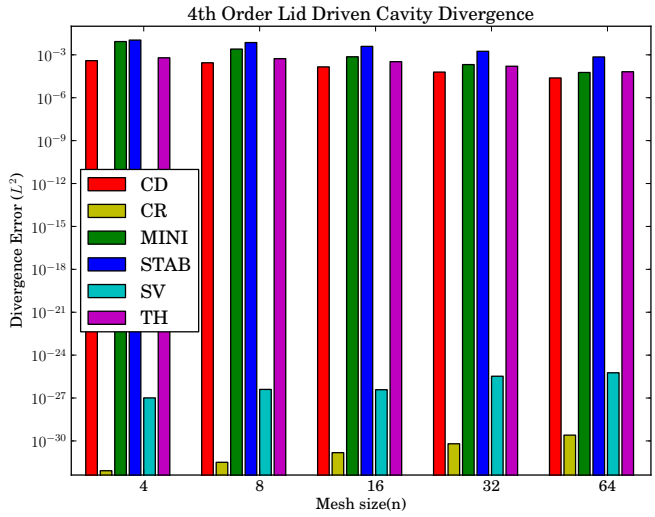


Figure 20.16: Divergence error of the lid-driven cavity test problem.

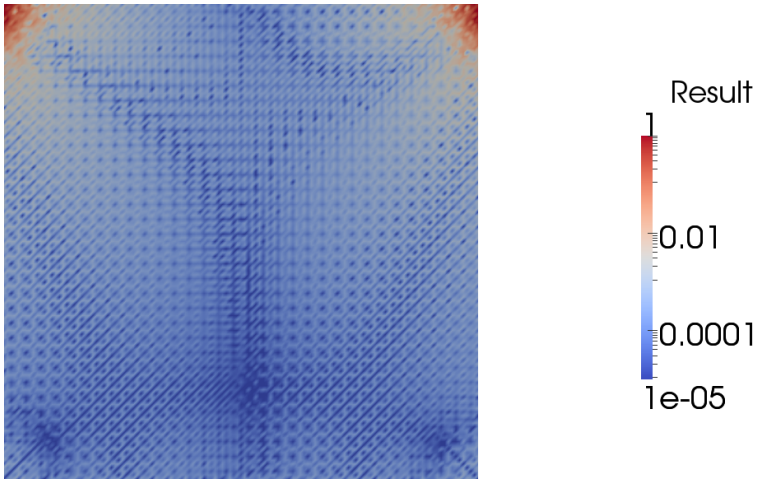


Figure 20.17: Local divergence error in lid-driven cavity using P_2–P_1 Taylor–Hood elements. Note in particular the large error at the corners with the lid.

penalty parameter had to be increased to 1×10^8 for the fixed point iteration to converge.

20.5 Conclusions

Comparisons between different finite elements for the Stokes problem have been presented. The flexibility afforded by automated code generation has been demonstrated via the ease with which solvers for a range of methods can be produced. The observed convergence rates for all cases are consistent with *a priori* estimates. Of the elements examined, the Crouzeix–Raviart element and the Scott–Vogelius lead to the smallest divergence error and Taylor–Hood the smallest velocity errors. If mass conservation properties are not crucial, the simplicity of elements such as the Taylor–Hood or STAB is attractive.

21 A comparison of finite element schemes for the incompressible Navier–Stokes equations

By Kristian Valen-Sendstad, Anders Logg, Kent-Andre Mardal, Harish Narayanan and Mikael Mortensen

Numerical algorithms for the computation of fluid flow have been an active area of research for several decades and still remain an important field to study. As a result, there exists a large literature on discretization schemes for the incompressible Navier–Stokes equations, and it can be hard to judge which method works best for any particular problem. Furthermore, since the development of any particular discretization scheme is often a long process and tied to a specific implementation, comparisons of different methods are seldom made.

FEniCS is a flexible platform for the implementation of different kinds of schemes based on finite element methods. To illustrate the simplicity by which different schemes can be implemented in FEniCS, we have implemented a test consisting of six distinct schemes. All schemes have been tested on six different test problems to compare their accuracy and efficiency. The schemes we have implemented are Chorin's projection scheme by Chorin (1968) and Temam (1969), the incremental pressure correction scheme (IPCS) by Goda (1979), the consistent splitting scheme (CSS) by Guermond et al. (2006), a least-squares stabilized Galerkin scheme (G2) by Hoffman and Johnson (2007), and a saddle point solver based on a Richardson iteration on the pressure Schur complement (GRPC) as described in Turek (1996).

All solvers and test problems have been implemented in Python (with a few C++ extensions) using DOLFIN. The source code for all solvers and test problems is available online[1] and can be used to reproduce all results shown in this chapter.

21.1 Preliminaries

We consider the incompressible Navier–Stokes equations with unit fluid density written in the form

$$\dot{u} + u \cdot \nabla u - \nabla \cdot \sigma = f, \tag{21.1}$$

$$\nabla \cdot u = 0, \tag{21.2}$$

where σ is the Cauchy stress tensor which for a Newtonian fluid is defined as

$$\sigma(u, p) = 2\nu\varepsilon(u) - pI. \tag{21.3}$$

[1] http://launchpad.net/nsbench/

Here, u is the unknown velocity vector, p is the unknown pressure, ν is the (kinematic) viscosity, f is the body force per unit volume, and $\epsilon(u)$ is the symmetric gradient:

$$\epsilon(u) = \frac{1}{2}(\nabla u + \nabla u^\top). \tag{21.4}$$

The above quantities σ and ϵ may be defined as follows in DOLFIN/UFL:

Python code

```
def epsilon(u):
    return 0.5*(nabla_grad(u) + nabla_grad(u).T)
```

Python code

```
def sigma(u, p, nu):
    return 2*nu*epsilon(u) - p*Identity(u.cell().d)
```

In all discretization schemes below, V_h and Q_h refer to the discrete finite element spaces used to discretize the velocity u and pressure p, respectively. For all schemes except the G2 scheme, V_h is the space of vector-valued continuous piecewise quadratic polynomials, and Q_h is the space of scalar continuous piecewise linear polynomials (Taylor–Hood elements). For the G2 scheme, continuous piecewise linears are used for both the velocity and the pressure. We will further use h to denote the local mesh size, $k_n = t_n - t_{n-1}$ to denote the size of the local time step, and $D_t^n u_h$ to denote the discretized form of the time derivative $(u_h^n - u_h^{n-1})/k_n$. For all schemes except the fully implicit schemes G2 and GRPC described below, the convective term is treated explicitly.

21.2 Implementation

We have implemented the solvers and test problems as two class-hierarchies in Python where the base classes are `SolverBase` and `ProblemBase`, respectively. The solvers derived from `SolverBase` implement the scheme; that is, they define the finite element spaces, assemble and solve linear systems, and perform time-stepping. Code from several solvers will be shown throughout this chapter. The problems derived from the `ProblemBase` class define the mesh, initial and boundary conditions, and other parameters.

A main script `ns` allows a user to solve a given problem with a given solver. All available problems and solvers may be listed by typing

Bash code

```
$ ns list
```

which results in the following output:

Bash code

```
Usage: ns problem solver

Available problems:

  drivencavity
  channel
  taylorgreen
  cylinder
  beltrami
  aneurysm
```

```
Available solvers:

  chorin
  css1
  css2
  ipcs
  g2
  grpc
```

The `ns` script accepts a number of optional parameters to enable refinement in space and time, storing the solution in VTK or DOLFIN XML format, computing stresses, or plotting the solution directly to screen. As an example, to solve the lid-driven cavity test problem using Chorin's method and plot the solution, one may issue the following command:

Bash code

```
$ ns drivencavity chorin plot_solution=True
```

Another script named `bench` allows a user to iterate over all solvers for a given problem, over all problems for a given solver, or over all problems and all solvers. As an example, the following command may be used to solve the channel test problem with all solvers on a mesh refined twice:

Bash code

```
$ bench channel refinement_level=2
```

21.3 Solvers

In this section, we present an overview of the six different schemes that have been tested.

21.3.1 Chorin's projection method

This scheme, often referred to as a non-incremental pressure correction scheme, was first proposed by Chorin (1968) and Temam (1969). For simplicity we will here refer to this scheme as *Chorin*. To solve the system of equations (21.1)–(21.2), the idea is first to compute a tentative velocity by neglecting the pressure in the momentum equation and then to project the velocity onto the space of divergence free vector fields. The projection step is a Darcy problem for u_h^n and p_h^n:

$$\frac{u_h^n - u_h^\bigstar}{k_n} + \nabla p_h^n = 0, \tag{21.5}$$

$$\nabla \cdot u_h^n = 0, \tag{21.6}$$

which is in fact reducible to a Poisson problem $-\Delta p_h^n = -\nabla \cdot u_h^\bigstar / k_n$ for the corrected pressure p_h^n. This is summarized in Scheme 1 and the implementation is shown in Figure 21.1. We note that since the velocity correction step is implemented as the solution of a linear system (involving a mass matrix that has not been lumped), the discrete incompressibility constraint is not satisfied exactly. On the other hand, the Dirichlet boundary conditions for the velocity are applied strongly as part of the velocity correction step and are thus satisfied exactly (at the nodal points).

21.3.2 Incremental pressure correction scheme (IPCS)

An improvement of the non-incremental pressure correction scheme is possible if the previous value for the pressure is used to compute the tentative velocity. This idea was first introduced by Goda

Scheme 1: Chorin's projection method

1. Compute a tentative velocity $u_h^\bigstar$ by solving

$$\langle D_t^n u_h^\bigstar, v\rangle + \langle u_h^{n-1}\cdot\nabla u_h^{n-1}, v\rangle + \langle \nu\nabla u_h^\bigstar, \nabla v\rangle = \langle f^n, v\rangle \quad \forall\, v\in V_h, \tag{21.7}$$

   including any boundary conditions for the velocity.

2. Compute the corrected pressure p_h^n by solving

$$\langle \nabla p_h^n, \nabla q\rangle = -\langle \nabla\cdot u_h^\bigstar, q\rangle/k_n \quad \forall\, q\in Q_h, \tag{21.8}$$

   including any boundary conditions for the pressure.

3. Compute the corrected velocity u_h^n by solving

$$\langle u_h^n, v\rangle = \langle u_h^\bigstar, v\rangle - k_n\langle \nabla p_h^n, v\rangle \quad \forall\, v\in V_h, \tag{21.9}$$

   including any boundary conditions for the velocity.

Python code

```
# Tentative velocity step
F1 = (1/k)*inner(u - u0, v)*dx \
   + inner(dot(u0, nabla_grad(u0)), v)*dx \
   + nu*inner(nabla_grad(u), nabla_grad(v))*dx \
   - inner(f, v)*dx
a1 = lhs(F1)
L1 = rhs(F1)

# Poisson problem for the pressure
a2 = inner(nabla_grad(p), nabla_grad(q))*dx
L2 = -(1/k)*nabla_div(us)*q*dx

# Velocity update
a3 = inner(u, v)*dx
L3 = inner(us, v)*dx - k*inner(nabla_grad(p1), v)*dx
```

Figure 21.1: Implementation of variational forms for the Chorin solver.

Scheme 2: Incremental pressure correction (IPCS)

1. Compute the tentative velocity $u_h^\bigstar$ by solving

$$\langle D_t^n u_h^\bigstar, v\rangle + \langle u_h^{n-1}\cdot\nabla u_h^{n-1}, v\rangle + \langle \sigma(u_h^{n-\frac{1}{2}}, p_h^{n-1}), \epsilon(v)\rangle + \langle p_h^{n-1} n, v\rangle_{\partial\Omega} - \langle \nu n\cdot(\nabla u_h^{n-\frac{1}{2}})^\top, v\rangle_{\partial\Omega} = \langle f^n, v\rangle \quad (21.10)$$

   for all $v \in V_h$, including any boundary conditions for the velocity. Here, $u_h^{n-\frac{1}{2}} = (u_h^\bigstar + u_h^{n-1})/2$.
2. Compute the corrected pressure p_h^n by solving

$$\langle \nabla p_h^n, \nabla q\rangle = \langle \nabla p_h^{n-1}, \nabla q\rangle - \langle \nabla\cdot u_h^\bigstar, q\rangle/k_n, \quad (21.11)$$

   including any boundary conditions for the pressure.
3. Compute the corrected velocity u_h^n by solving

$$\langle u_h^n, v\rangle = \langle u_h^\bigstar, v\rangle - k_n\langle \nabla(p_h^n - p_h^{n-1}), v\rangle \quad \forall\, v \in V_h, \quad (21.12)$$

   including any boundary conditions for the velocity.

(1979). The IPCS scheme is summarized in Scheme 2 and the implementation is shown in Figure 21.2. The IPCS scheme as implemented here also differs from the Chorin scheme in that the viscous term is evaluated at $(t_{n-1}+t_n)/2$ and a stress formulation is used in place of the Laplacian formulation used for the Chorin scheme. Note the importance of the term $\langle \nu n\cdot(\nabla u_h^{n-1/2})^\top, v\rangle_{\partial\Omega}$ which arises as a result of integrating the stress term by parts. Without this term an incorrect velocity profile is obtained at inlets and outlets where the velocity will tend to "creep" around the corners.

21.3.3 Consistent splitting scheme (CSS)

The consistent splitting scheme, as described in Guermond et al. (2006); Guermond and Shen (2003), is derived differently from the other splitting schemes and requires a more detailed description. The scheme is based on deriving an equation for the pressure p by testing the momentum equation (21.1) against ∇q. In combination with the incompressibility constraint, an equation for the pressure results. After solving for the pressure, the velocity is updated based solely on the momentum equation by an appropriate approximation (extrapolation) of the pressure. The derivation of the consistent splitting scheme is as follows. Multiply the momentum equation (21.1) by ∇q for $q \in H^1(\Omega)$ and integrate over the domain Ω to obtain $\langle \dot{u} + u\cdot\nabla u - \nu\Delta u + \nabla p, \nabla q\rangle = \langle f, \nabla q\rangle$. Since $\langle \dot{u}, \nabla q\rangle = \langle \nabla\cdot\dot{u}, -q\rangle + \langle \dot{u}, qn\rangle_{\partial\Omega} = 0$, it follows by (21.2) that

$$\langle \nabla p, \nabla q\rangle = \langle f - u\cdot\nabla u + \nu\Delta u, \nabla q\rangle, \quad (21.13)$$

if we assume that $\dot{u} = 0$ on $\partial\Omega$. Next, we use the identity $\Delta v \equiv \nabla\nabla\cdot v - \nabla\times\nabla\times v$ together with the incompressibility constraint (21.2) to write the diffusive term of (21.13) in *rotational form*:

$$\langle \nabla p, \nabla q\rangle = \langle f - u\cdot\nabla u - \nu\nabla\times\nabla\times u, \nabla q\rangle. \quad (21.14)$$

Python code

```
# Tentative velocity step
U = 0.5*(u0 + u)
F1 = (1/k)*inner(u - u0, v)*dx \
   + inner(dot(u0, nabla_grad(u0)), v)*dx \
   + inner(sigma(U, p0, nu), epsilon(v))*dx \
   + inner(p0*n, v)*ds \
   - beta*nu*inner(dot(n, nabla_grad(U).T), v)*ds \
   - inner(f, v)*dx
a1 = lhs(F1)
L1 = rhs(F1)

# Pressure correction
a2 = inner(nabla_grad(p), nabla_grad(q))*dx
L2 = inner(nabla_grad(p0), nabla_grad(q))*dx \
   - (1.0/k)*nabla_div(u1)*q*dx

# Velocity correction
a3 = inner(u, v)*dx
L3 = inner(u1, v)*dx - k*inner(nabla_grad(p1 - p0), v)*dx
```

Figure 21.2: Implementation of variational forms for the IPCS solver. The flag `beta = 1` is set to zero in the case when periodic boundary conditions are used.

This equation is the basis for the consistent splitting scheme. At this point, we may formulate the CSS scheme as the solution of the following pair of variational problems:

$$\langle D_t^n u_h, v\rangle + \langle u_h^{n-1}\cdot\nabla u_h^{n-1}, v\rangle + \langle \nu\nabla u_h^n, \nabla v\rangle - \langle p_h^\bigstar, \nabla\cdot v\rangle = \langle f^n, v\rangle, \tag{21.15}$$

$$\langle \nabla p_h^n, \nabla q\rangle = \langle f^n - u_h^{n-1}\cdot\nabla u_h^{n-1} - \nu\nabla\times\nabla\times u_h^n, \nabla q\rangle, \tag{21.16}$$

where $D_t^n u_h$ is an appropriate approximation of $\dot{u}_h$ and $p_h^\bigstar$ is an appropriate approximation of the pressure. In the simplest case, one may choose $p_h^\bigstar = p_h^{n-1}$ but higher order approximations are also possible. For example, one may take $p_h^\bigstar$ to be the linear extrapolation of p_h from p_h^{n-2} and p_h^{n-1} given by $p_h^\bigstar = p_h^{n-1} + (p_h^{n-1} - p_h^{n-2}) = 2p_h^{n-1} - p_h^{n-2}$. We will refer to the simplest approximation as CSS_1 and to the higher-order approximation as CSS_2.

To avoid computation of the term $\nabla\times\nabla\times u_h^n$ in (21.16), we take the inner product of (21.15) with ∇q and subtract the result from (21.16) to obtain

$$\begin{aligned}\langle \nabla p_h^n - \nabla p_h^\bigstar, \nabla q\rangle &= \langle D_t^n u_h - \nu\nabla\times\nabla\times u_h^n - \nu\Delta u_h^n, \nabla q\rangle \\ &= \langle D_t^n u_h - \nu\nabla\nabla\cdot u_h^n, \nabla q\rangle,\end{aligned} \tag{21.17}$$

where we have again used the identity $\Delta v \equiv \nabla\nabla\cdot v - \nabla\times\nabla\times v$. Finally, we define an auxiliary field $\psi_h^n = p_h^n - p_h^\bigstar + \nu\nabla\cdot u_h^n$ to write (21.17) in the form

$$\langle \nabla\psi_h^n, \nabla q\rangle = \langle D_t^n u_h, \nabla q\rangle. \tag{21.18}$$

The CSS scheme is summarized in Scheme 3/4.

To solve for the auxiliary variable ψ, appropriate boundary conditions must be used. Since ψ is a *pressure correction* and not the pressure itself, we use homogenized versions of the pressure boundary conditions which are zero at the boundary in the case of Dirichlet boundary conditions. This can be accomplished in DOLFIN using the function `homogenize`.

We remark that the derivation of the consistent splitting scheme is based on the assumption that

Scheme 3/4: Consistent splitting

1. Compute the pressure approximation (extrapolation) $p_h^\bigstar$ by

$$p_h^\bigstar = \begin{cases} p_h^{n-1}, & \text{for CSS}_1, \\ 2p_h^{n-1} - p_h^{n-2}, & \text{for CSS}_2. \end{cases} \tag{21.19}$$

2. Compute the velocity u_h^n by solving

$$\langle D_t^n u_h, v\rangle + \langle u_h^{n-1} \cdot \nabla u_h^{n-1}, v\rangle + \langle \sigma(u_h^{n-\frac{1}{2}}, p_h^\bigstar), \epsilon(v)\rangle + \langle \bar{p}n, v\rangle_{\partial\Omega} - \langle \nu n \cdot (\nabla \bar{u}_h^n)^\top, v\rangle_{\partial\Omega} = \langle f^n, v\rangle, \tag{21.20}$$

   including any boundary conditions for the velocity. Here, $u_h^{n-\frac{1}{2}} = (u_h^n + u_h^{n-1})/2$ and $\bar{p}$ is a given boundary condition for the pressure.

3. Compute the pressure correction ψ_h^n by solving

$$\langle \nabla \psi_h^n, \nabla q\rangle = \langle u_h^n - u_h^{n-1}, \nabla q\rangle / k_n - \langle u_h^n - u_h^{n-1}, qn\rangle_{\partial\Omega} / k_n \quad \forall\, q \in Q_h. \tag{21.21}$$

4. Compute the corrected pressure p_h^n by solving

$$\langle p_h^n, q\rangle = \langle p_h^\bigstar + \psi_h^n - \nu \nabla \cdot u_h^n, q\rangle \quad \forall\, q \in Q_h, \tag{21.22}$$

   including any boundary conditions for the pressure.

$\dot{u} = 0$ on $\partial\Omega$ which gives $\langle \dot{u}, \nabla q\rangle = -\langle \nabla \cdot u, q\rangle + \langle \dot{u}_{\partial\Omega}, qn\rangle = -\langle \nabla \cdot u, q\rangle$. For non-constant Dirichlet boundary conditions, this assumption is not valid. This issue is not addressed in Guermond and Shen (2003), but it is easy to add the missing term as shown in Figure 21.3 where the missing term is included in the linear form `L2`.

21.3.4 A least-squares stabilized Galerkin method (G2)

The G2 method is a stabilized finite element method using piecewise linear discretization in space and time. The version of G2 studied in this work is a simplified version of the G2 method presented in Hoffman and Johnson (2007) and identical to the version of G2 found in the latest stable release of Unicorn (version 0.1.0). For a discussion of the simplifications involved, see Section 21.6.3.

In each time step, the simplified G2 solution is defined by

$$\begin{aligned} \langle D_t^n u_h, v\rangle + \langle u_h^{n-\frac{1}{2}} \cdot \nabla u_h^{n-\frac{1}{2}}, v\rangle + \langle \sigma(u_h^{n-\frac{1}{2}}, p_h^n), \epsilon(v)\rangle - \langle \nu n \cdot (\nabla u_h^{n-\frac{1}{2}})^\top, v\rangle_{\partial\Omega} + \langle \bar{p}n, v\rangle_{\partial\Omega} \\ + SD_\delta = \langle f^n, v\rangle, \\ \langle \nabla p_h^n, \nabla q\rangle = -\langle \nabla \cdot u_h^n / \delta_1, q\rangle, \end{aligned} \tag{21.23}$$

for all $(v, q) \in V_h \times Q_h$, where $u^{n-\frac{1}{2}} = (u_h^n + u_h^{n-1})/2$ and

$$SD_\delta = \langle \delta_1 u_h^{n-\frac{1}{2}} \cdot \nabla u_h^{n-\frac{1}{2}}, u_h^{n-\frac{1}{2}} \cdot \nabla v\rangle + \langle \delta_2 \nabla \cdot u_h^{n-\frac{1}{2}}, \nabla \cdot v\rangle. \tag{21.24}$$

The G2 equations may be obtained by testing the incompressible Navier–Stokes equations against modified test functions $v \to v + \delta_1(u_h^{n-\frac{1}{2}} \cdot \nabla v + \nabla q)$ and $q \to q + \delta_2 \nabla \cdot v$ and dropping a number of

Python code

```
# Tentative pressure
if self.order == 1:
    ps = p1
else:
    ps = 2*p1 - p0

# Tentative velocity step
F1 = (1/k)*inner(u - u0, v)*dx \
   + inner(dot(u0, nabla_grad(u0)), v)*dx \
   + inner(sigma(u, ps, nu), epsilon(v))*dx \
   - beta*nu*inner(dot(n, nabla_grad(u).T), v)*ds \
   + inner(pbar*n, v)*ds \
   - inner(f, v)*dx
a1 = lhs(F1)
L1 = rhs(F1)

# Pressure correction
a2 = inner(nabla_grad(p), nabla_grad(q))*dx
L2 = (1/k)*inner(u1 - u0, nabla_grad(q))*dx \
   - (1/k)*inner(u1 - u0, q*n)*ds

# Pressure update
a3 = p*q*dx
L3 = p1*q*dx + psi*q*dx - nu*nabla_div(u1)*q*dx
```

Figure 21.3: Implementation of variational forms for the CSS solver(s). The flag `beta = 1` is set to zero in the case of periodic boundary conditions.

terms, including all stabilizing terms involving the time derivative $D_t^n u_h$. The stabilization parameters are set to $\delta_1 = \frac{\kappa_1}{2}(k_n^{-2} + |u^{n-1}|^2 h_n^{-2})^{-\frac{1}{2}}$ and $\delta_2 = \kappa_2 h_n$ in the convection dominated case; that is, if $\nu < uh$. In the diffusion dominated case, the parameters are set to $\delta_1 = \kappa_1 h_n^2$ and $\delta_2 = \kappa_2 h_n^2$. The constants κ_1 and κ_2 are here set to $\kappa_1 = 4$ and $\kappa_2 = 2$.

The discrete system of equations is solved by a direct fixed-point iteration between the velocity and pressure equations obtained by setting the test functions $q = 0$ and $v = 0$, respectively. Note that as a result of the stabilization, one obtains a Poisson equation for the pressure involving the stabilization parameter δ_1. The G2 scheme is summarized in Scheme 5 and the implementation is shown in Figure 21.4.

21.3.5 A saddle point solver for a pure Galerkin discretization (GRPC)

Finally, we test a scheme based on a pure space-time Galerkin finite element discretization of the incompressible Navier–Stokes equations and iterative solution of the resulting saddle point system. The saddle point system is obtained by testing the momentum equation (21.1) against a test function $v \in V_h$ and the continuity equation (21.2) against a test function $q \in Q_h$ and integrating over $\Omega \times [t_{n-1}, t_n]$. This corresponds to a space-time discretization using continuous piecewise quadratic and linear polynomials in space (for V_h and Q_h, respectively), and continuous piecewise linear polynomials in time (with discontinuous piecewise constant test functions in time). Integrating the

Scheme 5: G2

1. Compute stabilization parameters δ_1 and δ_2.
2. Repeat until convergence:
   (a) Update the pressure p_h^n by solving
   $$\langle \nabla p_h^n, \nabla q \rangle = -\langle \nabla \cdot u_h^n / \delta_1, q \rangle \quad \forall\, q \in Q_h, \tag{21.25}$$
   including any boundary conditions for the pressure.
   (b) Update the velocity u_h^n by solving
   $$\langle D_t^n u_h, v \rangle + \langle w \cdot \nabla u_h^{n-\frac{1}{2}}, v \rangle + \langle \sigma(u_h^{n-\frac{1}{2}}, p_h^n), \epsilon(v) \rangle - \langle \nu n \cdot (\nabla u_h^{n-\frac{1}{2}})^\top, v \rangle_{\partial\Omega} + \langle \bar{p} n, v \rangle_{\partial\Omega} + \langle \delta_1 w \cdot \nabla u_h^{n-\frac{1}{2}}, w \cdot \nabla v \rangle + \langle \delta_2 \nabla \cdot u_h^{n-\frac{1}{2}}, \nabla \cdot v \rangle = \langle f^n, v \rangle \tag{21.26}$$
   for all $v \in V_h$, including any boundary conditions for the velocity. Here, $u_h^{n-\frac{1}{2}} = (u_h^n + u_h^{n-1})/2$, $\bar{p}$ is a given boundary condition for the pressure, and w is an approximation of the velocity u_h^n from the previous iteration.
   (c) Compute a piecewise constant approximation w of u_h^n.
   (d) Compute the residuals of the momentum and continuity equations and check for convergence.

Figure 21.4: Implementation of variational forms for the G2 solver.

Python code

```
# Velocity system
U = 0.5*(u0 + u)
P = p1
Fv = (1/k)*inner(u - u0, v)*dx \
   + inner(dot(W, nabla_grad(U)), v)*dx \
   + inner(sigma(U, P, nu), epsilon(v))*dx \
   - beta*nu*inner(dot(n, nabla_grad(U).T), v)*ds \
   + inner(pbar*n, v)*ds \
   - inner(f, v)*dx \
   + d1*inner(dot(W, nabla_grad(U)), \
              dot(W, nabla_grad(v)))*dx \
   + d2*nabla_div(U)*nabla_div(v)*dx
av = lhs(Fv)
Lv = rhs(Fv)

# Pressure system
ap = inner(nabla_grad(p), nabla_grad(q))*dx
Lp = -(1/d1)*nabla_div(u1)*q*dx

# Projection of velocity
aw = inner(w, z)*dx
Lw = inner(u1, z)*dx
```

Scheme 6: GRPC

1. Repeat until convergence:
   (a) Assemble the residual vector R_U of the momentum equation.
   (b) Update the velocity vector U according to
   $$U := U - K^{-1}R_U. \tag{21.30}$$
   (c) Assemble the residual vector R_P of the continuity equation.
   (d) Update the pressure vector P according to
   $$P := P - \tau_1 L_1^{-1}R_P - \tau_2 L_2^{-1}R_P. \tag{21.31}$$

stress term by parts, one obtains the following variational problem: find (u_h^n, p_h^n) in $V_h \times Q_h$ such that

$$\langle D_t^n u_h, v\rangle + \langle u_h^{n-\frac{1}{2}} \cdot \nabla u_h^{n-\frac{1}{2}}, v\rangle + \langle \sigma(u_h^{n-\frac{1}{2}}, p_h^n), \epsilon(v)\rangle - \langle \nu n \cdot (\nabla u_h^{n-\frac{1}{2}})^\top, v\rangle_{\partial\Omega} + \langle \bar{p}n, v\rangle_{\partial\Omega} = \langle f^n, v\rangle, \tag{21.27}$$

$$\langle \nabla \cdot u_h^{n-\frac{1}{2}}, q\rangle = 0, \tag{21.28}$$

where $u_h^{n-\frac{1}{2}} = (u_h^n + u_h^{n-1})/2$ and $\bar{p}$ is a given boundary condition for the pressure. The resulting algebraic system of equations takes the form

$$\begin{bmatrix} M + \Delta t N(U) & \Delta t B \\ \Delta t B^\top & 0 \end{bmatrix} \begin{bmatrix} U \\ P \end{bmatrix} = \begin{bmatrix} b \\ 0 \end{bmatrix}, \tag{21.29}$$

where U and P are the vectors of degrees of freedom for u_h^n and p_h^n, respectively, M is the mass matrix, N is a convection–diffusion operator (depending on U^n), B is the discrete gradient, and b is a vector depending on the solution on the previous time step, body forces and boundary conditions. Notice that we have multiplied the incompressibility constraint by Δt to obtain symmetry in case when N is symmetric.

To solve this system of equations, we employ an algebraic splitting technique sometimes referred to as generalized Richardson iteration on the pressure Schur complement (GRPC) (Turek, 1999). The convergence of this method depends critically on the efficiency of two preconditioners, K and L. The preconditioner K should approximate $M + \Delta t N$, while L should approximate the pressure Schur complement $B^\top (M + \Delta t N)^{-1} B$. It is well known that if an explicit scheme is used for convection, then order-optimal solution algorithms for both $M + \Delta t N$ and $B^\top (M + \Delta t N)^{-1} B$ are readily available (Cahouet and Chabard, 1988; Turek, 1999; Mardal and Winther, 2004, 2011). In fact $L^{-1} \approx \Delta t M_Q^{-1} + A_Q^{-1}$, where M_Q and A_Q are the mass and stiffness matrices associated with the pressure discretization. Hence, we let $L_1 = \frac{1}{\Delta t} M_Q$ and $L_2 = A_Q$ and approximate L^{-1} by $\tau_1 L_1^{-1} + \tau_2 L_2^{-1}$. For simplicity, we here let $\tau_1 = \tau_2 = 2$. For a further discussion on these preconditioners, we refer to Chapter 35. In the implementation, we have chosen to exclude the convective term in the preconditioners K and L to avoid reassembly. The GRPC scheme is summarized in Scheme 6 and the implementation is shown in Figure 21.5.

Figure 21.5: Implementation of variational forms for the GRPC solver.

Python code

```
# Velocity and pressure residuals
U = 0.5*(u0 + u1)
P = p01
Ru = inner(u1 - u0, v)*dx \
   + k*inner(dot(U, nabla_grad(U)), v)*dx \
   + k*inner(sigma(U, P, nu), epsilon(v))*dx \
   - beta*k*nu*inner(dot(n, nabla_grad(U).T), v)*ds \
   + k*inner(pbar*n, v)*ds \
   - k*inner(f, v)*dx
Rp = k*nabla_div(U)*q*dx
```

Problems	Functionals / norms
Driven cavity, 2D	Minimum of stream function at $t = 2.5$
Channel flow, 2D	Velocity u_x at $(x, y) = (1, 0.5)$ at $t = 0.5$
Flow past a cylinder, 2D	Pressure difference across cylinder at $t = 8$
Taylor–Green vortex, 2D	Kinetic energy at $t = 0.5$
Beltrami flow, 3D	Relative L^2 error in velocity at $t = 0.5$
Idealized aneurysm, 3D	Velocity u_x at $(x, y, z) = (0.025, -0.006, 0)$ at $t = 0.05$

Table 21.1: Summary of test problems.

21.4 Test problems and results

To test the accuracy and efficiency of Schemes 1–6, we apply the schemes to a set of test problems. For each test problem, we make an *ad hoc* choice for how to measure the accuracy; we either measure the error in a certain functional of interest or a norm of the global error. The choice of test problems and functionals clearly affects the conclusions one may draw regarding the schemes. However, together the six test problems should give a good indication of the accuracy and efficiency of the tested schemes. We emphasize that all schemes have been implemented in the same framework and with minor differences in their implementation to make a fair comparison. All test problems represent laminar flow for small to moderate size Reynolds numbers in the range 1–1000. The test problems are listed in Table 21.1.

21.4.1 Common parameters

For all solvers, the time step is chosen based on an approximate CFL condition $k = 0.2\,h/U$ where U is an estimate of the maximum velocity.

Comparisons of solvers are made by plotting the CPU time / seconds and error against the number of degrees of freedom. Since all solvers except the G2 solver use the same type of discretization ($\mathcal{P}_2$–$\mathcal{P}_1$), this is equivalent to plotting CPU times and errors against refinement level or mesh size for those solvers. However, since the G2 method uses a $\mathcal{P}_1$–$\mathcal{P}_1$ discretization, the graphs will change depending on whether the x-axis is given by the number of degrees of freedom or the mesh size. In particular, the G2 method will seem slower (but at the same time more accurate) when plotting against the number of degrees of freedom, while seeming to be faster (but at the same time less accurate) when plotting against mesh size.

All simulations have been performed on a Linux cluster on a single node with 8 GB of memory. The test problems have been solved several times and the recorded CPU times have been compared with previous runs to ensure that the results are not influenced by any "noise".

To ensure accurate solution of linear systems, the absolute and relative tolerances for the DOLFIN

Python code

```
class BoundaryValue(Expression):
    def eval(self, values, x):
        if x[0] > DOLFIN_EPS and \
           x[0] < 1.0 - DOLFIN_EPS and \
           x[1] > 1.0 - DOLFIN_EPS:
            values[0] = 1.0
            values[1] = 0.0
        else:
            values[0] = 0.0
            values[1] = 0.0
```

Figure 21.6: Implementation of velocity boundary conditions for the driven cavity test problem.

(PETSc) Krylov solvers were set to 1e-25 and 1e-12, respectively. In all cases, the velocity system was solved using GMRES with ILU preconditioning, and the pressure system was solved using GMRES with an algebraic multigrid preconditioner (Hypre). For the iterative methods G2 and GRPC, the tolerance for the main iteration was set to a value between 1e-6 to 1e-12 with higher values in cases where the convergence was slow (or non-existent).

21.4.2 Driven cavity (2D)

A classical benchmark problem for fluid flow solvers is the two-dimensional lid-driven cavity problem. We consider a square cavity with sides of unit length and kinematic viscosity $\nu = 1/1000$. No-slip boundary conditions are imposed on each edge of the square, except at the upper edge where the velocity is set to $u = (1, 0)$. Figure 21.6 shows the implementation of these boundary conditions in DOLFIN. The initial condition for the velocity is set to zero. The resulting flow is a vortex developing in the upper right corner and then traveling towards the center of the square as the flow evolves.

As a functional of interest, we consider the minimum value of the stream function at final time $T = 2.5$. Reference values for this functional are available in Pandit et al. (2007), where a reference value of $\min \psi = -0.0585236$ is reported, and in Chudanov et al. (2007), where a value of $\min \psi = -0.058048$ is reported. These values differ already in the third digit. To obtain a better reference value, we have therefore computed the solution using the spectral element code Semtex (Blackburn, 2011; Blackburn and Sherwin, 2004) with up to 80×80 10^{th} order elements heavily refined in the area in the vicinity of the minimum of the stream function. The time-stepping for computing the reference solution was handled by a third order implicit discretization, and a very short time step was used to minimize temporal errors. The resulting reference value for the minimum of the stream function was $\min \psi = -0.061076605$. This value differs remarkably much from the available reference values in the literature, but seems to be correct judging from the convergence plots for the different solvers in Figure 21.8.

Computing the stream function. The stream function is defined by

$$u = \nabla \times \psi = (\frac{\partial \psi}{\partial y}, -\frac{\partial \psi}{\partial x}) \tag{21.32}$$

and can be computed by solving the Poisson problem

$$-\nabla^2 \psi = \omega, \tag{21.33}$$

Figure 21.7: Computing the stream function in DOLFIN.

Python code

```
# Define variational problem
psi = TrialFunction(V)
q   = TestFunction(V)
a   = dot(nabla_grad(psi), nabla_grad(q))*dx
L   = dot(u[1].dx(0) - u[0].dx(1), q)*dx

# Define boundary condition
g  = Constant(0)
bc = DirichletBC(V, g, DomainBoundary())

# Compute solution
psi = Function(V)
solve(a == L, psi, bc)
```

where ω is the vorticity given by

$$\omega = \nabla \times u = \frac{\partial u_y}{\partial x} - \frac{\partial u_x}{\partial y}. \tag{21.34}$$

For a more thorough description, see White (1999) or White (1991). Figure 21.7 shows how to compute the stream function in DOLFIN.

Results. Figure 21.8 shows the results for the driven cavity test problem. The smallest errors are obtained with the Chorin and GRPC schemes. The GRPC solver is also the slowest solver. We further observe a clear difference between CSS_1 and CSS_2.

21.4.3 Pressure-driven channel flow (2D)

As a second test problem, we seek the solution of the Navier–Stokes equations in a two-dimensional pressure-driven channel. The geometry of the channel is the unit square $[0,1]^2$ and the kinematic viscosity is $\nu = 1/8$. No-slip boundary conditions are applied to the velocity at the upper and lower walls, and Neumann boundary conditions are applied at the inlet and outlet. Dirichlet boundary conditions are applied to the pressure at the inlet and outlet, with $p = 1$ at the inlet and $p = 0$ at the outlet. The initial condition is $u = (0,0)$ for the velocity. As a functional of interest, we consider the x-component of the velocity at $(x,y) = (1,0.5)$ at final time $T = 0.5$. By a Fourier series expansion, it is easy to show that the exact value of the velocity at this point is given by

$$u_x(1,0.5,t) = 1 - \sum_{n=1,3,\ldots}^{\infty} \frac{32}{\pi^3 n^3} e^{-\frac{\pi^2 n^2 t}{8}} (-1)^{(n-1)/2}. \tag{21.35}$$

At final time $T = 0.5$, this values is $u_x(1,0.5,0.5) \approx 0.44321183655681595$.

Results. Figure 21.9 shows the results for the pressure-driven channel test problem. Again, the smallest error is obtained with the GRPC solver closely followed by the IPCS solver. The W-shaped curve for the G2 solver is an effect of the $\mathcal{P}_1$–$\mathcal{P}_1$ discretization which results in a vertex located at $(x,y) = (1,0.5)$ only for every other refinement level.

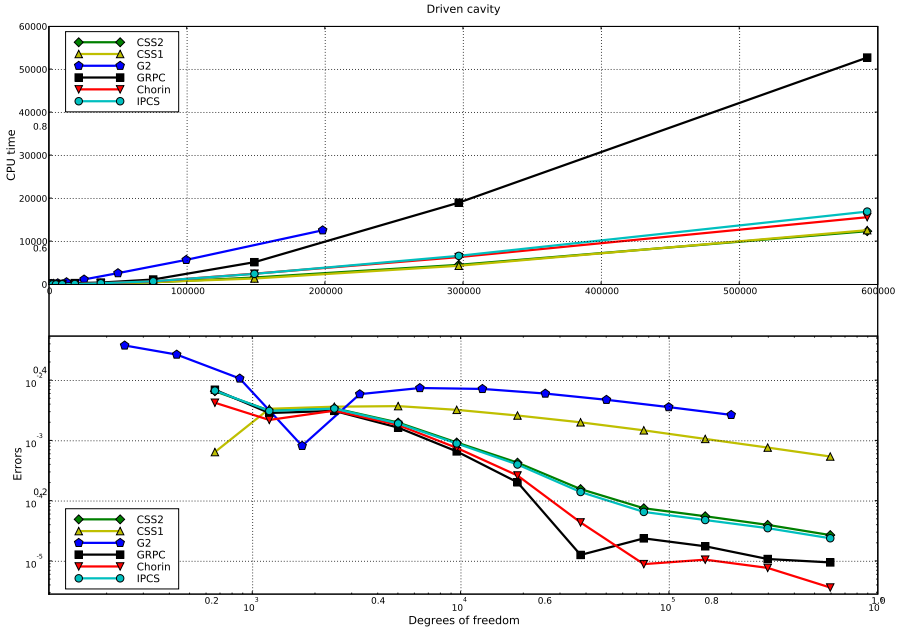


Figure 21.8: Results for the driven cavity test problem.

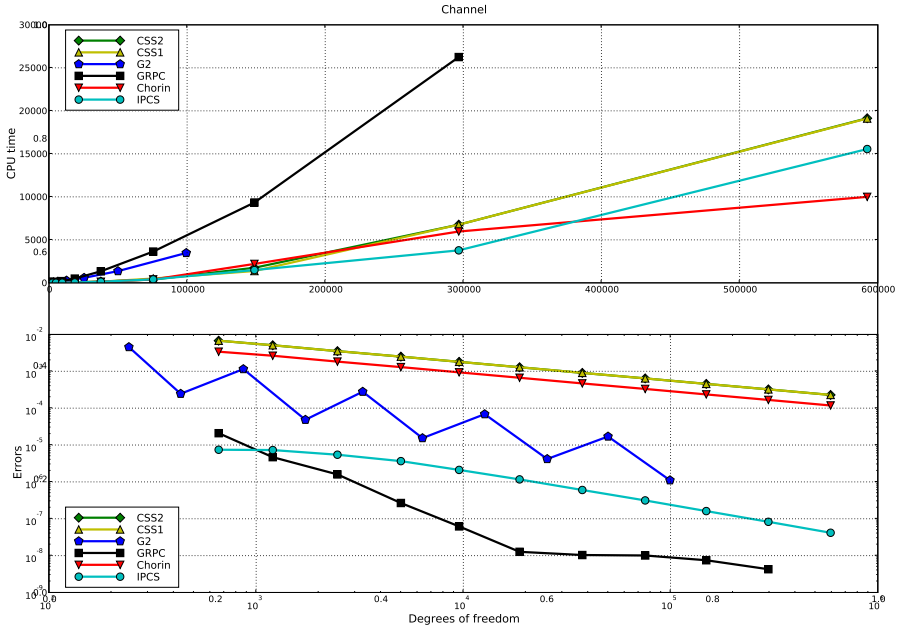


Figure 21.9: Results for the pressure-driven channel test problem.

Figure 21.10: Implementation of periodic boundary conditions for the Taylor–Green vortex test problem.

Python code

```
class PeriodicBoundaryX(SubDomain):
    def inside(self, x, on_boundary):
        return x[0] < (-1.0 + DOLFIN_EPS) and \
               x[0] > (-1.0 - DOLFIN_EPS) and \
               on_boundary

    def map(self, x, y):
        y[0] = x[0] - 2.0
        y[1] = x[1]

class PeriodicBoundaryY(SubDomain):
    def inside(self, x, on_boundary):
        return x[1] < (-1.0 + DOLFIN_EPS) and \
               x[1] > (-1.0 - DOLFIN_EPS) and \
               on_boundary

    def map(self, x, y):
        y[0] = x[0]
        y[1] = x[1] - 2.0
```

21.4.4 Taylor–Green vortex (2D)

As our next test problem, we consider the Taylor–Green vortex described in Canuto et al. (2007), which is a periodic flow with exact solution given by

$$\begin{aligned} u(x,y,t) =& (\cos(\pi x)\sin(\pi y)e^{-2t\nu\pi^2}, \cos(\pi y)\sin(\pi x)e^{-2t\nu\pi^2}), \\ p(x,y,t) =& -0.25(\cos(2\pi x) + \cos(2\pi y))e^{-4t\nu\pi^2}, \end{aligned} \tag{21.36}$$

on the domain $[-1,1]^2$. The kinematic viscosity is set to $\nu = 1/100$. Periodic boundary conditions are imposed in both the x and y directions. The implementation of these boundary conditions in DOLFIN is shown in Figure 21.10. The initial velocity and pressure fields are shown in Figure 21.11. As a functional of interest, we measure the kinetic energy $K = \frac{1}{2}\|u\|_{L^2}^2$ at final time $T = 0.5$.

Results. Figure 21.12 shows the results for the Taylor–Green test problem. The smallest error is obtained with the IPCS solver. For this test problem, the G2 solver is overly dissipative and produces an error which is six orders of magnitude larger than that of the IPCS solver.

21.4.5 Flow past a cylinder (2D)

We next consider a test problem from Turek (1996), which is a two-dimensional cylinder submerged into a fluid and surrounded by solid walls as illustrated in Figure 21.13. The cylinder is slightly displaced from the center of the channel, and the resulting flow is a vortex street forming behind the cylinder. No-slip boundary conditions are applied to the cylinder as well as the upper and lower walls of the channel. A zero Dirichlet boundary condition is imposed on the pressure at the outlet. The inflow velocity is a time-varying parabolic profile given by

$$u(0,y,t) = (4U_m y(H-y)\sin(\pi t/8)/H^4, 0), \quad t \leqslant 8, \tag{21.37}$$

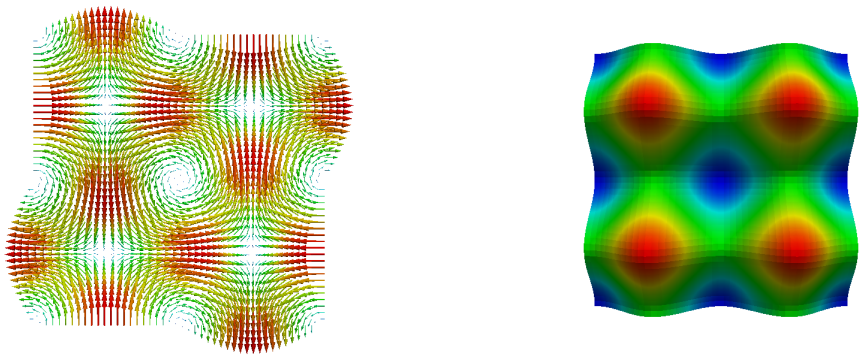


Figure 21.11: An illustration of the initial conditions for the Taylor–Green vortex test problem: the velocity field with vectors to the left and the corresponding pressure field to the right.

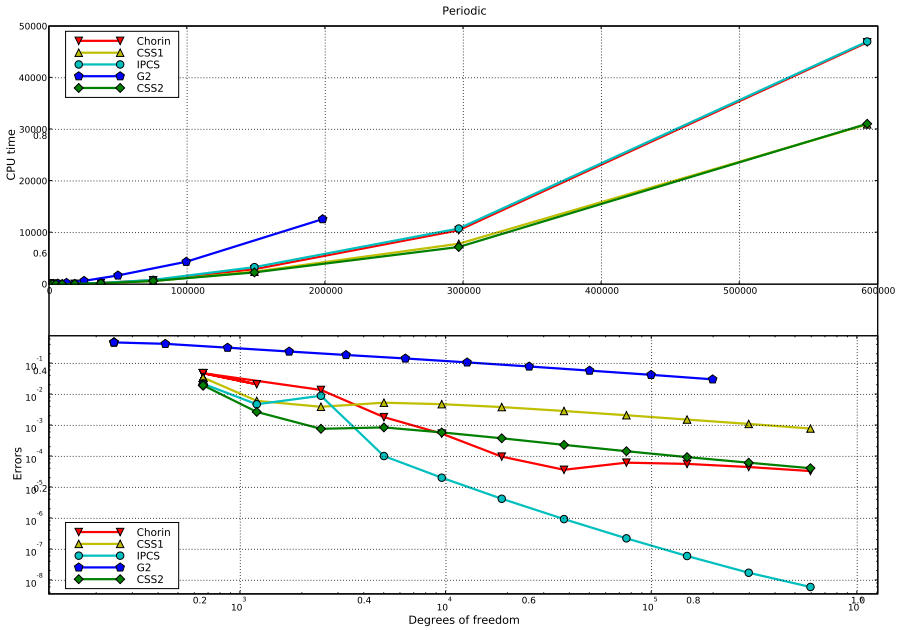


Figure 21.12: Results for the Taylor–Green vortex test problem.

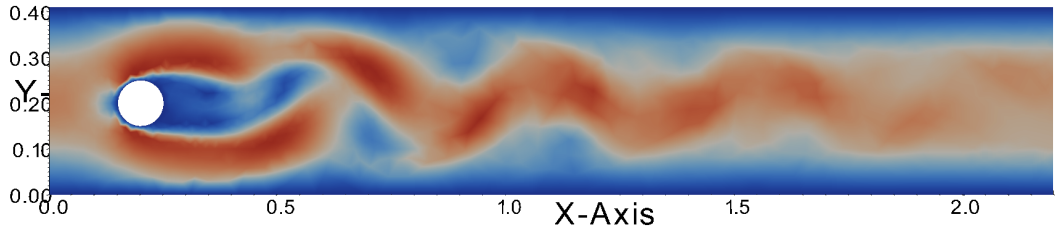


Figure 21.13: Illustration of the velocity field for the cylinder test problem at $t = 5$.

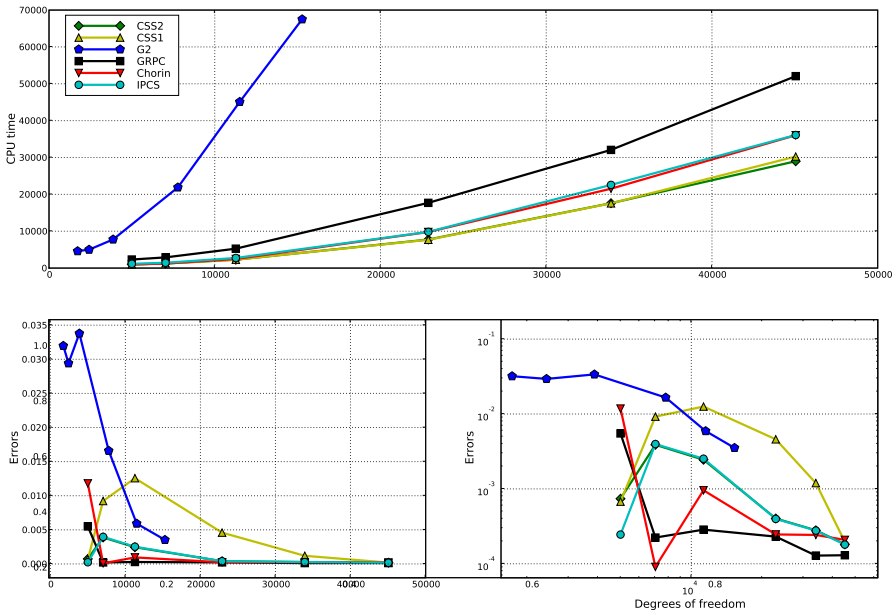


Figure 21.14: Results for the cylinder test problem.

where $U_m = 1.5$ and $H = 0.41$. The kinematic viscosity is $\nu = 1/1000$. As a functional of interest, we consider the pressure difference between the front and back of the cylinder at final time $T = 8$; that is,

$$\Delta p = p(0.45, 0.2, 8) - p(0.55, 0.2, 8). \tag{21.38}$$

A reference value -0.11144 for this functional was obtained using the IPCS solver on a mesh that was approximately of twice the size (in terms of the number of cells) as the finest mesh used in the test, with a time step of approximately half the size of the finest used time step.

Results. Figure 21.14 shows the results for the cylinder test problem. The smallest error is obtained with the GRPC solver closely followed by CSS_2 and IPCS. It is interesting to note that for this test problem, the CSS_2 solver is also the fastest.

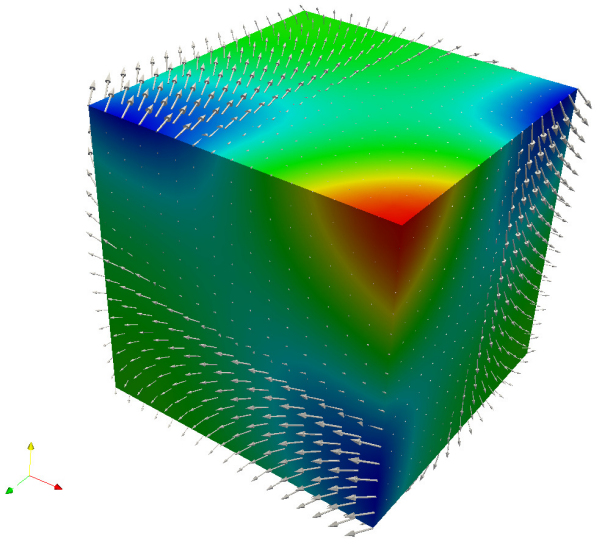


Figure 21.15: Solution of the Beltrami flow test problem.

21.4.6 Beltrami flow (3D)

We next consider a problem described in Ethier and Steinmann (1994), where an exact fully three-dimensional solution of the Navier–Stokes equations is derived. The flow is a so-called Beltrami flow, which has the property that the velocity and vorticity vectors are aligned. The domain is a cube with dimensions $[-1,1]^3$. The exact velocity is given by

$$\begin{aligned} u(x,y,z,t) &= -a[e^{ax}\sin(ay+dz) + e^{az}\cos(ax+dy)]e^{-d^2t}, \\ v(x,y,z,t) &= -a[e^{ay}\sin(az+dx) + e^{ax}\cos(ay+dz)]e^{-d^2t}, \\ w(x,y,z,t) &= -a[e^{az}\sin(ax+dy) + e^{ay}\cos(az+dx)]e^{-d^2t}, \end{aligned} \tag{21.39}$$

and the exact pressure is given by

$$\begin{aligned} p(x,y,z,t) = -a^2e^{-2d^2t}\left(e^{2ax}+e^{2ay}+e^{2az}\right)\Big(&\sin(ax+dy)\cos(az+dx)e^{a(y+z)} \\ &+ \sin(ay+dz)\cos(ax+dy)e^{a(x+z)} + \sin(az+dx)\cos(ay+dz)e^{a(x+y)}\Big). \end{aligned} \tag{21.40}$$

The solution is visualized in Figure 21.15. The constants a and d may be chosen arbitrarily and have been set to $a = \pi/4$ and $d = \pi/2$ as in Ethier and Steinmann (1994). The kinematic viscosity is $\nu = 1$. To measure the error, we compute the L^2 norm of the error in the velocity field at final time $T = 0.5$ divided by the L^2 norm of the exact solution as in Ethier and Steinmann (1994).

Results. Figure 21.16 shows the results for the Beltrami test problem. The smallest errors are obtained with the GRPC solver, while the largest errors are obtained with the CSS_1 solver.

21.4.7 Aneurysm (3D)

Finally, we consider an idealized model of an artery with a saccular aneurysm (see Chapter 23). The diameter of the artery is set to 4 mm and the length is set to 50 mm. The aneurysm is of medium size with a radius of 2.5 mm. Inserting the density and viscosity of blood and suitably scaling to dimensionless quantities, we obtain a kinematic viscosity of size $\nu = 3.5/(1.025 \cdot 10^3) \approx 3.4146 \cdot 10^{-6}$.

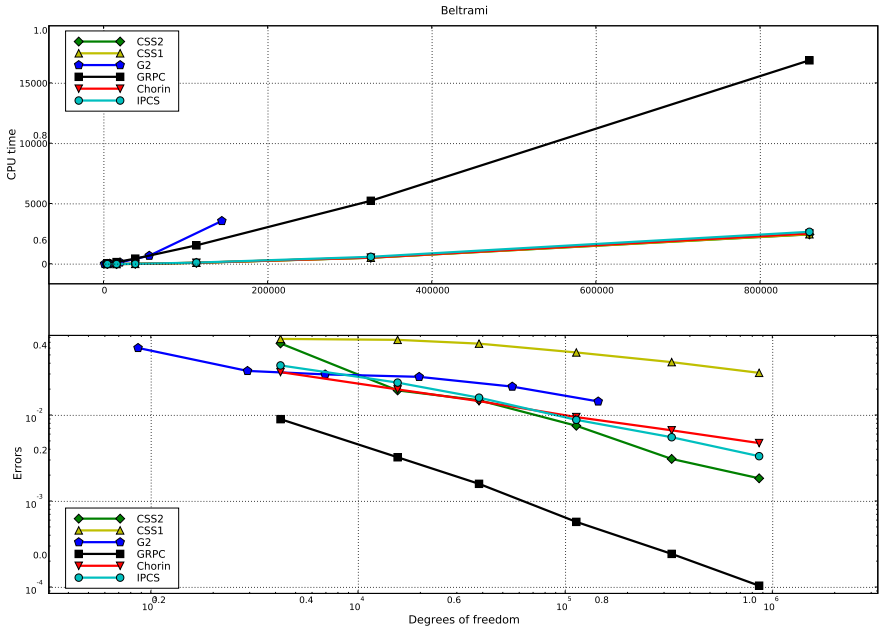


Figure 21.16: Results for the Beltrami flow test problem.

The geometry and flow at the final time $T = 0.05$ (ms) are shown in Figure 21.17. We impose no-slip boundary conditions on the vessel walls. At the inlet, we set the velocity to $u(x, y, z, t) = \sin(30t)\,(1 - (y^2 + x^2)/r^2)$ where $r = 0.002$ (mm). At the outlet, we enforce a zero Dirichlet boundary condition for the pressure. As a functional of interest, we consider the x-component of the velocity at the point $(x, y, z) = (0.025, -0.006, 0)$ (mm) located inside the aneurysm at final time $T = 0.05$ (ms). A reference value -0.0355 (mm/ms) for this functional was obtained using the IPCS solver on a fine mesh.

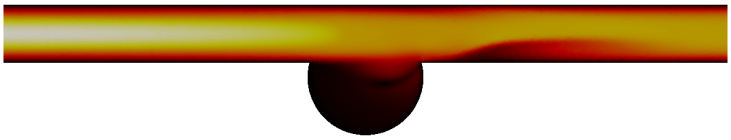
Results. Figure 21.18 shows the results for the aneurysm test problem. Reasonable convergence is obtained for all solvers except the G2 solver which does not seem to converge towards the computed reference value.

21.5 Summary of results

To summarize the results for all solvers and test problems, we plot all timings and errors in a single scatter plot. The rationale behind the plot is to get an indication of which solver(s) is the most accurate and efficient. Each data point in the plot is the result of solving one of the above test problems using one particular solver on one particular refinement level. To be able to compare different test problems (which vary in simulation time and size of error), the CPU time is scaled by the average CPU time for all solvers on each refinement level and the errors are scaled similarly. We also scale CPU times and errors by the number of degrees of freedom (total number of unknowns for both velocity and pressure). The resulting scatter plot is shown in Figure 21.19. An ideal solver (which is both fast and accurate) should be located in the lower left corner of this plot.

As can be seen in Figure 21.19, the Chorin, CSS_1 and CSS_2 solvers have an average performance

Figure 21.17: Velocity magnitude for the aneurysm test problem sliced at the center at final time $T = 0.05$ ms.



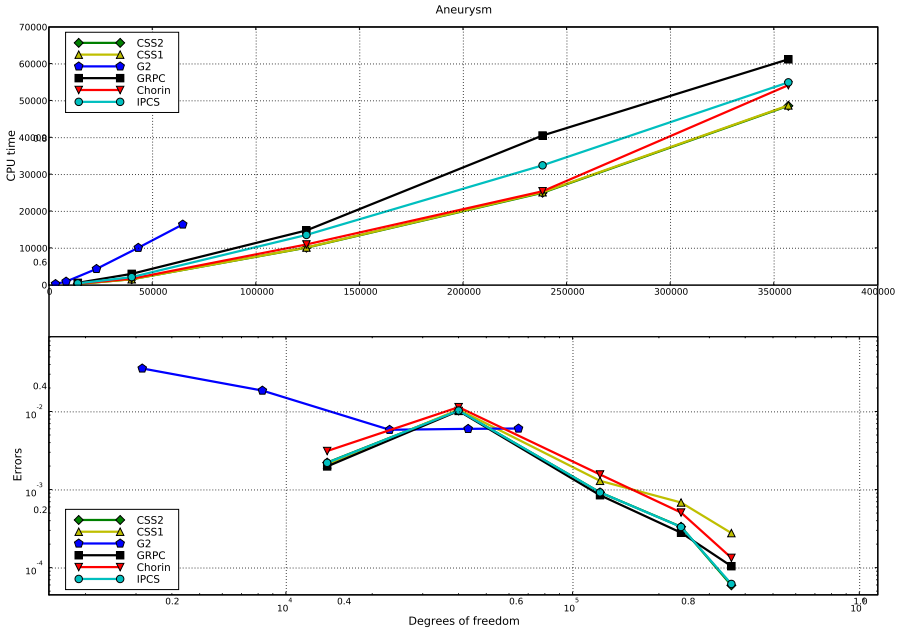


Figure 21.18: Results for the aneurysm test problem.

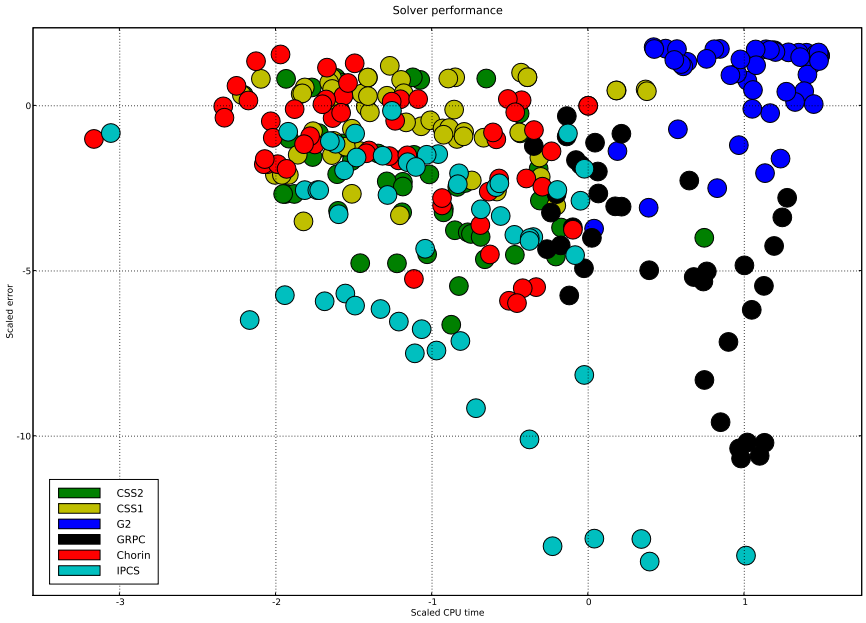


Figure 21.19: Scatter plot summarizing the results for all test problems and solvers (logarithmic scale).

and are mostly clustered around the center of mass of the scatter plot. The G2 solver is mainly located in the upper right corner. The results for the IPCS solver are less clustered, but it is the solver with most points located in the lower left corner. The GRPC solver is mostly located in the lower right corner of the scatter plot. This indicates that the GRPC solver is accurate but expensive.

21.6 Discussion

21.6.1 Numerical boundary layers

As pointed out in Guermond et al. (2006), the fractional step solvers are usually plagued by an artificial boundary layer, because the boundary condition $\nabla p_h^n \cdot n|_{\partial\Omega} = 0$ is enforced on the pressure. This "unphysical" Neumann boundary condition can create a numerical boundary layer simply because the velocity update $u_h^n = u^{n-1} - \Delta t \nabla p_h^n$ may lead to nonzero velocities in the tangential direction on no-slip walls (this follows since there is nothing preventing the pressure gradient from being nonzero in the tangential direction). However, in this work the velocity is being updated through a weak form where the no-slip boundary condition is strongly enforced. Therefore, the tangential velocity is set to zero and an artificial boundary layer is not observed in our simulations using the fractional step solvers Chorin and IPCS.

21.6.2 Time discretization

For the channel test problem, the convective term is zero and the discretization of the diffusive term is of particular importance. A formally second-order accurate in time Crank–Nicolson type scheme for the viscous term will in general improve the accuracy over the merely first-order explicit or fully implicit schemes. This is why the GRPC, IPCS and G2 solvers perform well on this problem. The

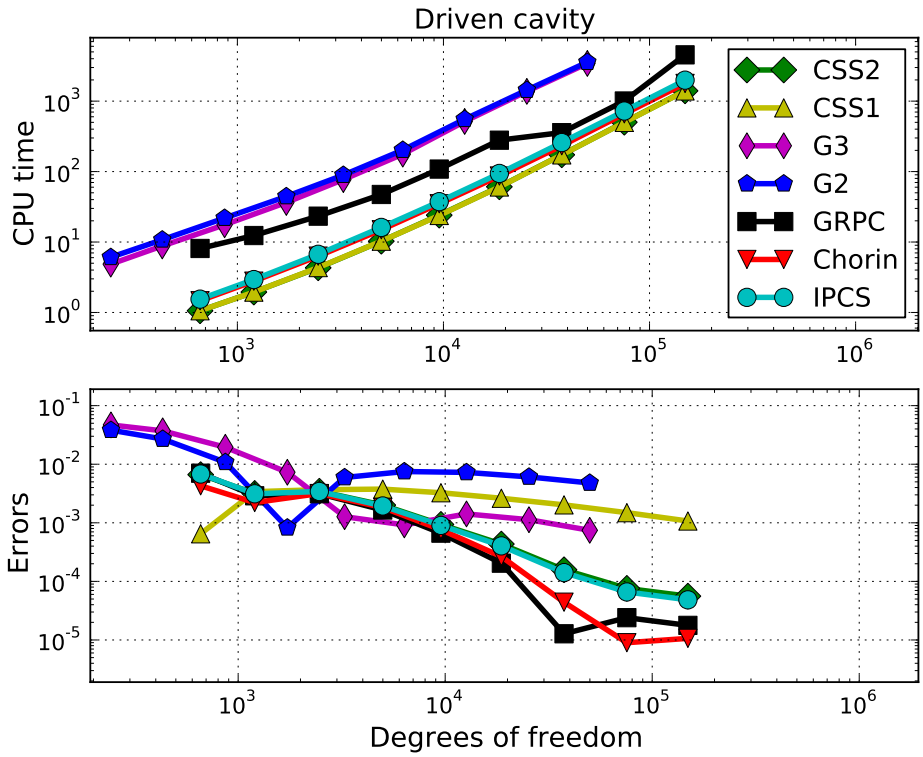


Figure 21.20: Updated results showing the speed and accuracy of the set of test solvers for the driven cavity test problem. In these plots, results have been included for an updated version of the G2 method, here referred to as "G3".

channel test problem is the problem where G2 performs best relative to the other solvers, which could also be attributed to the fact that both stabilization terms in the momentum equation of G2 are zero for this flow.

21.6.3 G2 vs. G2

The version of G2 described in Section 21.3.4 is modeled after the G2 version implemented in the latest available stable release of Unicorn at the time of writing (version 0.1.0). This version of G2 is a simplified version of G2 as presented in Hoffman and Johnson (2007). In particular, the simplified version does not include stabilization terms involving the right-hand side f, drops the stabilization term $\langle \delta_1 \nabla p_h^n, u_h^{n-\frac{1}{2}} \cdot \nabla v \rangle$ from the right-hand side of the velocity equation, and drops the stabilization term $-\langle \delta_1 u_h^{n-\frac{1}{2}} \cdot \nabla u_h^{n-\frac{1}{2}}, \nabla q \rangle$ from the right-hand side of the pressure equation. The authors of Unicorn have kindly provided an updated version of the G2 method including these terms. This updated solver has been added to the collection of benchmarks and solvers available at `http://launchpad.net/nsbench`. Inclusion of the missing terms improves the accuracy of the G2 method, as shown in Figure 21.20 and Figure 21.21. In these plots, the updated G2 method is referred to as "G3".

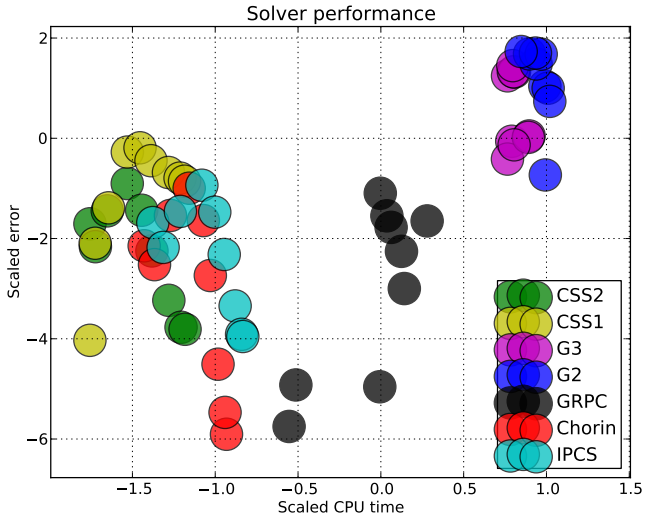


Figure 21.21: Updated results showing a scatter plot of the speed and accuracy of the set of test solvers for the driven cavity test problem. In these plots, results have been included for an updated version of the G2 method, here referred to as "G3".

21.7 Conclusions

From the scatter plot in Figure 21.19, we conclude that the IPCS solver is overall the most efficient and accurate method. Another advantage of the IPCS method is that it is easy to implement and does not require the iterative solution of a nonlinear system in each time step. The GRPC method (straightforward standard finite element Galerkin discretization) also obtains high accuracy, but does not deliver the same speed. It is possible that better tuning of the iterative solution of the saddle point system would change this picture. The same can be said for the G2 method, the speed of which will likely increase by adopting a more efficient iterative algorithm.

22 *Simulation of transitional flows*

By Mikael Mortensen, Kent-Andre Mardal and Hans Petter Langtangen

The purpose of this work is to validate Navier–Stokes (NS) solvers implemented in FEniCS for unstable, transitional flows. Solvers for the NS equations have been discussed in Chapter 21 for laminar flows. In this chapter, focus is put more directly on energy and energy conservation, features of primary importance in turbulence applications. We emphasize the treatment of the nonlinear convection term, where various forms (standard, divergence and skew-symmetric) are implemented and tested for both accuracy and stability. The algorithm chosen to advance the momentum and pressure in time is a fractional step approach that is memory efficient, but incurs a splitting error due to the uncoupling of the velocity and pressure. The significance of this splitting error is validated through comparison with a more accurate fully coupled solver that, due to its higher memory cost, is less suitable for large-scale turbulence applications. The performance of the solvers is validated with the one-dimensional Burgers' equation, the Orr–Sommerfeld perturbation in two dimensions and finally the three-dimensional unstable and transitional Taylor–Green vortex. All solvers and problems can be found as part of the openly distributed CBC.PDESys package, see also Mortensen et al. (2011b).

22.1 Background

The Navier–Stokes (NS) equations represent a differential form of the principle of conservation of mass and momentum. They govern both laminar and turbulent fluid motion in three-dimensional space and time for incompressible and compressible fluids. There are generally no closed form analytical solutions to the NS equations and the study of fluid dynamics thus relies heavily on numerical solutions.

For incompressible Newtonian fluids, the NS equations read

$$\frac{\partial u}{\partial t} + u \cdot \nabla u = \nu \nabla^2 u - \frac{1}{\rho} \nabla p + f, \tag{22.1}$$

$$\nabla \cdot u = 0. \tag{22.2}$$

Here u is the velocity vector, $\nu = \mu/\rho$ is the kinematic viscosity, where ρ is the density and μ is the molecular viscosity, p is the pressure, and volumetric body forces are represented by f In the absence of viscosity the principle of energy conservation can also (in addition to mass and momentum) be directly imposed on the NS equations. This particular property is especially important for turbulent flows, since a fundamental feature of turbulence is that kinetic energy is extracted from the flow system and eventually converted into internal energy (heat) by the action of viscosity (rate of dissipation). The largest and most energetic turbulence structures are primarily responsible for efficient mixing of momentum and other scalar quantities. These structures are by a series of instability processes broken down to smaller and smaller spatial scales and eventually dissipated into heat. The conservation of

kinetic energy (in the absence of viscosity) is thus an important feature of turbulent fluid flows which is formally consistent with the NS equations. Unfortunately, though, this feature is not necessarily retained by the numerical scheme used to solve the discretized NS equations numerically. A numerical method can be both dissipative and dispersive, recognized for example by the order of the derivative in the truncation error of Taylor expansions. A numerical scheme with even order derivatives in truncated terms is dissipative, whereas odd derivatives lead to dispersion.

The often used terminology Direct Numerical Simulations (DNS) is understood as the three-dimensional and time dependent numerical simulations of the NS equations that resolve all all turbulence scales and that have negligible numerical dissipation (artificial viscosity) and dispersion. For this reason, DNS are often performed with highly accurate spectral methods (Canuto et al., 2007) in homogeneous flows or higher-order central finite differences or spectral element methods (Blackburn, 2011) for more geometrical flexibility in inhomogeneous flows. The results of carefully executed DNS have in the fluid mechanics community the same status as carefully executed experiments. Unfortunately, DNS are very demanding of computer resources. A good part of the expense is incurred in capturing the smallest scales of turbulence; that is, the scales that are responsible for dissipating energy. Yet another complication in non-periodic flows is to describe inflow and outflow boundary conditions that are consistent with the NS equations.

The computational cost of DNS can (at the expense of accuracy in computed statistics) be reduced by capturing only the largest scales and using a dissipative model in place of the smaller eddies (to compensate for the loss of accuracy). This method is referred to as Large Eddy Simulation (LES) and it too requires a three-dimensional and time-dependent solution. The dissipative model introduces into the NS equations something that is no longer physically exact. However, to the extent that the dissipative model does not contaminate the large scales, LES can provide NS simulations from which statistics may be obtained with satisfactory accuracy for many purposes. However, the results depend inherently on the grid, because the grid-independent solution is nothing but the DNS solution that one in most cases cannot afford. The art of LES is to find the best possible compromise between efficiency and accuracy.

It should be mentioned that some practitioners of LES use numerical dissipation to model the unresolved physical dissipation (see the review of implicit LES given by Grinstein et al. (2007)). In this paper, though, we will only consider numerical schemes with little or no dissipation applicable for DNS and regular LES.

Turbulent flows and the physical mechanisms responsible for the transition to turbulence from a laminar flow are not very well understood and have been researched extensively. At the turn of the 19th century, Osbourne Reynolds discovered that for cylindrical pipes the transition to turbulence occurred at a Reynolds number of 2300 ($Re = Uh/\nu$, where U is the average velocity and h is half the pipe diameter). Later, with carefully executed experiments in smooth pipes scientists have been able to increase this number considerably, revealing that velocity is not really the triggering factor, even though there clearly is a strong correlation (which follows since as the Reynolds number increases, the stabilizing viscous damping term becomes comparatively less than the unstable nonlinear convection term). Another example of transition can be found in the wakes downstream of bluff bodies placed in an incoming laminar flow. Here the transition is promoted by the strong shear layer formed by the recirculation region downstream of the body. In any case, in order for transition to occur, imposed disturbances triggered by obstacles, sudden pressure fluctuations, or even a sound waves, must grow and become unstable and finally chaotic. By introducing systematic perturbations of the NS equations one can study these phenomena and watch how they experience resonance and grow or gracefully die. Here, the numerical scheme will be of utmost importance because a dissipative scheme will damp (kill) the imposed perturbations.

The most famous early work aimed to study perturbations of the NS equations was conducted more

than 100 years ago by William McFadden Orr and Arnold Sommerfeld. The epitome of their analytical work is the celebrated Orr–Sommerfeld equation, which is an eigenvalue problem describing the linear two-dimensional modes of disturbance to a viscous parallel shear flow. Although the Orr–Sommerfeld equation only represents one simplified class of laminar-to-turbulence transition, it nevertheless constitutes a powerful method to assess numerical schemes since it provides an analytical transient solution to the NS equations that remains non-trivial for long integration times. The Orr–Sommerfeld test case will be further discussed in Section 22.3.2, but first we need to turn our attention to the NS solvers, the numerical methods, and their implementation in FEniCS.

22.2 Numerical method and energy conservation

In this section we will discuss both the spatial and temporal discretizations of the Navier–Stokes (NS) equations, and special attention will be focused on the nonlinear convection term. Furthermore, since the NS equations represent a system of equations, we will discuss both a fully coupled method where u and p are solved simultaneously and a fractional step method that solves for the pressure and velocity in a segregated manner. We also outline the implementation in FEniCS, and some optimization techniques that can speed up the code significantly.

22.2.1 Convection

Let $\Omega \subset \mathbb{R}^d$ be an open and bounded region in $\mathbb{R}^d$, where d is the number of spatial dimensions, with smooth boundary Γ. The L^2 inner product of fields on Ω is denoted as

$$\langle a, u \rangle = \int_\Omega a \cdot u \, \mathrm{d}x, \tag{22.3}$$

where a and u are arbitrary vector fields on Ω. Furthermore, let L^2 be the space of square integrable functions, and we denote the space of divergence-free vector fields by Z.

Let the convective transport of any vector field be written in general form as $B(u, a)$. Here, u is the *convecting velocity*, while a is the *convected vector field*. Then the standard convective term,

$$B(u, a) = u \cdot \nabla a \tag{22.4}$$

can be multiplied by the vector b and integrated by parts to yield

$$\langle B(u, a), b \rangle = -\langle a, B(u, b) \rangle - \langle B(a, u), b \rangle + \int_\Gamma (b \cdot a)\,(u \cdot n)\; d\Gamma. \tag{22.5}$$

If we for simplicity assume homogeneous Dirichlet boundary conditions the last term vanishes. Furthermore, if the velocity is divergence-free ($u \in Z$), the following result can be obtained for the standard convection form

$$\langle B(u, a), b \rangle = -\langle a, B(u, b) \rangle. \tag{22.6}$$

This equation implies that if the standard convective form is adopted and $\nabla \cdot u = 0$, then

$$\langle B(u, a), a \rangle = 0 \tag{22.7}$$

for any choice of a (follows by setting $b = a$ in (22.6)). This is an important and necessary result for conservation of kinetic energy. This observation is perhaps more transparent if we rewrite the

convective term to show that it in fact represents transport of kinetic energy:

$$(B(u,u),u) = \int_\Omega (u \cdot \nabla u) \cdot u \, \mathrm{d}x = \int_\Omega u \cdot \nabla K(u) \, \mathrm{d}x = B(u, u \cdot u), \tag{22.8}$$

where $K(u)$ is the (specific) kinetic energy of the fluid flow and is defined as

$$K(u) = \frac{1}{2} u \cdot u. \tag{22.9}$$

The result in (22.8) means that the integral contribution from the convection of momentum to the accumulation of kinetic energy will be zero.

There are several alternative representations of the convective term. The divergence form

$$B(u,a) = \nabla \cdot (u \otimes a) \tag{22.10}$$

follows from the standard format by utilizing the divergence constraint. The well-known skew-symmetric (or just skew) form is a combination of the standard and divergence forms

$$B(u,a) = \frac{1}{2} \left[u \cdot \nabla a + \nabla \cdot (u \otimes a) \right]. \tag{22.11}$$

It can easily be shown, by multiplying (22.11) with a and integrating by parts, that the skew-symmetric form of (22.11) ensures that (22.7) holds for any (smooth) velocity field, and not just the divergence-free $u \in Z$. This is an important result, because in fractional step (projection) methods for the NS equations the divergence constraint is not always fulfilled, at least not for intermediate velocity fields. With the skew-form it is ensured that this divergence flaw does not propagate and contaminate the (of primary importance) kinetic energy of the flow.

22.2.2 Kinetic energy

A dynamic equation for K can be derived from (22.1) by taking the scalar product of the momentum equation and u, and thereafter rearranging using the divergence constraint to arrive at

$$\frac{\partial K(u)}{\partial t} + \nabla \cdot [u K(u)] = \nu \nabla^2 K(u) - \nu \nabla u : \nabla u - \frac{1}{\rho} \nabla \cdot (up) + f \cdot u. \tag{22.12}$$

The second term on the right-hand side represents dissipation of kinetic energy. The role of the remaining terms (neglecting body forces) is to transport $K(u)$ within the computational domain. This is made clear if (22.12) is integrated over the domain, neglecting body forces and making use of boundary conditions (all terms that can be written as divergences will then vanish, because of the divergence theorem). The well-known identity for the rate of change of total kinetic energy is obtained (see Simo and Armero (1994)):

$$\frac{\mathrm{d}\bar{K}}{\mathrm{d}t} = -\nu \int_\Omega \nabla u : \nabla u \, \mathrm{d}x, \quad \bar{K} = \int_\Omega K(u) \, \mathrm{d}x. \tag{22.13}$$

Evidently, since $\nu \geq 0$, energy should only decay and not be created within the domain. For inviscid flows (Euler equations) $\nu = 0$, transport is merely through the convective term and $\mathrm{d}\bar{K}/\mathrm{d}t = 0$. This means that as a consequence of NS equations, energy should be conserved through convective transport and dissipated only through the action of viscosity.

22.2.3 Nature of discretization schemes

There are numerous examples of numerical schemes that dissipate energy. The most familiar in fluid mechanics are probably the stabilizing upwinding-schemes (favored in many commercial software packages for their robustness) and streamline diffusion methods in finite element formulations. In general, numerical schemes that are asymmetric about the grid point (like upwind schemes) are known to be both dissipative and dispersive, whereas central (symmetric) schemes are non-dissipative, yet dispersive. Understanding the fundamental effects of dispersion and dissipation/diffusion and their relation to numerical discretizations is a key issue when performing the investigations of the present chapter. We shall therefore devote some space to illustrate the basic mechanisms, which can be conveniently done by studying a one-dimensional conservation equation

$$\frac{D\phi}{dt} = \frac{\partial \phi}{\partial t} + v\frac{\partial \phi}{\partial x} = 0, \tag{22.14}$$

where v is constant velocity. The initial condition is

$$\phi(x,0) = f(x). \tag{22.15}$$

Equation (22.14) expresses non-dissipative transport in the direction of the x axis (if $v > 0$), which means that the initial shape just moves with velocity v:

$$\phi(x,t) = f(x - vt). \tag{22.16}$$

An energy measure $\int_{-\infty}^{\infty} \phi^2 \, \mathrm{d}x$ remains constant in time.

We can build an arbitrary shape of ϕ as a Fourier series and study the behavior of one Fourier component. A complex Fourier mode $\phi = A \exp\left(ik(x - ct)\right)$ is a solution of (22.14) for an arbitrary amplitude A and frequency k, provided $c = v$. All such components move with constant velocity v and the energy of each component is constant in time.

Many finite difference schemes for (22.14) also allow Fourier components as solutions. More precisely, we have

$$\phi_j^n = A \exp\left(i(kx - \tilde{c}t)\right), \tag{22.17}$$

where j denotes a grid point on the x axis and n denotes the time level. The numerical wave velocity $\tilde{c} \neq v$ is a function of k, Δt, and Δx. When $\tilde{c}$ is real, but deviates from the exact value v, the Fourier component moves with slightly the wrong velocity. This dispersion error gives rise to a change of shape of the solution when we sum all components. If $\tilde{c}$ is complex, the imaginary value will lead to an effective amplitude that either grows or decreases in time. A growth will make the solution arbitrarily large for some large t, which is unphysical and hence ruled out as an unstable numerical scheme. A decrease in amplitude can be tolerated physically, but the discrete energy $\sum_j |\phi_j^n|^2$ decreases in time and the wave is said to dissipate. For this model problem, the error in $\tilde{c}$ usually depends on the non-dimensional Courant number $C \equiv v\Delta t/\Delta x$.

Using a central difference scheme in space and time for (22.14) results in a real $\tilde{c}$ if $C \leqslant 1$. For $C < 1$ the scheme is dispersive, but the discrete energy is conserved in time. Choosing $C > 1$ gives a complex $\tilde{c}$ and a growing discrete Fourier component, which implies $C \leqslant 1$ for numerical stability.

Looking at a forward scheme in time and upwind scheme in space; that is, two asymmetric differences, the numerical wave velocity $\tilde{c}$ becomes complex: $\tilde{c} = \tilde{c}_r + i\tilde{c}_i$. The value of c_r deviates from the exact velocity v, implying dispersion, while the imaginary value c_i is positive, giving rise to a decreasing amplitude $A \exp\left(-ikc_i n\Delta t\right)$ in time. This is a dissipative effect of the asymmetric difference(s). With a decreasing amplitude of the various Fourier components, the integral of the squared numerical solution will naturally decrease, and energy is lost.

22.2.4 A generic Navier–Stokes discretization

For the reasons explained above, central difference schemes are usually favored in the solution of the chaotic and transient velocity fields governed by the NS equations. Upwind schemes or streamline diffusion, on the other hand, are often used for Reynolds Averaged Navier–Stokes (RANS) equations, where the kinetic energy transport is solved for via a separate PDE and not implied by the computed deterministic mean velocity field. A Galerkin finite element method, where the basis functions of the test and trial spaces are the same (modulo boundary conditions), will produce discrete equations that correspond to central differencing in space. Therefore, we employ the standard Galerkin method for spatial discretization. For the temporal discretization we follow Simo and Armero (1994) and describe a family of solvers for the transient NS equations.

Let $t_n \in \mathbb{R}^+$ denote a discrete point in time. The velocity $u_{n-1} = u(\mathrm{x}, t_{n-1})$ is known, and we wish to advance the solution to $u_n = u(\mathrm{x}, t_n)$. To this end we use the following general algorithm

$$\frac{u_n - u_{n-1}}{\Delta t} = -B(\tilde{u}, \bar{u}) + \nu \nabla^2 u_{n-\alpha} - \nabla p_{n-1/2} + f_{n-\alpha} \tag{22.18}$$

$$\nabla \cdot u_{n-\alpha} = 0, \tag{22.19}$$

where

$$u_{n-\alpha} = (1-\alpha)u_n + \alpha u_{n-1}, \tag{22.20}$$

and $\alpha \in [0,1]$. The idea is to discretize all terms at the time level $n-\alpha$. The nature of the time difference over the interval $\Delta t = t_n - t_{n-1}$ then depends on α. For $\alpha = 1/2$ we have a centered scheme in time, while $\alpha = 1$ and $\alpha = 0$ are fully explicit and fully implicit schemes, respectively. Note that at any time the pressure can be determined from the velocity, and as such it is not directly a function of time. However, since it appears only on the right-hand side it is common to compute the pressure at discrete points located at the midpoint between the time steps where velocity is computed $t_{n-1/2}, t_{n-3/2}, \ldots$.

The convecting and convected velocity fields $\tilde{u}$ and $\bar{u}$ in the B formula can be approximated in various ways. The most obvious choice is $\tilde{u} = \bar{u} = u_{n-\alpha}$ to be consistent with the other terms. However, alternative choices may simplify the solution process. For example, $\tilde{u} = \bar{u} = u_n$ yields in combination with $\alpha = 0$ a consistent nonlinear backward Euler scheme. An explicit treatment of the convection term is obtained by $\tilde{u} = \bar{u} = u_{n-1}$. A linear implicit scheme requires that u_n is present (linearly) in either $\tilde{u}$ or $\bar{u}$, but not in both.

In this work we will make use of one explicit and two implicit discretizations of the convection term B:

$$B(\tilde{u}, \bar{u}) = \frac{3}{2} B(u_{n-1}, u_{n-1}) - \frac{1}{2} B(u_{n-2}, u_{n-2}), \tag{22.21}$$

$$B(\tilde{u}, \bar{u}) = B(u_{n-1}, u_{n-\alpha}), \tag{22.22}$$

$$B(\tilde{u}, \bar{u}) = B(\frac{3}{2} u_{n-1} - \frac{1}{2} u_{n-2}, u_{n-\alpha}). \tag{22.23}$$

The explicit Adams-Bashforth scheme (22.21) is chosen primarily because of its popularity in the fluid mechanics community. The implicit schemes use a centered "Crank–Nicholson" time discretization with $\alpha = 1/2$ for the convected velocity, in combination with forward Euler (22.22) and Adams-Bashforth projection (22.23) for the convecting velocity. The scheme (22.22) is first-order, whereas the remaining two are second-order accurate in time (see, for instance, Figure 3 in Simo and Armero (1994)).

Equations (22.18) and (22.19) contain (in three-dimensional space) four unknown fields and four PDEs. Although the system of equations can be solved in the fully coupled way formulated in (22.18)

and (22.19), it is common to split the system into a set of simpler equations so that we can compute the velocity and pressure separately. This class of approaches is often referred to as fractional step methods.

The fundamental problem in (22.18) is that the pressure $p_{n-1/2}$ is unknown. One approximation is to use the known pressure gradient $\nabla p_{n-3/2}$ from the previous time step as a first guess. Then, (22.18) can be solved for u_n. Unfortunately, this u_n will most likely not also fulfill (22.19). Moreover, there is no obvious way to advance p to time $t_{n-1/2}$. Still, we may correct the solution of (22.18) by using an old pressure. Let us denote this tentative, or intermediate, solution by u_I (I for intermediate). Its equation is

$$\frac{u_I - u_{n-1}}{\Delta t} = -B_I(\tilde{u}, \bar{u}) + \nu\nabla^2((1-\alpha)u_I + \alpha u_{n-1}) - \nabla p_{n-3/2} + f_{n-\alpha}. \tag{22.24}$$

Note that in the expressions for $B(\tilde{u}, \bar{u})$ we replace u_n by u_I, which is why there is a subscript added to the B term.

We are now interested in correcting for the error $u_n - u_I$. Subtracting the exact equation (22.18) with $\alpha = 0$ from (22.24) yields an estimate of the error:

$$\frac{u_n - u_I}{\Delta t} = -\nabla\Phi + B - B_I + (1-\alpha)\nabla^2(u_n - u_I), \tag{22.25}$$

where $\Phi = p_{n-1/2} - p_{n-3/2}$ is a pressure correction. Note that for an explicit scheme with $\alpha = 1$, only the $-\nabla\Phi$ term remains on the right-hand side of (22.25) since in that case $B = B_I$. Even when $\alpha < 1$ it is common to drop the terms $B - B_I + (1-\alpha)\nabla^2(u_n - u_I)$. One therefore considers the simplified equation

$$\frac{u_n - u_I}{\Delta t} = -\nabla\Phi, \tag{22.26}$$

coupled with the requirement that the new velocity must fulfill:

$$\nabla \cdot u_n = 0. \tag{22.27}$$

We can easily eliminate u_n from (22.26) and (22.27) by solving for u_n in the former and inserting in the latter. This procedure results in a Poisson equation for Φ:

$$\nabla^2\Phi = -\frac{1}{\Delta t}\nabla \cdot u_I. \tag{22.28}$$

After solving this equation for Φ, we can finally update the velocity and pressure from (22.26) and the definition of Φ:

$$u_n = u_I - \Delta t\nabla\Phi, \tag{22.29}$$

$$p_{n-1/2} = p_{n-3/2} + \Phi. \tag{22.30}$$

To summarize, the fractional step algorithm involves solving (22.24), (22.28), (22.29), and (22.30). The latter two are trivial, the Poisson equation (22.28) is straightforward, and (22.24) is easy to step forward if $\alpha = 1$, otherwise we need to solve a potentially nonlinear convection-diffusion vector equation. All of these equations are simpler than the original coupled problem in (22.18)–(22.19).

A particular advantage of the fractional step method is that it opens up the possibility of decoupling the vector equations (22.24) and (22.29). The latter can be updated pointwise, one velocity component at a time. In a finite element context, however, values of $\nabla\Phi$ at points where velocity degrees of freedom are defined can be cumbersome to compute since $\nabla\Phi$ is a discontinuous field. Solving

(22.29) by projection is then a viable alternative. Also in this case, we can take advantage of the fact that (22.29) are three decoupled scalar equations, and solve each scalar equation separately. The resulting linear system, involving a "mass matrix", then has the size corresponding to a scalar partial differential equation and not the triple size corresponding to the vector formulation in (22.29).

With $\alpha = 1$ in (22.24) the three component equations decouple so that we can solve one of them at a time. In that case we get a linear system with a "mass matrix" as coefficient matrix, exactly as when decoupling (22.29). For $\alpha < 1$ the convective term may lead to coupling of the component equations. Treating the convective term explicitly, but allowing implicitness in the viscosity term implies decoupled component equations and a possibility to solve a scalar heat or diffusion equation, with source terms, for each component separately. The size of coefficient matrices in the decoupled cases is one third of the size for a coupled vector equation, leading to much less storage and more efficient solutions.

A disadvantage of the fractional step method is that even though the resulting velocity field should be divergence-free due to the pressure correction (22.28), the corrected velocity field will no longer satisfy the discretized momentum equation. This "splitting error" associated with the fractional step method is known to be first or second order in time depending on whether the pressure is explicitly included or not included at all in the first velocity step (Guermond et al., 2006). To eliminate the splitting error it is possible to iterate over the three steps, a practice that is rarely followed for incompressible flows. A formally superior approach, which will be explored in this work, is to solve for the velocity and pressure simultaneously; that is, in a fully coupled manner. Such a coupled solver comes at a larger memory cost, which makes it less suitable for large-scale turbulence applications. However, there is no splitting error and thus the method can in general take longer time steps and be particularly useful for validating fractional step solvers.

The convective term contains two velocity fields $\tilde{u}$ and $\bar{u}$ that are equivalent in the exact NS-formulation, but that may differ when discretized. As such, the convective term $B(\tilde{u}, \bar{u})$ discussed above can alternatively be implemented by switching convecting and convected velocities to $B(\bar{u}, \tilde{u})$, which in discretized form will differ from $B(\tilde{u}, \bar{u})$. Nevertheless, recall that it is only the first velocity field in B that needs to be divergence-free for the convection to be energy conservative. Hence, since the velocity fields of previous time-steps are (nearly) divergence-free, it is preferable for fractional step methods to employ an explicit discretization (as in (22.21)–(22.23)) of the first, convecting velocity. Furthermore, making the convecting velocity implicit introduces additional coupling between the velocity components, which makes it less suitable for exploiting enhanced computational efficiency through solving the component equations one by one.

Implementation in FEniCS The solvers and problems under investigation are implemented in much the same way as described in Chapter 21, but the naming convention for the variables is somewhat different. Here we introduce `u` and `p` for the unknown velocity u_n and pressure p in the variational formulation of the governing equations. The compound (u, p) field is named `up` and defined on the composite space of the velocity and pressure spaces. Such a compound field is needed in the fully coupled formulation. Both `u` and `p` are `TrialFunction` objects, while `up` is of type `TrialFunctions`. An appended underscore indicates the most recently computed approximation to `u`, `p`, and `up`: `u_`, `p_`, and `up_`, all of which are `Function` objects. The velocities at previous time steps, u_{n-1}, u_{n-2}, ..., are denoted by `u_1`, `u_2`, ... These are `Function` objects. Similarly, `p_1` represents the old (corresponding to $p_{n-3/2}$) pressure (`Function`). The quantities $\tilde{u}$ and $\bar{u}$ are named `u_tilde` and `u_bar`, respectively, in the code. Below we only show some key snippets from the FEniCS implementation.

Given a `Mesh` object `mesh`, a string `mode` describing the type of formulation of the convective term, and `Constant` objects `dt` and `nu` for the time step and viscosity, the key steps in formulating the variational problem for the coupled problem are as follows:

Python code

```
from dolfin import nabla_grad as grad         # Used throughout chapter
from dolfin import nabla_div as div
V = VectorFunctionSpace(mesh, "Lagrange", 2)  # velocity space
Q = FunctionSpace(mesh, "Lagrange", 1)        # pressure space
VQ = V * Q  # composite space (Taylor-Hood element)
u, p = TrialFunctions(VQ)
v, q = TestFunctions (VQ)
up_  = Function(VQ)
up_1 = Function(VQ)
up_2 = Function(VQ)
u_, p_ = up_.split()
u_1, p_1 = up_1.split()
u_2, p_2 = up_2.split()
u_tilde = 1.5*u_1 - 0.5*u_2
u_bar = 0.5*(u + u_1)

F  = inner(u - u_1, v)*dx + dt*nu*inner(grad(u_bar), grad(v))*dx + \
     dt*conv(u_tilde, u_bar, v, mode)*dx - dt*inner(f, v)*dx - \
     dt*inner(p, div(v))*dx + inner(div(u), q)*dx
a = lhs(F); L = rhs(F)

x_ = up_.vector()  # unknown solution vector (u,p)
dx = Function(VQ)  # correction vector in Newton system
```

Note that the unknown vector `x_` in the nonlinear algebraic equations is just the vector of degrees of freedom in the `up_` finite element function so that `up_` and `x_` shares memory. Moreover, `u_` and `p_` are parts (views) of `up_` and share memory with the latter and `x_`. That is, we can choose between a linear algebra view `x_` (vector of degrees of freedom) or a finite element function view `up_`, or the velocity part `u_` or pressure part `p_` of `up_`. In memory there is no duplication of velocity and pressure data.

The three alternative versions of the convective term discussed in Section 22.2.1 have be implemented in the method `conv` as

Python code

```
def conv(u_tilde, u_bar, v, mode="standard"):
    if (mode == "Standard"):
        return inner(u_tilde*grad(u_bar), v)
    elif (mode == "Divergence"):
        return inner(div(outer(u_tilde, u_bar)), v)
    elif (mode == "Skew"):
        return 0.5*(inner(u_tilde*grad(u_bar), v) + \
        inner(div(outer(u_tilde, u_bar)), v))
```

The fractional step Navier–Stokes solver is somewhat more elaborate than the fully coupled, since there are more steps involved. The details of several fractional step solvers are given in Chapter 21, and thus not repeated here.

22.2.5 Speed-up of "naive" Navier–Stokes solvers

The previous section presents the straightforward (or naive) implementation of a coupled vector and scalar equation in FEniCS, using mixed finite elements. Examining the structure of the NS equations, one realizes that many of the terms give rise to similar block matrices in the coefficient matrix for the complete linear system. The linear system has size $(n_v d + n_p) \times (n_v d + n_p)$ if d is the number of space dimensions, n_v is the number of degrees of freedom for a velocity component field, and n_p is the number of degrees of freedom for the pressure field. Several blocks of size $n_v \times n_v$ are identical since there are three scalar time-derivative terms, giving rise to three identical mass matrix blocks, and three

scalar viscosity (or similarly convection) terms, giving rise to three Laplacian "stiffness matrix" blocks. Moreover, these blocks are constant in time and do not have to be reassembled at each time step.

We could gain a potentially significant speed-up by exploiting the mentioned properties and thereby avoid computing and assembling large parts of the total coefficient matrix. This is perhaps not important for smaller two-dimensional problems, but for larger three-dimensional problems the naive implementation is much slower than an algorithm exploiting the special structure of the NS equations. For CFD practitioners using FEniCS this speed–up is significant and makes the efficiency of a fairly simple FEniCS-based NS solver compete with expensive, and much less flexible, state-of-the-art CFD software. We shall therefore go through the relevant optimization steps here in detail.

1. Split `F` into accumulation $(\partial u/\partial t)$, convection $(u \cdot \nabla u)$, and diffusion terms $(\nu \nabla^2 u)$, and take advantage of the fact that for the total coefficient matrix it is only the nonlinear convection term that needs to be reassembled at each time step. The matrices for the linear, constant-in-time accumulation and diffusion terms can be assembled before going into the time loop.

2. For known convecting velocity, the velocity components in the momentum equation (22.1) are decoupled and can as such be solved for in a memory efficient segregated manner, treating one component equation at a time with the same (small) coefficient matrix. An additional requirement is that some old value of p is used, which makes this optimization relevant only for velocity steps in a fractional step method.

3. In a fully coupled formulation, we can assemble small $n_v \times n_v$ matrices for a term in a component equation and insert it into the relevant places in the total coefficient matrix. For example, to assemble a convection matrix for one component, define a `FunctionSpace Vc` for one scalar velocity component identical to the vector space `V.sub(0)`:

   Python code

   ```
   uc = TrialFunction(Vc)
   vc = TestFunction(Vc)
   Ac = assemble(conv(u_1, uc, vc)*dx)
   ```

   Here `u_1` is the `Function` on the space `V` holding the approximation of the convecting velocity (taken as u_{n-1} in this example). The matrix `Ac` is of size $n_v \times n_v$ (n_v equals `Vc.dim()`). The large matrix for the complete velocity vector field can now be obtained simply by copying this `Ac` matrix to the three diagonal slots in the 3×3 block matrix that makes up the whole convection matrix for the velocity vector field. At the time of writing, it is for a three-dimensional problem approximately 20 times faster to assemble this `Ac` than assembling the $n_v d \times n_v d$ matrix for the complete convection term $u_{n-1} \cdot \nabla u$ by `assemble(conv(u_1, u, v)*dx)`.

4. The right-hand side of the linear system can be reassembled at each time step using matrix–vector products and vector additions only, a procedure that is described in Chapter 1.

5. The sparse coefficient matrix can be compressed by removing redundant zeroes. For three-dimensional problems the assembled diffusion matrix (using a `VectorFunctionSpace`) contains approximately three times as many zeroes as nonzeros, since the sparsity pattern of the matrix is determined by the connectivity of the degrees of freedom of the finite element fields. The assembled zeros slow down Krylov solvers, which rely on fast matrix–vector products, as well as preconditioners.

In addition to these steps there are also some simple switches that can be turned on in the form compiler that optimize the assembly process.

Convection	Fully coupled		Segregated	
	Optimized	Naive	Optimized	Naive
Explicit 22.21	46/18 (4.5/3.7)	170/31 (11.3/122)	45/11 (3.0/2.8)	134/31 (8.3/81)
Implicit 22.23	45/17 (4.4/3.5)	688/662 (10.5/628)	44/11 (3.1/2.5)	498/462 (8.5/434)

Table 22.1: CPU times (total/inside time loop) for the fully coupled and segregated solvers using two different convection schemes in the Taylor Green problem. For each solver the first two numbers represent total time and time spent in the solver loop. Numbers in parenthesis show timings for the solver and assembler respectively.

Table 22.1 summarizes the effect of the speed-up routines on the CPU-time for the Taylor–Green problem (see Section 22.3.3) with $Re = 100$ on a `UnitCube` of size 16^3 using four time steps and a total integration time of 0.5. The naive implementations referenced in Table 22.1 use one single form for the Navier–Stokes equations and lhs/rhs to extract forms for the numerical schemes. This corresponds roughly to the code presented at the beginning of Section 22.2.4 for the coupled solver and similar for the segregated solver. The optimized versions have been implemented following the steps outlined above. The two CPU times shown are the total time and the time spent inside the time integration loop; that is, the total time minus the time it takes to set up the problem for looping. The two numbers in parenthesis show the time spent in the Krylov solvers and the assemblers respectively. Evidently, with the implicit solvers we can for this problem obtain a speed-up factor of nearly 40 (662/17) for the coupled solver. Most of this speed–up follows from minimizing the amount of code that needs to be reassembled every time step and by avoiding a direct assembly of a large $(n_v d + n_p) \times (n_v d + n_p)$ matrix. As can be seen, the Krylov solvers are for the optimized solvers approximately a factor three faster, which is attributed to the compression of the sparse matrices, which speeds up the matrix–vector products in the Krylov solvers. It has been verified that the naive and optimized solvers produce exactly the same results.

22.3 Numerical investigations

In this section, we will look at three popular test cases for validation of the numerical methods outlined in Section 22.2. The simplest and most straightforward test case is the Burgers' equation, which is widely used in numerical benchmarks because of its simplicity and resemblance to the Navier–Stokes equations. The inviscid form of Burgers' equation will be used to illustrate differences between convective terms described in Section 22.2.1. The second, more elaborate, test case is the Orr–Sommerfeld eigenvalue problem, which will be used to evaluate the performance of NS solvers discussed in Section 22.2.4 for long integration times. The final test case is the Taylor–Green vortex, which is a three-dimensional and transient instability problem where an analytical, yet unstable, initial condition is evolved in a triply periodic domain with no obstructions.

22.3.1 Burgers' equation

Burgers' equation reads:

$$\frac{\partial u}{\partial t} + u\nabla u = \nu\nabla^2 u, \quad x \in (-1,1), \quad 0 < t, \quad u(\pm 1, t) = 0, \tag{22.31}$$

$$u(x,0) = -\sin(\pi x) + \kappa\xi, \tag{22.32}$$

where ξ is a random number between zero and one and is used to create a discrete "white noise" (uncorrelated) fluctuating velocity field resembling turbulence, and κ is the amplitude of the perturbation.

The variational form of Burgers' equation is obtained by multiplying (22.32) with test function v and integrating over the domain, neglecting boundary terms due to the homogeneous Dirichlet boundary conditions. The resulting variational form involves finding u_n such that

$$\frac{1}{\Delta t}\langle u_n - u_{n-1}, v\rangle = -\langle B(\tilde{u},\bar{u}), v\rangle - \nu\langle \nabla u_{n-\alpha}, \nabla v\rangle. \tag{22.33}$$

We have used Crank–Nicholson-style time discretization ($\alpha = 0.5$) in all our investigations of this case. In one space dimension the convection terms need some modification from Section 22.2.4 due to the fact that in one dimension the velocity is a scalar and the correlation between standard and divergence forms reads $\nabla u\, u = 0.5\nabla u^2$. To arrive at a skew-symmetric form the following combination of standard and divergence forms is used

$$B(\tilde{u},\bar{u}) = \frac{1}{3}\left(\tilde{u}\nabla\bar{u} + \nabla(\tilde{u}\bar{u})\right). \tag{22.34}$$

Initialization of the FEniCS `Function u0` can be performed by subclassing class `Expression` as

Python code

```
from numpy import sin
from numpy.random import randn
class U0(Expression):
    def eval(self, values, x):
        if x[0] < -1. + DOLFIN_EPS or x[0] > 1. - DOLFIN_EPS:
            # no noise/perturbation at the boundary:
            values[0] = -sin(pi*x[0])
        else:
            values[0] = -sin(pi*x[0]) + self.kappa*randn()
u0 = U0(element=V.ufl_element()); u0.kappa = 0.2
u0 = interpolate(u0, V)
```

The variational problem can be implemented and solved as

Python code

```
bc = DirichletBC(V, Constant(0), DomainBoundary())

T = 0.25; Nt = 200; k = Constant(T/Nt); t = 0
alfa = Constant(0.5); nu = Constant(0)
u_tilde = u_1 # or u_tilde = 1.5*u_1 - 0.5*u_2 for Adams-Bashforth
u_bar = alfa*(u + u_1)
mode = "standard"  # "skew" or "divergence" (convection term)

F = v*(u - u_1)*dx + k*conv(u_tilde, u_bar, v, mode)*dx + \
    k*nu*u_bar.dx(0)*v.dx(0)*dx
a = lhs(F); L = rhs(F)
u_1 = interpolate(u_, V); u_2 = interpolate(u_, V)
while t < T:
    t = t + dt
    A = assemble(a)
    b = assemble(L)
    bc.apply(A, b)
    solve(A, u_.vector(), b, "gmres", "ilu")
    u_2.assign(u_1); u_1.assign(u_)
```

Note that the coefficient matrix `A` needs to be reassembled due to the implicit treatment of

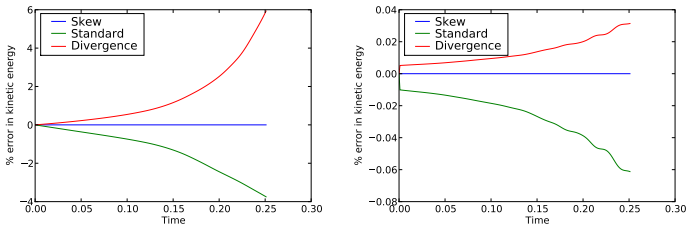


Figure 22.1: Accumulation of error in kinetic energy for the inviscid Burgers' equation initialized as $u(x,0) = -\sin(\pi x) + 0.1\xi(x)$. Left and right figures represent the results of using (22.22) and (22.23) for convection respectively.

convection. Figure 22.1 shows how the standard, divergence and skew forms of the convective term perform for the two implicit solvers. As expected the errors of using the second-order accurate Adams-Bashforth projection (right) are much smaller than for the forward Euler method (left). Most noteworthy, though, is the exact conservation of kinetic energy achieved by the skew form. As mentioned before, this feature follows simply from the fact that the assembled matrix `A` is perfectly skew-symmetric – a feature that also is retained by the skew form in two- and three-dimensional cases. Results of using the explicit convection are not shown, since for the current problem the three convection forms differ only when treated implicitly.

22.3.2 Orr–Sommerfeld equation

The Orr–Sommerfeld equation (see Orzag (1971)) is derived by assuming a wave-like disturbance (perturbation) that is proportional to $\exp(i(\alpha x - \lambda t))$, where λ is an eigenvalue (the complex frequency), α is a prescribed wavenumber (we use $\alpha = 1$), x is the streamwise direction and t is time. The perturbation is applied to the stream function $\psi(x,y,t)$ such that $\psi = \phi(y)\exp(i(\alpha x - \lambda t))$, where ϕ is the eigenfunction of λ and the y-direction is normal to x (we consider only two dimensions). Consequently the velocity perturbations are

$$u'(x,y,t) = \frac{\partial \psi}{\partial y} = i\alpha\phi \exp(i(\alpha x - \lambda t)), \tag{22.35}$$

$$v'(x,y,t) = -\frac{\partial \psi}{\partial x} = -\phi' \exp(i(\alpha x - \lambda t)). \tag{22.36}$$

If we insert this perturbation into the Navier–Stokes equation, an eigenvalue problem, the Orr–Sommerfeld equation, will appear. The equation reads

$$\left(\frac{d^2}{dy^2} - \alpha^2\right)^2 \psi - (\bar{U} - \lambda)\frac{i\alpha}{\nu}\left(\frac{d^2}{dy^2} - \alpha^2\right)\psi - \bar{U}''\psi = 0, \tag{22.37}$$

where ν is the kinematic viscosity and $\bar{U}(y)$ is the unperturbed, or basic, velocity.

The Orr–Sommerfeld equation can be solved numerically for any type of basic flow, but is particularly simple for a channel or Couette flow where $\bar{U}$ is known analytically. If the channel spans $-1 \leqslant y \leqslant 1$, then the perturbed velocity in a parallel channel flow equals

$$\begin{aligned} u(x,y,t) &= 1 - y^2 + \epsilon\,\mathrm{Re}\left(i\alpha\phi\exp(i(\alpha x - \lambda t))\right), \\ v(x,y,t) &= -\epsilon\,\mathrm{Re}\left(\phi'\exp(i(\alpha x - \lambda t))\right), \end{aligned} \tag{22.38}$$

where ϵ is the perturbation amplitude, which needs to be much smaller than unity.

The Orr–Sommerfeld disturbance evolves very slowly and for $Re = 8000$ it takes approximately $2\pi/\mathrm{Re}(\lambda) \approx 25$ time units to travel through the domain. In other words, the NS equations typically need to be integrated for very long times and the stability of the numerical time integration scheme

thus becomes an important factor. Furthermore, the Reynolds number may be varied over decades (both the viscous and the inviscid limits), and a wide range of different solutions may be explored, as any mode (not just the unstable one) yields a different analytical solution. Altogether, this makes the Orr–Sommerfeld equation an ideal test case for NS solvers.

Solution of the Orr–Sommerfeld equation The Orr–Sommerfeld eigenvalue problem must be solved with high numerical accuracy. Here the equations are solved using spectral collocation with Chebyshev polynomials, as described by Trefethen (2006). We consider a channel with Reynolds number $Re = 1/\nu = 8000$, where the mean pressure gradient is a constant equal to $2/Re$. Using 80 Chebyshev points the eigenvalues for this problem are plotted in Figure 22.2. Note the eigenvalue shown as an open square ($\lambda = 0.24707506 + 0.00266441i$), which is the only eigenvalue with a positive imaginary part. Since the imaginary part is positive, it is evident that this represents an unstable mode that will grow in time. Hence one might argue that eventually this disturbance will become unstable and lead to transition from laminar to turbulent flow.

The Orr–Sommerfeld equation is derived directly from the NS equations by assuming that the perturbation is small compared to the mean flow. Hence if the mean flow in a channel is initialized like (22.38), the instability should grow "exactly" like implied by the Orr–Sommerfeld equation (22.37). This has been used to validate NS solvers by Malik et al. (1984). The perturbation flow energy is here used as a measure for the accuracy of the solver:

$$E(t) = \int_0^{2\pi} \int_{-1}^{1} \left(\left[u - (1 - y^2) \right]^2 + v^2 \right) \, \mathrm{d}x. \tag{22.39}$$

The exact analytical perturbation energy at any time should be

$$\frac{E(t)}{E(0)} = \exp(i\mathrm{Imag}(\lambda)t). \tag{22.40}$$

Note, however, that we are looking at the energy of the *disturbance* only. In other words, we are looking at an energy transfer drained from the mean field ($\bar{U} = 1 - y^2$) into the perturbation. This is a very different situation from looking at the total energy of the field, which should be conserved. The energy of the perturbation increases with time and as such it is no longer evident that an energy conserving scheme, like the skew form, has any significant advantage over the not necessarily conservative standard convection form.

Initialization in FEniCS The implementation of the Orr–Sommerfeld test-case in FEniCS requires a two-dimensional computational mesh with associated parameters, like viscosity, etc. The mesh and some necessary parameters are declared as:

Python code

```
from dolfin import *
from numpy import arctan
mesh = Rectangle(0., -1., 2*pi, 1., 40, 40)
x = mesh.coordinates()
x[:, 1] = arctan(2.*(x[:, 1]))/arctan(2.)  # stretch mesh toward wall
Re = 8000.; nu = Constant(1./Re)
f = Constant((2./Re, 0.)) # Pressure gradient
```

where the constant pressure gradient is implemented as a body force to enable the use of periodic boundary conditions for both velocity and pressure.

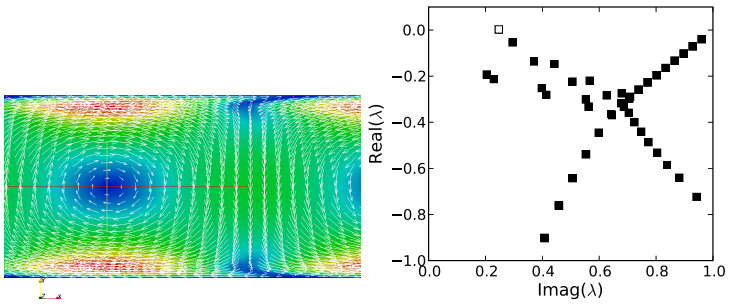


Figure 22.2: The left figure shows a snapshot of the initial perturbed velocity field. The figure on the right shows the eigenvalues for the Orr–Sommerfeld equation at $Re = 8000$. Note the open square, which is the only eigenvalue with a positive imaginary part. This represents an unstable mode.

At our disposal we have an Orr–Sommerfeld eigenvalue solver that uses spectral collocation in $n + 1$ Chebyshev points. The details of this solver is given by Trefethen (2006) and not repeated here, and the source code can be found in the file `OrrSommerfeld_eig.py` that resides within the CBC.PDESys package. For the initialization of DOLFIN `Functions` with the Orr–Sommerfeld solution, a subclass called `U0` of the DOLFIN class `Expression` is implemented such that it solves the eigenvalue problem on creation and overloads the `eval` function with the equivalence of (22.38). To initialize the specified initial velocity field we need to create an instance of the `U0` class and interpolate in the appropriate function space: `V` for fractional step and `VQ` for the coupled solver. The procedure for the fractional step solver is:

Python code

```
# Using 80 Chebyshev points and Reynolds number of 8000:
u = U0(Re=8000., N=80, element=V.ufl_element())
u_ = interpolate(u, V)
p_ = Constant(0.)
```

For the coupled solver, `VQ` is used in place of `V` and the pressure needs to be set in `U0`. The Orr–Sommerfeld perturbation leads to a non-trivial solution that evolves in time. The initial perturbed velocity field is illustrated on the left of Figure 22.2.

Results In this section we consider first the transient behavior of the Navier–Stokes solver using all three forms of convection discretization (standard, divergence and skew). The spatial discretization is kept well resolved with a Rectangle mesh class using $N = 48$ and the CFL number based on the mean velocity ($\bar{U} = 1$ ms^{-1}) is varied from 0.5 to 0.025. Figure 22.3 shows the accumulated error in the perturbation flow energy computed as

$$\text{Error} = \sum_{k=0}^{N} \frac{|E(t_k) - \exp(i\text{Imag}(\lambda)t_k)|}{N}. \tag{22.41}$$

Note that the integration time is kept quite low (from 0 to 0.5), in an effort to maintain stability for all schemes. However, using explicit convection the divergence form is still unstable for the highest CFL numbers. From Figure 22.3 we observe second-order accuracy in time and register that the accuracies of explicit and implicit methods for convection are similar. With implicit convection the superior accuracy of the coupled scheme versus the fractional step solver is evident, and the coupled scheme achieves the same accuracy with twice the CFL number, which is attributable to the splitting error. Using explicit convection, there is hardly any difference between fractional step and coupled solvers (the difference in the error is approximately 2% in the favor of the coupled solver throughout), indicating that the divergence of the intermediate velocity is small. Another interesting feature is that for explicit convection the standard form seems to be most accurate followed by the skew and divergence forms, whereas the opposite behavior is observed for the implicit solver.

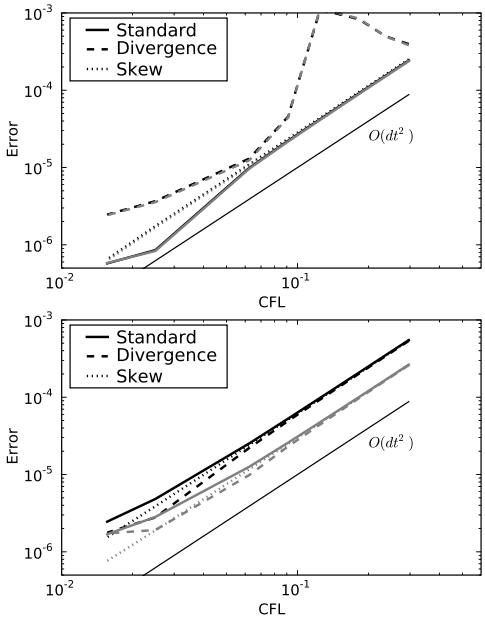


Figure 22.3: Accumulated error (22.41) vs CFL number for an integration time of 0.5 for standard, divergence and skew forms, here represented with solid, dashed and dotted lines respectively. The fully coupled and fractional step solvers are represented with gray and black lines, respectively. All results for a mesh size of $N = 48$. The figures on top and bottom use explicit (22.21) and implicit (22.23) convection respectively. Note that for the top figure the black and gray curves are practically identical (the error in the fully coupled solver is approximately 2% less throughout).

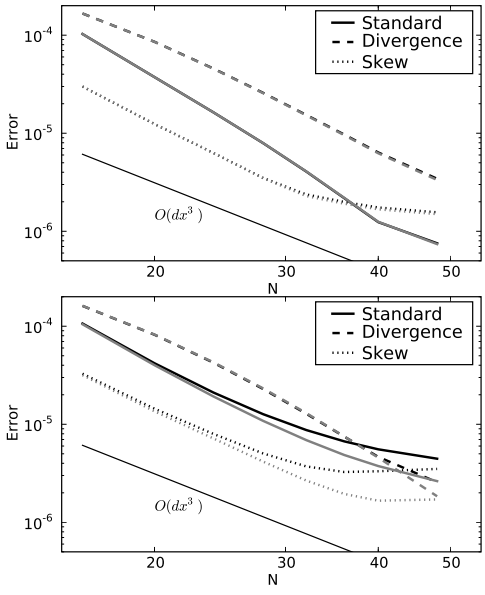


Figure 22.4: Accumulated error (22.41) vs mesh size for an integration time of 0.5. Standard, divergence and skew forms of convection are here represented with solid, dashed and dotted lines respectively. The fully coupled and fractional step solvers are represented with gray and black lines, respectively. For all results the time step used is 0.005. The figures on top and bottom use explicit (22.21) and implicit (22.23) convection respectively. Note that for the top figure the black and gray curves are practically identical (the error in the fully coupled solver is approximately 2% less throughout).

To investigate the spatial discretization with P2/P1 elements, we keep the time step constant and small at 0.005 and vary the mesh size from 16 to 48 in the `Rectangle` class. The accumulated error is shown in Figure 22.4, where we observe the third-order accuracy that is expected for Taylor-Hood elements. Again, the coupled solver performs better than the fractional step solver with implicit convection, whereas the solvers are practically identical with explicit convection. The larger splitting errors obtained with the fractional step solver using implicit treatment of convection (in both Figs. 22.3 and 22.4) can be understood by thinking of the fractional step solver as an operator splitting routine where the implicit diffusion and convection terms are neglected in the second pressure step. If the convection term is treated explicitly then the treatment is exact (since the old velocity is used in the term anyway). Hence, there is only an inconsistency for the diffusion that is being computed in the first step with an intermediate and not the end-of-step velocity field. With implicit convection as well, both diffusion and convection terms are computed with the intermediate (not divergence-free) velocity field and the inconsistency with the superior fully coupled scheme becomes more profound.

To validate the more interesting (from a turbulence instability point of view) long-term performance of the solvers, we integrate the equations as long as it takes for the perturbation to travel through the domain two times (end time ≈ 50). One single well-resolved mesh is used ($N = 40$ in the `Rectangle` class) and the CFL number is set to 0.05 or 0.1 to limit the temporal discretization errors. Figure 22.5 shows the evolution of the perturbation energy using both the fully coupled and fractional step solvers with the second-order implicit convection (22.23) and the second-order explicit scheme (22.21). Evidently, the standard form of convection is more stable than the divergence (most unstable) and skew forms for long integration times. The divergence and skew forms cannot capture the true evolution of the instability and the solution quickly blows up into a chaotic two-dimensional "turbulence" field. The standard form seems to capture the instability with ease and evolves more or less according to the true solution of the eigenvalue problem. There are only minor differences between the fractional step and the fully coupled solver, which is not unexpected since we are using a very short time step and the error in fractional step splitting (the only difference between the two methods) is thus minimized. By increasing the CFL number it can be shown that the fully coupled solver remains accurate for longer time steps. Note that the total kinetic energy remains more or less constant for all the simulations shown in Figure 22.5, even for the divergence and skew forms. Hence, the ability of the skew form to maintain total kinetic energy does not seem to be all that important when we are really interested in solving instability problems, where the most important physical process is that energy changes form (from the mean flow to the perturbation). Also plotted with circles in Figure 22.5 is the result of using the same number of degrees of freedom and time step with a standard cell-vertex based finite volume solver. The finite volume solver is discretized in a similar manner as our FEniCS solvers with implicit convection (22.23) using Adams-Bashforth projection and Crank–Nicholson diffusion. The integration method is fractional step, which is here slightly dissipative due to the collocated nature of the pressure and velocity. The implicit higher-order (P2/P1) FEniCS solvers are evidently much better at capturing this instability than the lower-order finite volume method, which is not surprising. The difficulties that low-order finite difference methods face when trying to capture the Orr–Sommerfeld instability have been reported by Malik et al. (1984) and Canuto et al. (2007).

22.3.3 Taylor–Green vortex

Finally, we consider a real transition to turbulence problem. The Taylor–Green vortex is characterized by an initialization based on an asymptotic expansion in time in a triply periodic domain spanning

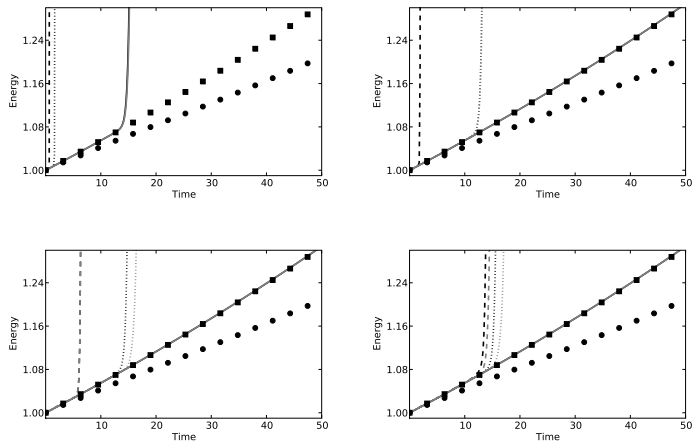


Figure 22.5: Temporal evolution of the perturbation energy. The gray and black lines correspond to the fully coupled and fractional step solvers respectively and the solid, dashed and dotted lines correspond to the standard, divergence and skew forms of the convection respectively. The symbolic dots represent the solution from a low-order finite volume solver and the squares represent the true solution. The left and right figures on the top row use explicit convection as in (22.21), with CFL of 0.1 and 0.05 respectively. The left and right figures on the lower row use implicit convection (22.23) and CFL = 0.1 and 0.05 respectively. Note that for explicit convection and the standard implicit form the gray and black curves are practically identical.

$[-\pi, \pi]$ in all three directions. The initial condition is:

$$u(x, y, t) = \sin(x)\cos(y)\cos(z), \tag{22.42}$$
$$v(x, y, t) = -\cos(x)\sin(y)\cos(z), \tag{22.43}$$
$$w(x, y, t) = 0. \tag{22.44}$$

The asymptotic expansion is known to diverge for $t \geq 3$, as the flow turns turbulent.

Due to the large memory requirements of this three-dimensional problem, we consider here only the fractional step solver. For validation we use the total kinetic energy and the total energy dissipation rate, computed respectively as

$$q = \frac{1}{2}\int_\Omega u \cdot u, \tag{22.45}$$
$$\varepsilon = \nu \int_\Omega \nabla u : \nabla u. \tag{22.46}$$

The average rate of dissipation is, as already mentioned, the single most important measure of a turbulent flow. It is implemented in FEniCS as

Python code

```
assemble(nu*inner(grad(u_), grad(u_))*dx)/(2*pi)**3.
```

Since the Taylor–Green vortex is (eventually) a turbulent flow there is no analytical solution that can be used to compare our results with. Hence, for validation the Taylor–Green vortex has also been simulated with Semtex (Blackburn, 2011), which is a well-tested open source spectral element Navier–Stokes solver that runs in parallel. Semtex uses quadrilateral spectral elements with standard nodal Gauss–Lobatto–Legendre basis functions and Fourier expansions in one homogeneous direction. To validate the Taylor–Green case we use Semtex with 30×30 homogeneous elements of order six in both x and y-directions and 144 planes in the z-direction that is solved using Fourier expansions. The

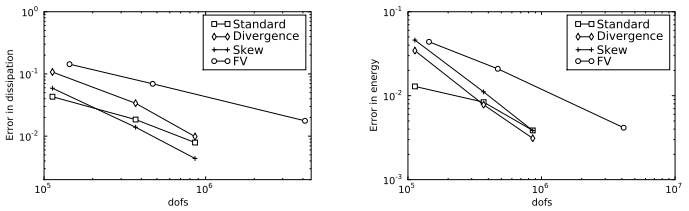


Figure 22.6: Relative errors in dissipation rate (22.46) is shown in the left figure and the energy (22.45) in the right. The results are displayed for implicit convection (22.23), and the squares, diamonds and pluses are used to represent standard, divergence and skew forms respectively. The open circles represent the solution obtained with a low-order finite volume code and the reference solution upon which the error is based is computed with Semtex (Blackburn, 2011).

time step used for the validation simulations is set to 0.005.

Figure 22.6 shows the error in average rate of dissipation and kinetic energy computed using $Re = 100$ and a `UnitCube` domain with 16^3, 24^3 and 32^3 bricks (each being divided into six tetrahedra). The CFL number used is 0.05, which practically eliminates temporal errors. With this small time step it is nearly impossible to distinguish between the results of using explicit (22.21) or implicit (22.23) convection and thus only the latter is shown. The conclusion that can be drawn from Figure 22.6 is that the standard convection form performs less satisfactory than both the skew and divergence forms. The skew form is best at capturing the average dissipation rate, whereas the divergence form does a slightly better job at capturing the total energy. Furthermore, the additional accuracy of higher-order elements is evidently superior to a low-order finite volume solver, both for the energy and the dissipation rate.

22.4 Conclusions

In this work we have validated FEniCS-based Navier–Stokes solvers aimed at applications involving turbulence and instabilities with transition to turbulence. Such solvers are of particular relevance to blood flow in the vicinity of aneurysms. Our focus has been on flow energy and energy conservation, features of great importance for turbulent flows. Discretizations of the nonlinear convection term have been considered both with standard, divergence and skew-symmetric forms - forms familiar from the vast literature on NS solvers. The numerical discretizations and solvers have been validated using Burgers' equation, the Orr–Sommerfeld perturbation to a plane channel flow in two dimensions and the three-dimensional unstable and transitional Taylor–Green vortex. We have briefly described the details of our NS solvers and outlined some optimizations that in our experience provide a speed-up of more than an order of magnitude compared to straightforward (naive) FEniCS implementation.

Two fundamentally different approaches to solving the NS equation have been tested: the fractional step method, which uncouples the velocity from the pressure, and a fully coupled solver. The fractional step method is generally favored by most CFD practitioners due to memory efficiency, even though it introduces a splitting error through uncoupling the velocity field from the pressure. The coupled solver naturally requires more memory, but on the other hand there is no splitting error as it simultaneously satisfies both the discretized momentum equation and divergence constraint. The splitting error introduced by the fractional step solver has been found here with the Orr–Sommerfeld test case to be small when convection is treated explicitly and enhanced when the convection term is treated semi-implicitly. With semi-implicit convection the fractional step method requires the CFL number to be half that of the coupled solver to achieve the same accuracy. The problem met by implicit discretizations of the fractional step method is here attributed to the fact that implicit terms are computed from the

(not necessarily divergence-free) intermediate velocity, as opposed to the divergence-free, end-of-step, velocity field. For the long integration times often associated with turbulence applications, though, we find that the implicit form remains stable and accurate where the explicit form cannot maintain sufficient stability.

For the Orr–Sommerfeld test case the standard form of convection seems to be the only form that remains stable for long integration times, even though the other forms are more accurate initially. For the Taylor–Green test-case the standard form is found to be less accurate than both the divergence and skew forms. Further studies with higher Reynolds numbers are required to more thoroughly validate the stability of NS solvers in the fully turbulent regime.

23 Computational hemodynamics

By Kristian Valen-Sendstad, Kent-Andre Mardal and Anders Logg

Computational fluid dynamics (CFD) is a tool with great potential in medicine. Using traditional engineering techniques, one may compute, e.g., the blood flow in arteries and the resulting stress on the vessel wall to understand, treat and prevent various cardiovascular diseases. This chapter is devoted to the computation of blood flow in large cerebral arteries and how the blood flow affects the development and rupture of aneurysms. We discuss the process, from generating geometries from medical imaging data to performing patient-specific simulations of hemodynamics in FEniCS. Specifically, we present three different applications: simulations related to a recently published study by Lindekleiv et al. (2010) concerning gender differences in cerebral arteries, a study of the carotid arteries of a canine with an induced aneurysm described in Jiang et al. (2010), and a study of the blood flow in a healthy Circle of Willis, where patient-specific velocity measurements are compared with a model for the peripheral resistance.

23.1 Medical background

Stroke is a leading cause of death in the developed part of the world (Feigin, 2005), and mortality rates could increase dramatically in the years to come (Murray and Lopez, 1997). Stroke is caused by an insufficient supply of blood to parts of the brain. There are mainly two different types of strokes: ischemia caused by obstructions in the blood vessels, and subarachnoid hemorrhage caused by the rupture of one or more aneurysms. Aneurysms typically develop in or near the so-called Circle of Willis, which is an arterial network of vessels at the base of the brain. The function of this circle is believed to be to ensure a robust and redundant system in the sense that the brain receives a sufficient amount of blood even if one of the vessels is occluded or under-developed. This network connects the internal carotid arteries (ICA) and the vertebral arteries (VA) in a circle-like structure, and it is the main supplier of blood to the brain. Figure 23.1 shows the circle as typically depicted in textbooks. Blood enters the circle through the ICAs, which are located at the front of the neck, and the VAs located in the back of the neck. The VAs join in the Basilar Artery (BA), and blood leaves the circle in the front through the Anterior Cerebral Arteries (ACA), in the back through the Posterior Cerebral Arteries (PCA), and at the sides through the Middle Cerebral Arteries (MCA). A patient-specific circle, the one used in Section 23.5, is shown in Figure 23.2.

Aneurysms are relatively common. As many as 1–6% of the population develop aneurysms during their lifetime (Weir, 2002). Unfortunately, aneurysms often rupture at a relatively early age. The average age of rupture is 52 years (Humphrey, 2001). An intracranial aneurysm is a dilatation of the blood vessel wall, and the reasons for initialization, growth and rupture of aneurysms are largely unknown. What is known is that increased wall shear stress (WSS) affects vascular endothelial cell turnover (Davies et al., 1986), that aneurysms may grow in the direction of low wall shear stress (Boussel et al., 2008), and that flow pattern and impingement zones affect the possibility of rupture

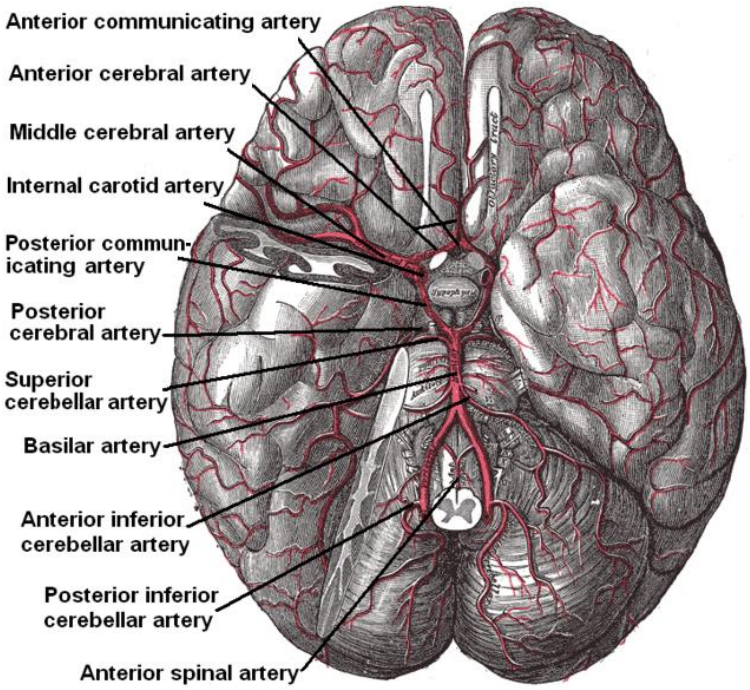


Figure 23.1: Illustration of the Circle of Willis and the base of the brain seen from beneath. The illustration is taken from Gray's Anatomy (Gray, 1897).

Figure 23.2: An image of a patient-specific Circle of Willis (of the second author) obtained with magnetic resonance angiography.

Figure 23.3: Computing normal and shear stresses from computed velocity and pressure fields u and p.

Python code

```
# Compute stress tensor
sigma = 2*nu*epsilon(u) - p*Identity(len(u))

# Compute surface traction
n = FacetNormal(mesh)
T = -sigma*n

# Compute normal and tangential components
Tn = inner(T, n) # scalar-valued
Tt = T - Tn*n    # vector-valued

# Piecewise constant test functions
scalar = FunctionSpace(mesh, "DG", 0)
vector = VectorFunctionSpace(mesh, "DG", 0)
v = TestFunction(scalar)
w = TestFunction(vector)

# Assemble piecewise constant functions for stress
normal_stress = Function(scalar)
shear_stress = Function(vector)
Ln = (1 / FacetArea(mesh))*v*Tn*ds
Lt = (1 / FacetArea(mesh))*inner(w, Tt)*ds
assemble(Ln, tensor=normal_stress.vector())
assemble(Lt, tensor=shear_stress.vector())
```

(Cebral et al., 2005). The vessel wall clearly responds to mechanical stimuli and this is the reason why wall shear stress is believed to be of special importance when trying to understand the pathogenesis of intracranial aneurysms. It is also known that the cerebral arteries lack perivascular support and the walls are relatively thin relative to the rest of the intracranial vasculature (Humphrey, 2001; Stehbens, 1975). Furthermore, the anatomy of cerebral vessels varies greatly. Only around 50% of the general population have a complete and well-balanced circle; the rest either have under-developed vessels or the vessels are missing completely (Fung, 1984). Gender, ethnicity and lifestyle have shown to be of importance (Mhurchu et al., 2001; Longstreth et al., 1994; Kongable et al., 1996).

23.2 Preliminaries

23.2.1 Stress calculation

We noted above that wall shear stress is of importance in computational hemodynamics. In Figure 23.3, we demonstrate how to compute stresses in FEniCS from a computed velocity field u and pressure field p. We start from the definition of the stress tensor $\sigma(u, p) = 2\nu\varepsilon(u) - pI$, where the $\varepsilon(u) = \frac{1}{2}(\nabla u + \nabla u^\top)$ is the symmetric velocity gradient. Then, the normal and tangential components of the stress are computed, where the tangential component is computed by subtracting the normal component from the traction $T = \sigma \cdot n$. Here, n is the inward-pointing unit normal from the vessel wall. In the code, n is the outward-pointing unit normal. To compute the shear and normal stresses as fields over the mesh, we test the stresses against piecewise constant test functions scaled by the inverse area of each facet. We thus obtain a piecewise constant representation of the stress which on each cell is equal to the average stress on that cell.

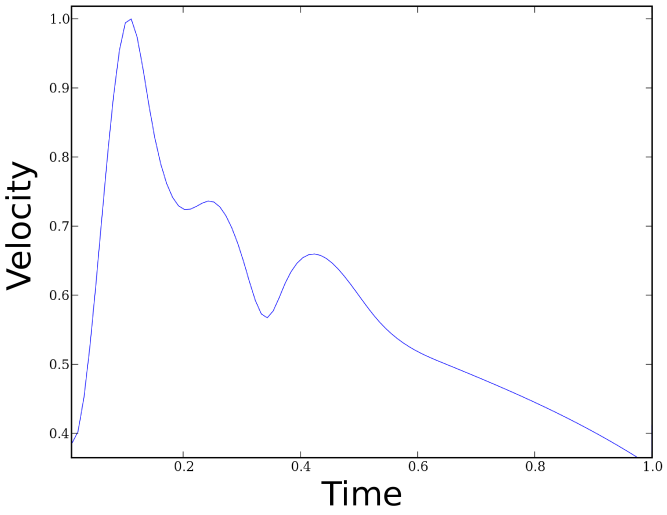


Figure 23.4: Nondimensionalized ICA inlet velocity profile.

23.2.2 *Boundary conditions*

For transient inlet boundary conditions, one option is to apply velocity waveform data in the ICA from Ford et al. (2005), where the average velocity was measured for seventeen young patients at rest. The nondimensionalized velocity is illustrated in Figure 23.4. The inlet velocity profile is easy to measure through the ICA or the VA using transcranial Doppler. This enables patient-specific velocity measurements such as in the Circle of Willis study in Section 23.5.

Further into the brain, the flow is divided into branches several times, which makes the outflow difficult to measure, both because of the thickness of the cranium and the decreasing size of the vessels. The effect of outflow boundary conditions in a complex network of blood vessels, such as in the Circle of Willis, is important to the flow division and wall shear stress.

The simplest way to describe the outflow is by applying a zero traction boundary condition at the outflow. However, the flow division in a bifurcation is dependent on the downstream vasculature, and the zero traction boundary condition does not capture this very well. Therefore, to model the peripheral resistance, a resistance model may be used for the pressure, while a Neumann condition ($\partial u/\partial n = 0$) is applied to the velocity. The value of the resistance boundary condition is proportional to the flow; that is, the pressure at the outlet Γ is modeled as,

$$p = p_0 + R = p_0 + C \int_\Gamma u \cdot n \, \mathrm{d}s, \tag{23.1}$$

where the resistance coefficient C was set according to Table 23.1, p_0 is the mean intracranial arterial pressure (85 mmHg), which is applied to the inlet, and u is the velocity. The coefficients in Table 23.1 are from Alastruey et al. (2007) and show a clear relation between the diameter of the vessel and the resistance coefficient. The implementation of the resistance boundary condition is shown in Figure 23.5.

The effect of the resistance boundary condition may bee seen in Figure 23.6 where the mass flux over two outlets in the canine geometry in Section 23.4 is calculated using both zero traction and a resistance boundary condition. The resulting flow division is clearly more evenly distributed (colored in red) between the daughter vessels, which intuitively also makes sense since the vessels reconnect further downstream. The method requires an iteration over a few cardiac cycles in order to converge.

Artery	C $[10^9 \text{Pa} \cdot \text{s} \cdot \text{m}^{-3}]$	Radius [mm]
Thoracic Aorta	0.18	9.99
External Carotid Artery	5.43	1.50
Middle Cerebral Artery	5.97	1.43
Anterior Communicating Artery	8.48	1.20
Posterior Communicating Artery	11.08	1.05

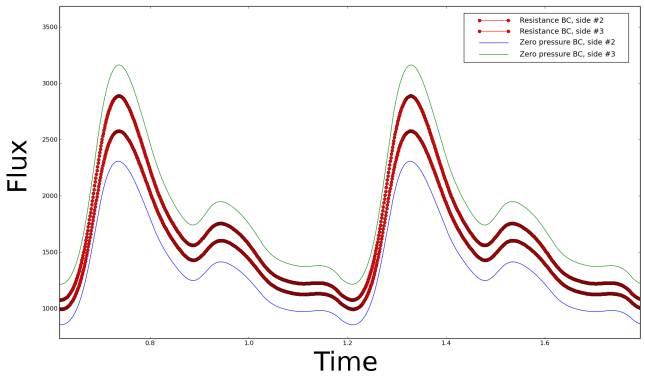
Table 23.1: Resistance boundary condition coefficient, C, for selected arteries of varying size; see Equation (23.1).

Figure 23.5: Calculation of the outflow boundary value for the pressure. The numbers have been multiplied with 10^{-3} and 10^{-6} to convert from SI units to millimeters, milliseconds and grams.

Python code

```
# Outflow boundary value for pressure
def OutflowBoundaryValue(self, i):
    u = self.problem.u
    n = FacetNormal(self.problem.mesh)
    flux = dot(u, n)*ds(i)
    Q = assemble(flux,
                 exterior_facet_domains=\
                 self.problem.sub_domains)
    C = 5.97*10**(-3)
    p0 = 11332.0*10**(-6) # 85 mmHg to Pascal
    R = (C*Q + p0)*(rhoinv)
    return R
```

Figure 23.6: Figure showing the difference of outflow flux when a resistance boundary condition (as described in section 23.2.2) is applied versus zero traction. When a resistance boundary condition is used, the flow is more evenly distributed between the two vessel outlets (red curves), compared to the case when a zero traction condition is used (blue and green curves).



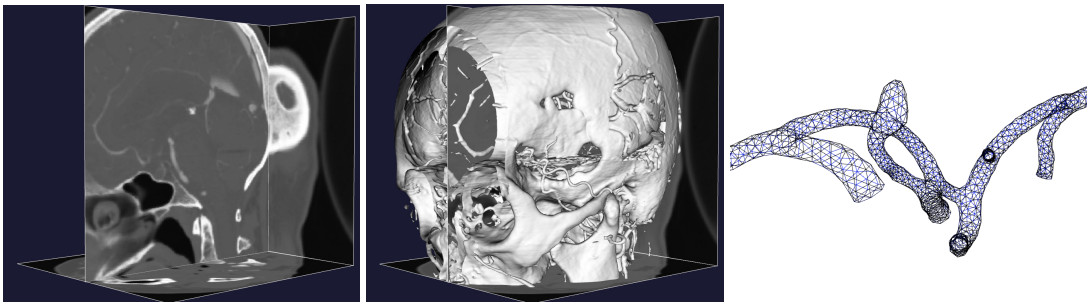


Figure 23.7: Image segmentation process; from MRI to mesh.

23.2.3 *Anatomical modeling*

Patient-specific geometries are obtained from Computed Tomography Angiography (CTA) or Magnetic Resonance Angiography (MRA) images as follows. A stack of 2D images is used as input to the Vascular Modeling Toolkit (VMTK), where a 3D surface model is generated based upon the light intensity in the pictures using level set techniques. A volume is then created from the surface, from which a mesh can be generated. This process is illustrated in Figure 23.7, where the mesh of a blood vessel extracted from the geometry is shown to the right. This mesh is used as input to the flow solver.

23.3 *Gender differences in the intracranial vasculature*

In this section, we present an overview of a recent study by Lindekleiv et al. (2010) where it is shown that on average, women have larger shear stresses than men in two intracranial bifurcations.

23.3.1 *Background*

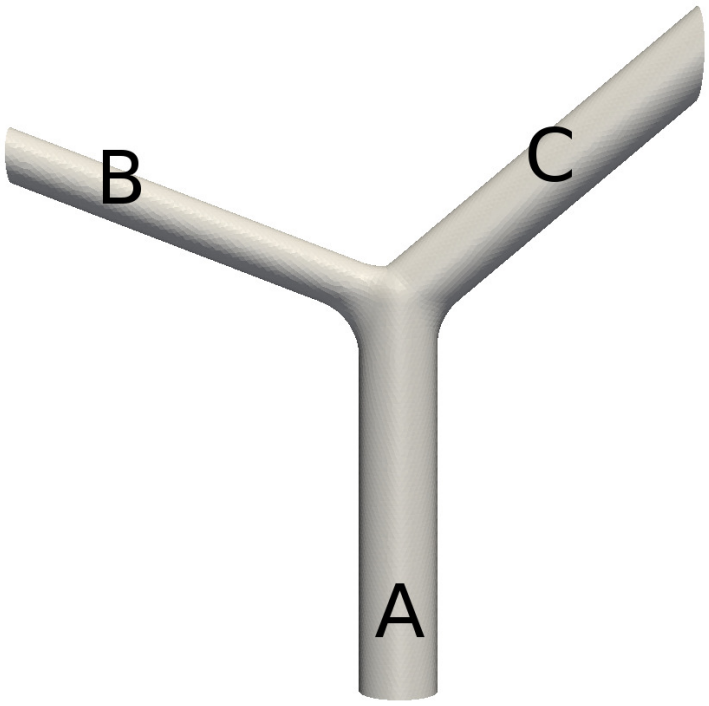
Females are more likely to harbor intracranial aneurysms than men and, consequently, more frequently develop subarachnoid hemorrhage (SAH) (Eden et al., 2008). The reason is not known, but studies suggest an increased risk of aneurysm rupture after the age of fifty, in the postmenopausal years. This might indicate the influence of hormonal factors on the vessel wall. This hypothesis is supported by the reduced risk of SAH with increasing number of births given by the female. However, studies have failed to prove a decisive correlation between hormonal factors and the risk of SAH. Another hypothesis is that high values of wall shear stress may influence the initialization of aneurysms. With measurements of radii and angles of intracranial bifurcations available from a previous study in our group, see Ingebrigtsen et al. (2004), we therefore wished to reanalyze the data and calculate the gender specific hemodynamic forces by numerical simulations.

23.3.2 *Method*

Measurements of 49 patients were performed to obtain the geometric quantities of the MCA and ICA bifurcations. The averaged values for the diameters were used to create one idealized bifurcation of the MCA and ICA for both females and males. The model basically consists of three cylinders connected with a smoothing at the interface to give a physiologically plausible appearance.

Average gender specific blood flow velocity measurements from the ICA and MCA from Krejza et al. (2005) were used as inflow boundary conditions in the simulations. Table 23.2 summarizes the input values to the simulations. At the outflows, we have applied a resistance boundary condition as described in Section 23.2.2.

Figure 23.8: Idealized model of a bifurcation.



23.3.3 Results

Table 23.2 shows that there is a significant gender difference in the diameters for the MCA. For the ICA, there are only statistically significant sex differences in the vessel size of the parent vessel and the smallest branch. CFD simulations show both increased wall shear stress and a larger affected area in the female MCA (Figure 23.9) and ICA (Figure 23.10) bifurcations. The maximum wall shear stress in the MCA bifurcations was 33.17 Pa for females and 27.82 Pa for males. Similar results for ICA were 15.20 Pa for females and 10.10 Pa for males. The values are reflected by a higher pressure drop in the female than in the male bifurcations (664 vs. 502 Pa for MCA and 344 vs. 202 Pa for ICA). For further discussion, see Lindekleiv et al. (2010).

23.3.4 Discussion

The above results are as expected from fluid mechanical reasoning, except for the peak values in the vicinity of the bifurcations. Even though the model is simple, the aim was to demonstrate a principle with a potentially important application; that is, that WSS may be of importance in the initialization and rupture of intracranial aneurysms. Furthermore, the results correlate well with the fact that women develop more aneurysms than men.

23.4 CFD versus 4D PC MRA in an experimental canine aneurysm

In another study, see Jiang et al. (2010), we quantitatively compared CFD, assuming Newtonian flow with rigid walls, with four-dimensional Phase Contrast Magnetic Resonance Angiography (PC MRA) techniques. The intention was both to verify the computational techniques for creating patient-specific models and corresponding CFD results and to understand and quantify the accuracy of the simplest possible flow model against state-of-the-art measurements.

	Male MCA	Female MCA	Male ICA	Female ICA
α	49.7°	50.5°	62.8°	57.2°
β	68.8°	72.5°	49.7°	50.5°
A [mm]	2.63	2.42	3.86	3.45
B [mm]	2.44	2.04	2.71	2.49
C [mm]	1.74	1.56	2.13	1.85
V [m/s]	0.68	0.74	0.34	0.42

Table 23.2: Summary of angles, diameters and velocities for the bifurcation of Figure 23.8, used in the simulations. The parameters α and β are the angles between the prolongation of the parent artery and the vessels C and B, respectively.

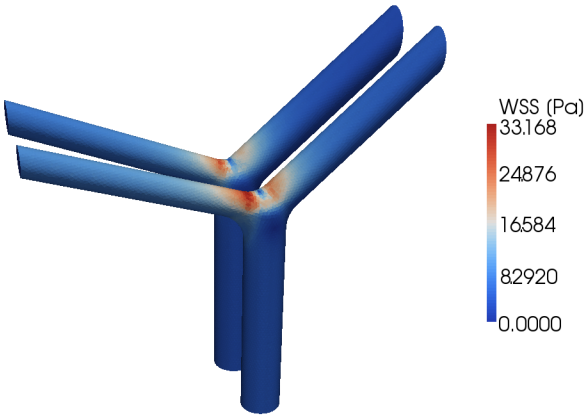


Figure 23.9: Resulting wall shear stress in the male and female MCA bifurcations. Female bifurcation in front.

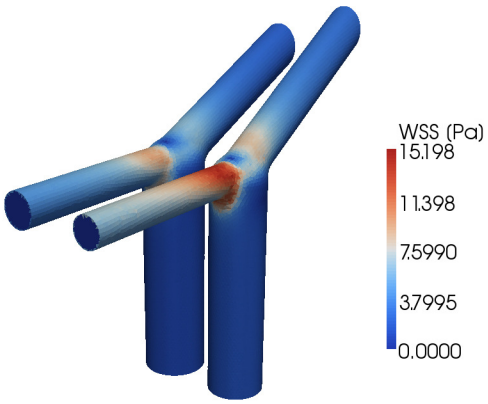


Figure 23.10: Resulting wall shear stress in the male and female ICA bifurcations. Female bifurcation in front.

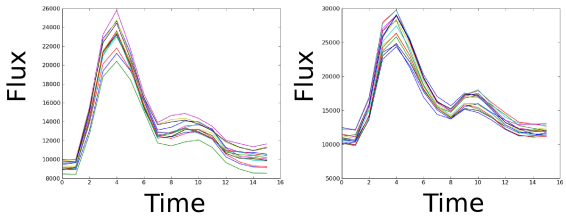


Figure 23.11: The figure shows the raw data obtained from the 4D PC MRA scan, which is phase averaged from 808 heart cycles. Each line corresponds to the sum of fluxes obtained from a cut plane in the vertical direction. Measurements of mass flow differ by up to ca. 20% between inlet (left) and outlet (right) in a canine. The x-axis shows the time step number.

Four-dimensional PC MRA is a noninvasive technique to measure flow in the vascular system. The image acquisition consists of a scan time of roughly 8 minutes. The canine had an average heart rate of 101 beats per minute, and the obtained images were averaged over 808 heart cycles. There is naturally no guarantee for a constant heart beat in the canine, which is a source for errors. Still, errors in the overall velocities have been shown to be of the order of 3-10% in large arteries such as the human pulmonary arteries; see Lotz et al. (2002); Evans et al. (1993), and the accuracy of the measurements are therefore judged to be acceptable for clinical use.

The resolution is coarse in both space and time, and the computation of forces such as WSS might thus be difficult. In addition to this, there might be locations in the vascular system where stenosis or plaque is present and the quality of the 4D PC MRA might be poor. These are also often the spots of most interest. In many cases, there are also problems with the Velocity Encoding Sensitivity (VENC) which may produce noise and useless data. The VENC may be adjusted to capture a velocity within a specific range. However, less accurate data is obtained for a wide VENC and vice versa.

23.4.1 Phase contrast magnetic resonance angiography

To test the above mentioned techniques in a complex case, our collaborators at the Wisconsin Institutes for Medical Research[1] created an artificial saccular aneurysm in a carotid bifurcation of a canine according to German and Black (1965). The inlet diameter was 3.2 mm, the height 9.4 mm, the width 4.3 mm, the volume 254.3 mm^3, the ostium area 17.10 mm^2, and the aspect ratio 2.18, where the aspect ratio is defined as the ratio between the aneurysm height and the neck width.

Three weeks after the artificial aneurysm was created, the canine was anesthetized and subjected to 4D PC MRA imaging studies; that is, the velocity measurements were performed. The raw data from the 4D PC MRA scan measurements are shown in Figure 23.11 where each solid line represents the sum of fluxes at different cross sections of the inlet and outlet arteries. The picture to the left in Figure 23.11 shows the sum of velocities at the inlet, which is one artery, while the picture to the right shows the output flux; that is, the sum of the outflow in both outflow arteries. For a more thorough description, we refer to Jiang et al. (2010). The coarse data obtained from 4D PC MRA is shown in the left and middle images of Figure 23.12, while the corresponding CFD simulation is shown to the right.

23.4.2 Computational fluid dynamics

The geometry was generated according to the procedure described in Section 23.2.3. We solved the incompressible Navier–Stokes equations using an Incremental Pressure Correction Scheme (IPCS) as described in Chapter 21. We used first order elements for both velocity and pressure, simulated over four heart beats, and obtained the results from the last cycle. With a CFL number of roughly ten, the number of time steps was 696 per cardiac cycle. As inflow boundary conditions, we used an average

[1] `http://www.med.wisc.edu/wimr/`

Figure 23.12: Figure showing resolution of data points where velocity measurements are made (left), velocity measurements from 4D PC MRA (middle), and CFD simulations (right).

Python code

```
def makeIC():
    Area = 8.04
    v = array([10939.45, 10714.00, 15406.95,
               25181.50, 27844.85, 24344.80,
               19578.05, 16479.55, 15168.80,
               16878.40, 16700.55, 15118.90,
               13032.50, 12121.65, 11885.90,
               11943.60, 10939.45 ]) \
                     / (Area*(133.0/256)**-2)

    t = 0.037*arange(len(v))
    t_period = 0.037*16
    return splrep(t, v), t_period
```

Figure 23.13: Measured values used for spline representation of the inflow.

value from the five lowermost voxels (3D pixels / samples from the measurements) in the z-direction. For the outflow, we applied a resistance boundary condition as described in section 23.2.2. The inflow was calculated according to Figure 23.13 and Figure 23.14.

Figure 23.13 shows how the values in Figure 23.11 are returned as a spline function. The factor `(133.0/256)**-2` scales the voxel size to the matrix size, so that the focus of the image corresponds to the actual size in millimeters. The `t` variable is the end time, and the scalar 0.037 is the equally spaced times of where measurements for `v` were made.

In Figure 23.14, a call is made to generate a spline representation of the velocity in time by calling `makeIC()`. Then, in `eval_data`, `n` is the outward facing facet normal and `t` is the time. The variable `val` is a spline evaluation such that the pulse goes in a continued cycle as time exceeds one heart beat. Finally, each component of the velocity vector, e.g., `values[0]`, is given the component-wise negative value of `n` (to create a flow going into the domain) times the velocity value corresponding to the current time.

23.4.3 Results

The resulting velocity field from 4D PC MRA and CFD calculations during peak systole are shown in Figure 23.12, and illustrates an overall good agreement between CFD and 4D PC MRA. For both canines (only one shown here), we obtained a similarity of more than 70% with respect to the velocity but only 22-31% similarity with respect to the WSS. For further details, we refer to Jiang et al. (2010).

23.4.4 Discussion

The reason for using the average values of the five lowermost cross sections as inflow is that given the resolution of the 4D PC MRA, each level of voxels is not necessarily mass conserving. As seen in

Figure 23.14: Calculation of inflow boundary value for the velocity.

Python code

```
class InflowBoundaryValue(Expression):

    def init(self, problem=None):
        self.problem = problem
        self.bc_func, self.t_period = makeIC()

    def eval_cell(self, values, x, ufc_cell):

        # Create DOLFIN Cell
        cell = Cell(mesh, ufc_cell.index)

        # Get normal for current facet
        assert(ufc_cell.local_facet >= 0)
        n = cell.normal(ufc_cell.local_facet)

        # Compute boundary value
        t = self.problem.t
        val = splev(t - int(t/self.t_period)*self.t_period,
                    self.bc_func)
        values[0] = -n.x()*val
        values[1] = -n.y()*val
        values[2] = -n.z()*val

    def value_shape(self):
        return (3,)
```

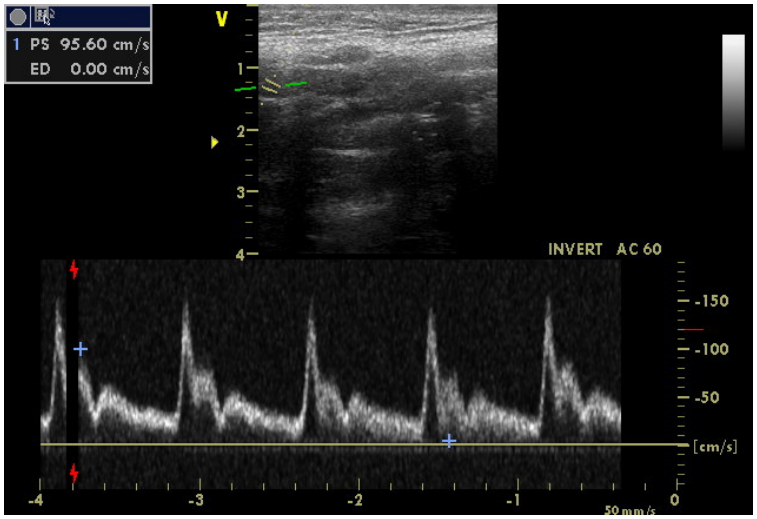
Figure 23.11, the sum of mass in a plane may vary by as much as 20% between sections (the solid lines). It is also clearly visible in this figure that peak systole appears at time step four in both left (inflow) and right (outflow) image of the figure, but the "bump" at mid deceleration has shifted from time step seven at the inflow to eight at the outflow. This may be because of the so-called Windkessel effect, which may only be captured using a fluid–structure interaction model, but it is difficult to conclude due to the coarseness of the measurements.

A limitation of the current study is that the results should not be interpreted as physiologically correct since the technique consists of cutting off one of the ICAs and creating an artificial bifurcation (and aneurysm) by moving the rest of the vessel over to the other ICA. This means that one of the ICAs supplies both left and right sides of the canine brain.

In the 4D PC MRA measurements at the left side of the parent artery in Figure 23.12, there are no boundary layers due to isotropic voxels, and the colors appear brighter since high velocities are possible close to the wall. The CFD simulations have short arrows at the same location indicating that the boundary layer has been resolved and we get lower velocity magnitudes. This is an obvious drawback with the 4D PC MRA. Thus, we get a good agreement with the velocity measurements, but poor agreement for computed wall shear stresses. The reason for this is believed to be the poor spatial resolution of the 4D PC MRA data. For a more thorough description, we refer to Jiang et al. (2010).

Each of the two methods has its strengths and weaknesses. While 4D PC MRA is fast, cheap and harmless, it uses average values over a voxel volume and fails to correctly compute WSS, recirculation zones, and possible turbulent structures. It also fails to provide values where the VENC is out of focus or in the presence of a stenosis. In contrast, CFD is expensive but may provide accurate computations of WSS over the entire domain.

Combined, the two methods may give a better understanding of the importance of boundary conditions, whether or not fluid–structure interaction is of importance, and possible pitfalls using the different methods. A first natural extension of this study may be to describe blood as a non-Newtonian fluid to determine whether or not non-Newtonian effects are of importance.



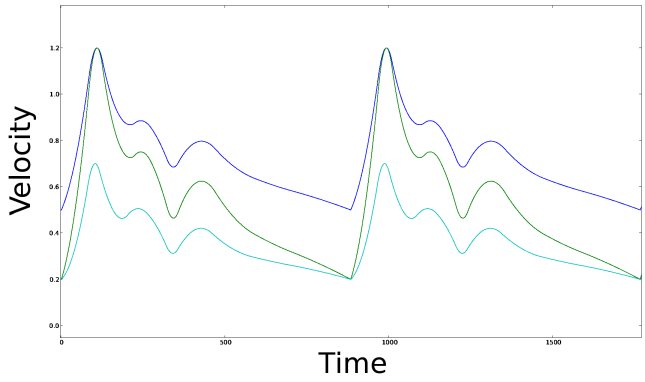


Figure 23.15: Inflow velocities used for the simulation of blood flow in a patient-specific Circle of Willis. Screenshot from TCD machine (top) with the artery on top and waveform below and implemented values (bottom). See text for details.

23.5 Patient-specific Circle of Willis

23.5.1 Background

In a study performed in collaboration with clinicians from the Neuroradiology department at Rikshospitalet University Hospital in Oslo, we wanted to investigate whether we are able to reproduce velocities in a full Circle of Willis with measurements at the inflow and compare with measurements at the outflow using resistance boundary conditions from the literature (Alastruey et al., 2007; Vignon-Clementel et al., 2006). Such an evaluation or verification of boundary conditions is essential before proceeding with more sophisticated models for the entire intracranial vasculature.

23.5.2 Method

Transcranial Doppler (TCD) was performed on a healthy volunteer at rest. During the velocity recording, the average pulse was about 73 beats per minute. The velocity measurements were used as boundary conditions for the vessels that are the main suppliers of blood to the brain; that is, the ICAs and VAs. Figure 23.15 shows the resulting waveform (right) that was applied from the measurements (left). The figure shows the ICA velocities from the top with equal value at peak systole (120 cm/s) and differing at end diastole (minimum 50cm/s in right ICA and 20 cm/s in left ICA). The lowermost line has a different waveform and shows values for two superimposed VAs since they are equal. The

Artery	Measured, L	Computed, L	Measured, R	Computed, R
MCA	70, 120	87	140, 150	55
ACA	200	100	90, 100	65
PCA	70, 80	62	80	100

Table 23.3: Measured versus computed values for flow velocities [cm/s] at left (L) and right (R) outflow arteries in a patient-specific Circle of Willis at peak systole. The cells with two values refer to different measurements made 5 weeks apart.

vasculature (based on an MRA scan) for this patient was already available from a previous study performed nine months earlier. The major vessels (ICA, MCA, PCA, ACA, VA, BA) were segmented as described in Section 23.2.3. The simulations were performed on meshes with three boundary layers where the number of tetrahedron cells were approximately 400,000, 900,000 and 2,600,000. We used continuous linear elements for both velocity and pressure, and an incremental pressure correction scheme with Adams–Bashforth implicit convection and Crank–Nicolson diffusion to solve the incompressible Navier–Stokes equations. The resolution in time was 3,532 time steps per heart beat on the coarsest mesh.

23.5.3 Results

Based upon images obtained from TCD, we compare only one point in time: peak systole. Since there is a large difference in the sum of inflow areas and outflow areas, we consider only the flow division between the arteries compared to measurements. Table 23.3 shows the measured and calculated velocities for the major arteries.

23.5.4 Discussion

The results of the current study do not match very well with measured values. This may indicate that the type of boundary conditions applied here may not describe the peripheral resistance properly. However, there are many sources of error that must be considered. First, TCD is difficult to perform and subject to errors. Personal communication with the neuroradiologist suggests errors at the scale of 20%. Second, we have no information on when peak systole appears in the different arteries. It seems reasonable that there is a small shift in time since the blood flows from the heart and through different arteries before it meets in the Circle of Willis. At present, we have not been able to quantify this shift. Third, the velocity itself may differ at different times for various reasons. This is illustrated by the cells containing two values in Table 23.3, which refer to two measurements of the same vessel in the same person only 5 weeks apart.

It is also a great challenge to segment the complete Circle of Willis due to great variations in diameters. This is clearly visible when performing an automatic segmentation where many of the smaller vessels disappear. It is known that the BA has approximately 50 tiny vessels that are clearly not present in Figure 23.17. The reason for this is that MRA measurements are based upon velocities, and hence the velocities in these vessels are too small to be captured. By calculating and summing up the in- and outflow areas using the code in Figure 23.16, we actually get an area difference of $37.18\,\mathrm{mm}^2 - 25.33\,\mathrm{mm}^2 = 11.85\,\mathrm{mm}^2$. It is not known what the correct area should be.

The simulations also show that it might be problematic to not include a large fraction of the parent artery when performing simulations on a smaller fraction of the vasculature. It is common to apply either a flat velocity profile or a Womersley profile upstream of the location of interest. This is clearly not the case as shown in Figure 23.18 where the flow is highly non-uniform.

Python code

```
def area(self, i):
    f = Constant(1)
    A = f*ds(i)
    a = assemble(A,
                 exterior_facet_domains=self.sub_domains,
                 mesh=self.mesh)
    return a
```

Figure 23.16: Calculation of the areas of the Circle of Willis geometry.

Figure 23.17: Patient-specific Circle of Willis (of the second author).

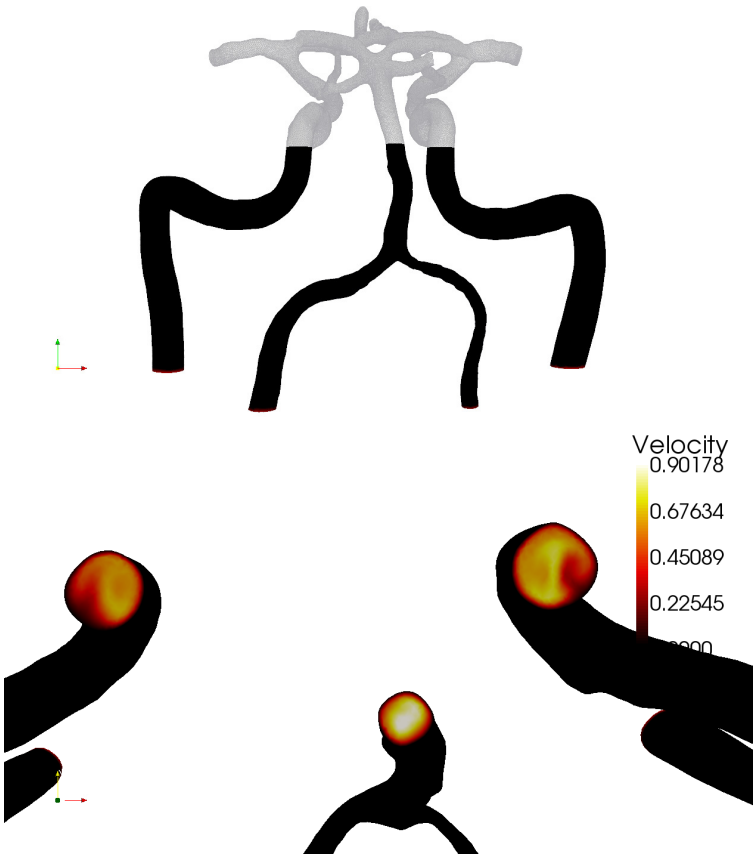


Figure 23.18: The image on the top shows the plane where the bottom image has been cut. The bottom image shows highly non-uniform flow in ICA.

24 Cerebrospinal fluid flow

By Susanne Støle-Hentschel, Svein Linge, Alf Emil Løvgren and Kent-Andre Mardal

This chapter concerns the flow of cerebrospinal fluid (CSF) in the subarachnoid space that surrounds the spinal cord. Particular attention is given to abnormal flow and pressure resulting from the Chiari I malformation and its often associated condition syringomyelia. The chapter builds on the software tools described in Chapter 21, and we will compare the Chorin, IPCS and G2 methods. In this chapter, we will also describe how to create meshes with Gmsh.

24.1 Medical background

CSF is a clear water-like fluid that occupies the subarachnoid space (SAS). It surrounds the brain and the spinal cord, and also fills the ventricular system within the brain. The SAS is bounded by strong tissue layers, the dura mater as the outer boundary and the pia mater as the inner boundary. A hole in the skull basis, foramen magnum, connects the cranial and spinal parts of the SAS. This hole is essential for CSF flow dynamics, since pulsating blood vessels in the brain cause the brain to expand and contract, a volume change that is made possible only by a simultaneous pulsating flow of CSF through the foramen magnum. Hence, the pulse that travels through the blood vessel network is transformed to a pulse in the CSF system, a pulse that is dampened on its way along the spinal canal. The CSF also plays an important role in cushioning the brain and the spinal cord.

The left picture in Figure 24.1 shows the CSF and the main structures in the brain of a healthy individual. In about 0.6% of the population the lower part of the cerebellum occupies parts of the CSF space in the upper spinal SAS and obstructs the CSF flow. This so-called Chiari I malformation (or Arnold-Chiari malformation) is shown in the right picture in Figure 24.1. A variety of symptoms are related to this malformation, including headache, abnormal eye-movement, motor or sensor-dysfunctions, etc. If the malformation is not treated surgically, the condition may become more severe and cause serious neurological deterioration, and may even lead to death.

Many people with the Chiari I malformation develop fluid filled cavities, often called syrinxes or cysts, within the spinal cord, a condition called syringomyelia. The exact relation between the Chiari I malformation and syringomyelia is not known. It is believed that flow and pressure disturbances caused by abnormal obstructions initiate the development of syringomyelia (Oldfield et al., 1994). Several authors have analyzed the relations between abnormal flow and syringomyelia development based on measurements in patients and healthy volunteers (Heiss et al., 1999; Pinna et al., 2000; Hofmann et al., 2000; Haughton et al., 2003). These studies also compare flow dynamics before and after decompressive surgery. The latter is an operation, where the SAS lumen around the obstructed area is increased by removing parts of the surrounding tissue and bone (Milhorat and Bolognese, 2003). Control images taken some weeks or months after the intervention often show a reduction of the size of the cyst in the spinal canal and patients usually report improvement of their condition. In

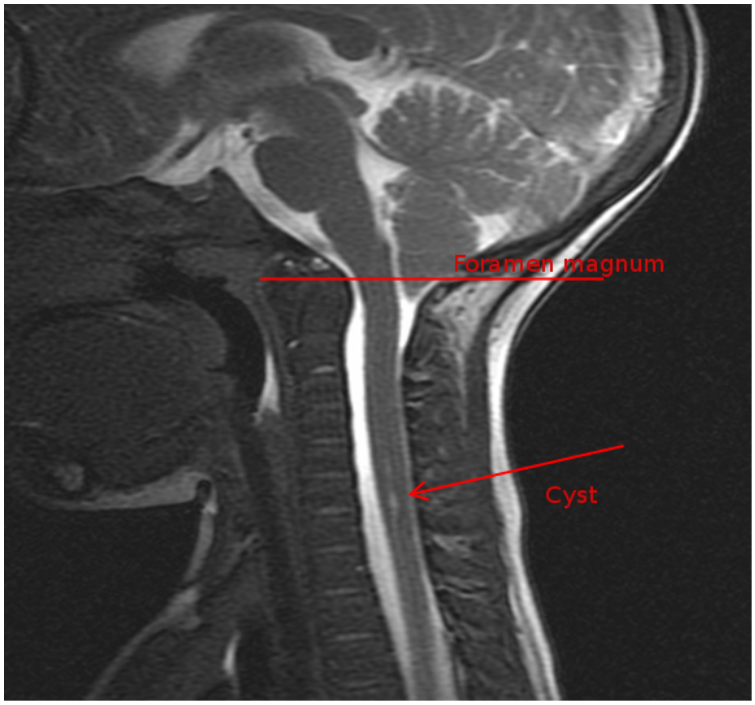


Figure 24.1: The picture shows an MR image of a patient with a Chiari I malformation. Chiari I patients are characterized by a very narrow passage at the foramen magnum; that is, the subarachnoid space, shown as white on the image, is small compared to normals. Chiari patients often develop cysts within the spinal cord, visible as white spots in the dark grey cord. Notice the relatively large distance between the narrow foramen magnum and the cyst.

some cases, the syrinx disappeared completely after some months (Oldfield et al., 1994; Pinna et al., 2000; Heiss et al., 1999).

Several studies (Quigley et al., 2004; Haughton et al., 2003) report that the measured CSF flow at foramen magnum is abnormal in the sense that the flow contains high speed jets and also synchronous bidirectional flow. Computational fluid dynamics (CFD) simulations have related the abnormal flow to abnormal pressure (Roldan et al., 2009; Hentschel et al., 2010; Linge et al., 2010, 2011). Many theories have been proposed to describe the relation between the Chiari I malformation and syringomyelia. However, it is hard to explain the relatively large distance between the Chiari I malformation and the cyst.

It is the purpose of this chapter to show how relevant CFD solvers in FEniCS may be used to investigate unresolved issues in CSF flow dynamics. Specifically, we investigate different boundary conditions, different geometries, and also how far velocity and pressure disturbances travel under realistic conditions. We also compare the different numerical schemes Chorin, IPCS and G2 described in Chapter 21.

24.2 Mathematical description

We model the CSF flow in the upper spinal canal as a Newtonian fluid with viscosity and density similar to water at body temperature. The upper spinal canal is represented as a tube with an inner elliptic or circular cylinder removed. In the presented experiments, we focus on the dynamics around the spinal cord. The tissue surrounding the fluid is modeled as impermeable and rigid throughout the cardiac cycle.

To simulate CSF flow, we apply the Navier–Stokes equations for an incompressible Newtonian

Symbol	Meaning	Unit	Value Used	Reference
$\mathbf{v}$	velocity variable	$\frac{\text{cm}}{\text{s}}$	—	$-1.3 \pm 0.6 \ldots 2.4 \pm 1.4$
p	pressure variable	$\frac{\text{dyne}}{\text{cm}^2}$	—	...
ρ	density	$\frac{\text{g}}{\text{cm}^3}$	—	0.993
μ	dynamic viscosity	$\frac{\text{gs}}{\text{cm}}$	—	0.0007
ν	kinematic viscosity	$\frac{\text{cm}^2}{\text{s}}$	0.710^{-2}	0.710^{-2}
SV	stroke volume	$\frac{\text{ml}}{\text{s}}$	0.27	0.27
HR	heart rate	$\frac{\text{beats}}{\text{s}}$	1.17	1.17
$A0$	tube boundary	cm^2	32	—
$A1$,$A2$	area of inlet/outlet	cm^2	0.93	$0.8 \ldots 1.1$
Re	Reynolds Number	–	–	70–200
We	Womersley Number	–	–	14–17

Table 24.1: Characteristic values and parameters for CSF flow modeling. The velocities are maximum absolute anterior CSF flow velocities taken from controls and Chiari I malformation patients (Hofmann et al., 2000). By stroke volume we mean the volume that moves up and down through cross section in the SAS during one cardiac cycle and the value is taken from (Gupta et al., 2009). Cross section areas 20–40 cm from the foramen magnum are taken from (Loth et al., 2001).

fluid,

$$\begin{aligned} \rho\left(\frac{\partial v}{\partial t} + v \cdot \nabla v\right) &= -\nabla p + \mu \Delta v + g, \\ \nabla \cdot v &= 0, \end{aligned} \tag{24.1}$$

with the variables as indicated in Table 24.1, and g, the body force; that is, gravity. We can eliminate gravity from the equation by assuming that the body force is balanced by the hydrostatic pressure. As a result, pressure describes only the dynamic pressure. To calculate the correct physical pressure, static pressure resulting from body forces has to be added. This simplification is not true, however, during sudden movements such as standing up.

24.3 Numerical experiments

24.3.1 Implementation

We refer to Chapter 21 for a complete description of the solvers and schemes implemented. In this chapter we concentrate on the use of these solvers in a few examples. Notice, however, that we use first order velocity elements, since the results with first order elements was virtually identical to the results with second order elements in this case. The code can be found in `csf_flow.py`.

Boundary conditions. The mesh boundaries at the inlet cross section, the outlet cross section, and the SAS boundaries are defined by the respective classes `Top`, `Bottom`, and `Contour`. They are implemented as subclasses of `SubDomain`, similarly to the given example of `Top`.

Python code

```
class Top(SubDomain):
    def __init__(self, z_index, z_max, z_min):
        SubDomain.__init__(self)
```

```
        self.z_index = z_index
        self.z_max = z_max
        self.z_min = z_min

    def inside(self, x, on_boundary):
        return bool(on_boundary and x[self.z_index] == self.z_max)
```

To define the domain correctly, we override the base class function `inside`. It returns a boolean evaluating if the inserted point `x` is part of the subdomain. The boolean `on_boundary` is very useful to easily partition the whole mesh boundary to subdomains.

It would be physically more correct to require that the no slip condition also is valid on the outermost/innermost nodes of the inflow and outflow sections as implemented below:

Python code

```
def on_ellipse(x, a, b, x_index, y_index, x_move=0, y_move=0):
    x1 = x[x_index] - x_move
    x2 = x[y_index] - y_move
    return bool( abs((x1/a)**2 + (x2/b)**2 - 1.0 ) < 10**-6 )
```

The vectors describing the ellipses of the cord and the dura in a cross section with the corresponding axes are required. The global function `on_ellipse` checks if `x` is on the ellipse defined by the x-vector `a` and the y-vector `b`. The variables `x_move` and `y_move` allow the definition of an eccentric ellipse.

Defining the inflow area at the top, with mantle nodes excluded, is done as shown in the following code. The outflow area at the bottom is defined analogously.

Python code

```
class Top(SubDomain): # bc for top
    def __init__(self, a2_o, a2_i, b2_o, b2_i, x_index, y_index, z_index, z_max, \
                 x2_o_move=0, y2_o_move=0, x2_i_move=0, y2_i_move=0):
        SubDomain.__init__(self)
        self.x_index = x_index
        self.y_index = y_index
        self.a2_o = a2_o
        self.a2_i = a2_i
        self.b2_o = b2_o
        self.b2_i = b2_i
        self.z_index = z_index
        self.z_max = z_max
        self.x2_o_move = x2_o_move
        self.x2_i_move = x2_i_move
        self.y2_o_move = y2_o_move
        self.y2_i_move = y2_i_move

    def inside(self, x, on_boundary):
        return on_boundary and abs(x[self.z_index] - self.z_max) < 10**-6 \
               and not on_ellipse(x, self.a2_o, self.b2_o, self.x_index, \
                                  self.y_index, self.x2_o_move, self.y2_o_move) \
               and not on_ellipse(x, self.a2_i, self.b2_i, self.x_index, \
                                  self.y_index, self.x2_i_move, self.y2_i_move))
```

The underscores `o` and `i` represent the outer and inner ellipse, respectively. The numbering with `2` distinguishes the subdomain at the top from that at the bottom, which may be defined differently. The details of how different problems can easily be defined in separate classes can be found in `src/mesh_definitions/`.

According to Gupta et al. (2009), a volume of 0.27 ml is transported back and forth through the spinal SAS cross sections during each cardiac cycle. For the average human, we assume a heart rate of 70 beats per minute. Furthermore, we define the cross sectional area to be $0.93\,\mathrm{cm}^2$, which matches a

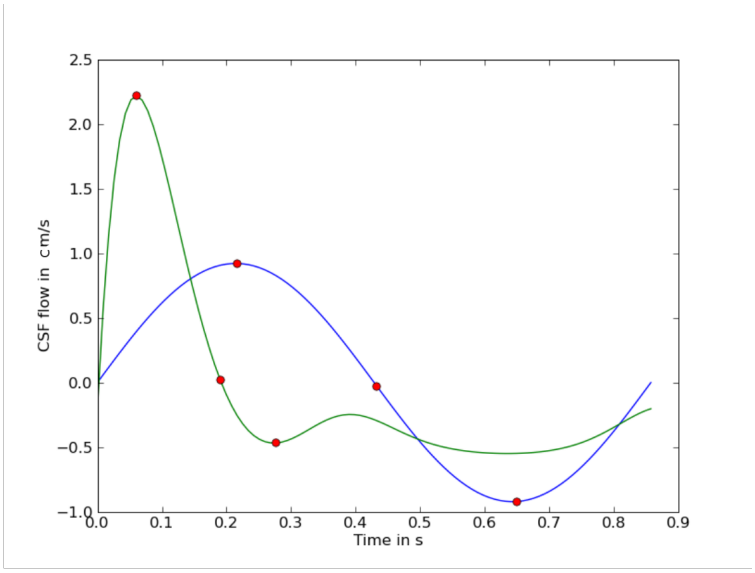


Figure 24.2: Two different CSF flow pulses. The blue line is a sine pulse, whereas the green is derived from the pressure pulse in a heart chamber.

segment from 20 to 40 cm down from the foramen magnum (Loth et al., 2001). In this region of the spinal canal, the cross sectional area varies little. In addition, the dura and the cord shape resembles a simple tube more than in other regions. According to Oldfield et al. (1994), syrinxes start at around 5 cm below the foramen magnum and extend up to 28 cm below the foramen magnum.

Moreover, we define a velocity pulse on the inflow and outflow boundaries, and since we are modeling incompressible flow between rigid impermeable boundaries, we must have equal inflow and outflow volumes at all times. The pulse values in these boundary cross sections were set equal in every grid point, and scaled to match the volume transport of 0.27 ml.

A function describing the varying blood pressure in a heart chamber is given in Smith et al. (2006). With some adjustment and additional parameters, the function was adapted to approximate the CSF flow pulse, see Figure 24.2. The systole of the pulse function is characterized by a high amplitude with a short duration while the negative counter movement has a reduced amplitude and lasts considerably longer. The global function for defining the pulse is:

Python code

```
def get_pulse_input_function(V, z_index, factor, A, HR_inv, HR, b, f1):
    C0 = 3.4 * pi
    rad = C0 /HR_inv
    v_z = "factor*(-A*(exp(-fmod(t, T)*rad)*Ees*(sin(-f1*fmod(t, T)*rad) - vd)
     - (1-exp(-factor*fmod(t, T)*rad))*p0*(exp(sin(-fmod(t, T)*rad) - vd) -1)) - b)"
    vel = ["0.0", "0.0", "0.0"]
    vel[z_index] = v_z
   defaults = {"factor":factor, "A":A, "p0":1, "vd":0.03, "Ees":50,
               "T": HR_inv, "HR":HR, "rad":rad, "b":b, "f1":f1}
    pulse = Expression(vel, defaults)
    return pulse
```

The following parameters have been used with this function.

Python code

```
A = 2.9/16
factor = self.flow_per_unit_area/0.324
v_max = 2.5 * self.factor
b = 0.465
f1 = 0.8
```

Initialization of the problem. The `Problem` class in `csf_flow`, derived from `ProblemBase` in `nsbench` described in Chapter 21, defines the mesh with its boundaries and provides the necessary information for the Navier–Stokes solvers. The mesh is ordered for all entities and initiated to compute its faces.

The values `z_min` and `z_max` mark the inflow and outflow coordinates along the tube's length axis. As mentioned above, the axis along the tube is indicated by `z_index`. If one of the coordinates, or the z-index, is not known, it may help to call the mesh in viper `unix>viper meshname.xml`. Typing `o` prints the length in x, y and z direction in the terminal window. Defining `z_min`, `z_max` and `z_index` correctly is important for the classes that define the boundary domains of the mesh `Top`, `Bottom` and `Contour`. As we have seen before, `z_index` is necessary to set the correct component to the nonzero boundary velocity.

Exterior forces on the Navier–Stokes flow are defined in the object variable `f`. Since gravity is neglected in the current problem formulation, the force function `f` is defined by a constant function `Constant` with value zero on the complete mesh.

After initializing the subdomains, `Top`, `Bottom` and `Contour`, they are marked with reference numbers attributed to the collection of all subdomains `sub_domains`.

To see the most important effects, the simulation was run slightly longer than one full period. A longer simulation time was not found necessary, since undesirable effects of the physiologically incorrect starting value (zero velocity) was dampened sufficiently already very early in the first period. Besides maximum and minimum velocities, the simulation includes the transition from diastole to systole, and vice versa. With the given physiological time scales of the problem, the chosen time step length (0.001 s) represents a high temporal resolution.

Python code

```
def __init__(self, options):
    ProblemBase.__init__(self, options)
    filename = "../../data/meshes/chiari/csf_extrude_2d_bd1.xml.gz"
    self.mesh = Mesh(filename)
    self.mesh.order()
    self.mesh.init(2)

    self.z_max = 5.0  # in cm
    self.z_min = 0.0  # in cm
    self.z_index = 2
    self.D = 0.5    # characteristic diameter in cm

    self.contour = Contour(self.z_index, self.z_max, self.z_min)
    self.bottom = Bottom(self.z_index, self.z_max, self.z_min)
    self.top = Top(self.z_index, self.z_max, self.z_min)

    # Load subdomain markers
    self.sub_domains = FacetFunction("uint", self.mesh)

    # Mark all facets as subdomain 3
    for i in range(self.sub_domains.size()):

        self.sub_domains.set(i, 3)
        self.contour.mark(self.sub_domains, 0)
        self.top.mark(self.sub_domains, 1)
        self.bottom.mark(self.sub_domains, 2)

        # Set viscosity
        self.nu = 0.7 * 10**-2 # cm^2/s

        # Create right-hand side function
        self.f = Constant((0.0, 0.0, 0.0))
        n = FacetNormal(self.mesh)
```

```
        # Set end-time
        self.T = 1.2 * 1.0/self.HR
        self.dt = 0.001
```

Increasing the time step length usually speeds up the calculation of the solution. As long as the CFL number with the maximum velocity v_{max}, time step length dt and minimal edge length h_{min} is smaller than one ($CFL = \frac{v_{max}dt}{h_{min}} < 1$), the solvers should converge. Too small time steps, however, can lead to an increasing number of iterations for the solver on each time step. As a characterization of the fluid flow, the Reynolds number ($Re = \frac{v_c l}{\nu}$) was calculated with the maximum velocity v_c at the inflow boundary and the characteristic length l of the largest gap between outer and inner boundary. A listing of Reynolds and Womersley numbers under different scenarios is given in the end of the chapter.

The area of the mesh surfaces and the mesh size can be found as follows.

Python code

```
def __init__(self, ...)
    ...
    self.h = MeshSize(self.mesh)
    self.A0 = self.area(0)
    self.A1 = self.area(1)
    self.A2 = self.area(2)

def area(self, i):
    f = Constant(1)
    A = f*ds(i)
    a = assemble(A, exterior_facet_domains=self.sub_domains)
    return a
```

Function objects. Being a subclass of `ProblemBase`, `Problem` overrides the object functions `update` and `functional`. The first ensures that all time-dependent variables are updated for the current time step. The latter prints the maximum values for pressure and velocity. The normal flux through the boundaries is defined in the separate function `flux`.

Python code

```
def update(self, t, u, p):
    self.g1.t = t
    self.g2.t = t

def functional(self, t, u, p):
    v_max = u.vector().norm(linf)
    f0 = self.flux(0,u)
    f1 = self.flux(1,u)
    f2 = self.flux(2,u)
    pressure_at_peak_v = p.vector()[0]

    # if current velocity is peak
    if v_max > self.peak_v:
        self.peak_v = v_max
        self.pressure_at_peak_v = pressure_at_peak_v

    return pressure_at_peak_v

def flux(self, i, u):
    n = FacetNormal(self.mesh)
    A = dot(u,n)*ds(i)
```

```
    a = assemble(A, exterior_facet_domains=self.sub_domains)
    return a
```

The boundary conditions are all given as Dirichlet conditions, associated with their velocity function space and the belonging subdomain. The additional functions `boundary_conditions` and `initial_conditions` define the respective conditions for the problem that are called by the solver. Boundary conditions for velocity and pressure are collected in the lists `bcv`, `bcp` and `bcpsi`.

Python code

```
def boundary_conditions(self, V, Q):

    # Create no-slip boundary condition for velocity
    self.g0 = Constant((0.0, 0.0, 0.0))
    bc0 = DirichletBC(V, self.g0, self.contour)

    # Create function for inlet and outlet BC
    self.g1 = get_sine_input_function(V, self.z_index, self.HR, self.HR_inv,
                                      self.v_max)
    self.g2 = self.g1

    # Create inflow boundary condition for velocity on side 1 and 2
    bc1 = DirichletBC(V, self.g1, self.top)
    bc2 = DirichletBC(V, self.g2, self.bottom)

    # Collect boundary conditions
    bcv = [bc1, bc0, bc2]
    bcp = []
    bcpsi = []

    return bcv, bcp, bcpsi

def initial_conditions(self, V, Q):
    u0 = Constant((0.0, 0.0, 0.0))
    p0 = Constant(0.0)
    return u0, p0
```

Running. Applying the "Chorin" solver, the Problem is started by typing :

Bash code

```
./ns csf_flow chorin
```

It approximates the Navier–Stokes equation with Chorin's method. The progress of different simulation steps and results, including maximum calculated pressure and velocity per time step, are printed out on the terminal. In addition, the solution for pressure and velocity are dumped to a file for each time step. Before investigating the results, we introduce how the mesh is generated.

24.3.2 *Mesh generation with Gmsh*

In this section we go briefly through the basic usage of Gmsh (Geuzaine and Remacle). The following code example shows the construction of a circular cylinder (representing the pia on the spinal cord) within an elliptic cylinder (representing the dura). We define a characteristic length scale `lc`, which is used in the definition of each point `Point`, which takes three coordinates and the characteristic length scale. The dura is defined by the ellipse vectors a=0.65 mm and b=0.7 mm in x and y direction, respectively. The cord has a radius of 4 mm with its center moved 0.8 mm in positive x-direction Since Gmsh only allows circular or elliptic arcs to be drawn for angles smaller than π, the basic ellipses

were constructed from four arcs each. Every arc is defined by the starting point, the center, another point on the arc and the end point. The value *lc* defines the maximal edge length in vicinity to the point.

Code

```
lc = 0.04;  //characteristic length for the cell
Point(1) = {0,0,0,lc};  // center point (x,y,z,lc)
//outer ellipses
a = 0.65;
b = 0.7;
Point(2) = {a,0,0,lc};
Point(3) = {0,b,0,lc};
Point(4) = {-a,0,0,lc};
Point(5) = {0,-b,0,lc};
Ellipse(10) = {2,1,3,3};
Ellipse(11) = {3,1,4,4};
Ellipse(12) = {4,1,5,5};
Ellipse(13) = {5,1,2,2};

// inner ellipses
move = 0.08; //"move" center
Point(101) = {move,0,0,lc};
c = 0.4;
d = 0.4;
Point(6) = {c+move,0,0,lc*0.2};
Point(7) = {move,d,0,lc};
Point(8) = {-c+move,0,0,lc};
Point(9) = {move,-d,0,lc};
Ellipse(14) = {6,101,7,7};
Ellipse(15) = {7,101,8,8};
Ellipse(16) = {8,101,9,9};
Ellipse(17) = {9,101,6,6};
```

The constructed ellipses are composed of separate arcs. To define them as single lines, the ellipse arcs are connected in line loops as follows:

Code

```
// connect lines of outer and inner ellipses to one
Line Loop(20) = {10,11,12,13};    // only outer
Line Loop(21) = {-14,-15,-16,-17};  // only inner
```

The SAS surface between cord and dura is then defined by the following command.

Code

```
Plane Surface(32) = {20,21};
```

Finally, to construct volumes, Gmsh contains the command `Extrude`, which will extrude the surface over a given length.

Code

```
length = 5.0
csf[] = Extrude(0,0,length){Surface{32};};
```

We store the above Gmsh commands in a `.geo` file that is read by Gmsh as:

Bash code

```
> gmsh filename.geo
```

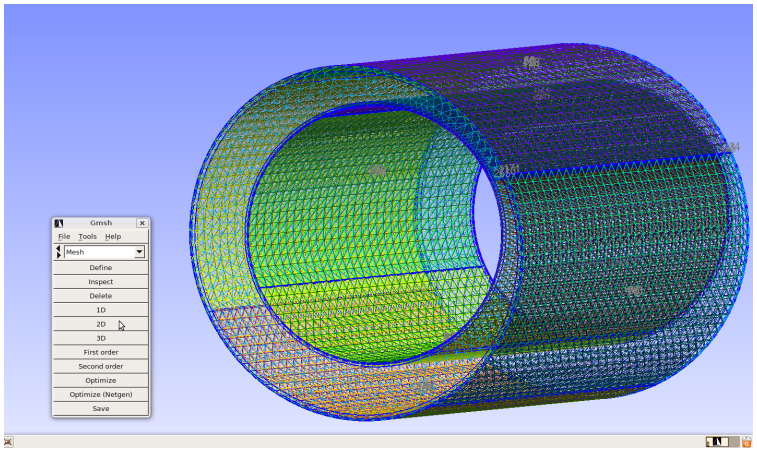


Figure 24.3: The figure shows the user-interface of Gmsh during the making of a mesh.

A screen shot of an interactive session is shown in Figure 24.3. We may then change to "Mesh modus" in the interactive panel and press `3d` to construct the mesh. Pressing `Save` will save the mesh under the name "filename", with extension `msh`. For use in DOLFIN, we apply the DOLFIN converter.

Bash code

```
> dolfin-convert filename.msh filename.xml
```

To capture possible sharp gradients close to the boundary we introduce a few boundary layers. This is obtained by adding mesh layers; that is, copies of the elliptic arcs that are gradually scaled to increase the maximum edge length. The code example below shows the creation of the layers close to the outer ellipse. The inner layers are created similarly.

Code

```
outer_b1[] = Dilate {{0, 0, 0}, 1.0 - 0.1*lc } {
Duplicata{  Line{10}; Line{11}; Line{12}; Line{13}; } };
outer_b2[] = Dilate {{0, 0, 0}, 1.0 - 0.3*lc } {
Duplicata{  Line{10}; Line{11}; Line{12}; Line{13}; } };
outer_b3[] = Dilate {{0, 0, 0}, 1.0 - 0.8*lc } {
Duplicata{  Line{10}; Line{11}; Line{12}; Line{13}; } };
```

Here the command `Duplicata` copies the expressions, while `Dilate` scales them. The single arcs are dilated separately since the arc points are necessary for further treatment. Remember that no arcs with angles smaller than π are allowed. Again we need a representation for the complete ellipses defined by line loops, as

Code

```
Line Loop(22) = {outer_b1[]};
```

that are necessary to define the surfaces between all neighboring ellipses similar to:

Code

```
Plane Surface(32) = {20,22};
```

Finally, all surfaces are listed in the Extrude command.

The test meshes of 1.75 cm seemed to have a fully developed region around the mid-cross sections, right where we want to observe the flow profile. Testing different numbers of tubular layers for the length of 1.75, 2.5 and 5 cm showed that the above mentioned observations of wave-like structures occurred less for longer pipes, even though the number of layers was low compared to the pipe length.

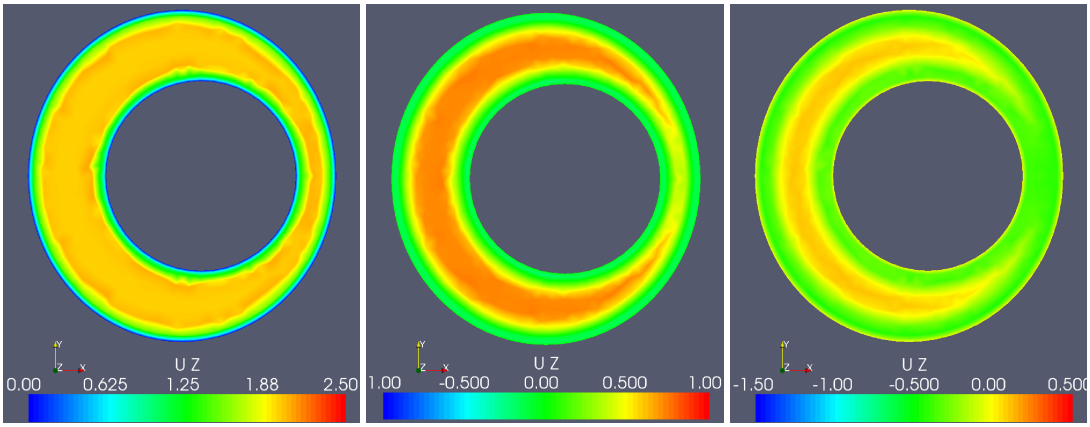


Figure 24.4: The case with a circular cord and the boundary condition based on the pressure pulse in the heart (Smith et al., 2006). The pictures show the velocity in z-direction for the non-symmetric pulse at the time steps $t = 0.07s, 0.18s, 0.25s$. Notice that we use different color scales for the different time steps. The same color scales will be used in all subsequent figures.

The presented results were simulated on meshes of length 5 cm with 30 layers in z-direction and three layers on the side boundaries. The complete code can be found in `csf/mesh_generation/gmsh`.

24.3.3 Example 1: simulation of a pulse in the SAS

In the first example we represent the spinal cord and the surrounding dura mater as two straight non-concentric cylinders, created using Gmsh. The simulation results for an appropriate mesh can be found in Figure 24.4. The plots show the velocity component in tubular direction at the mid-cross section of the model. The flow profiles are taken at the time steps of maximum flow in both directions and during the transition from systole to diastole. For maximal systole, the velocities have two peak rings, one close to the outer, the other close to the inner boundary. We can see sharp profiles at the maxima and bidirectional flow at the local minimum during diastole.

The cysts usually develop several centimeters below the Chiari I malformation. It is therefore of interest to quantify how far pressure and velocity instabilities can travel under realistic conditions. To create pressure and velocity instabilities, we assign time-dependent but flat inlet and outlet velocity conditions on the previously described geometry and investigate how far these instabilities travel with the flow. We also compare three different solvers, namely the Chorin, IPCS, and G2 schemes. All schemes do, however, use first order elements for both velocity and pressure. In Table 24.2, we list the pressure differences at various places along the spinal cord computed with the three different solvers, slightly after systole. Strangely, it appears that the G2 solver requires a 30% bigger pressure gradient between the top and the bottom in this case. However, the G2 solver has also produced some peculiar results in Chapter 21. We also remark that G2 was about 15 times slower than Chorin and IPCS. Our main interest here, however, is how far the pressure instabilities travel. All the solvers are consistent on this point, the pressure instabilities do not travel very far. After 1 cm, the pressure instabilities is lessened by a factor 5-10.

24.3.4 Example 2: simplified boundary conditions

Many researchers apply the sine function as inlet and outlet boundary conditions, since its integral is zero over one period. However, its shape is not in agreement with measurements of the cardiac flow pulse. To see the influence of the applied boundary condition for the defined mesh, we replaced

Pressure (Pa) Solver	Chorin	IPCS	G2
$\Delta_z p$ 5.0 cm	99.6	101.3	130
$\Delta_{xy} p$ 0.1 cm	2.1	2.6	0.9
$\Delta_{xy} p$ 0.5 cm	0.8	1.1	0.4
$\Delta_{xy} p$ 1.0 cm	0.3	0.3	0.2
$\Delta_{xy} p$ 2.0 cm	0.03	0.03	0.01

Table 24.2: Pressure differences between various places in the spinal canal. The first row, Δ_z5.0 cm list the pressure difference between the top and bottom. The next rows list pressure differences, $\Delta_{xy} p$ in the cross sections 0.1 cm, 0.5 cm, 1.0 cm, and 2.0 cm down from the top.

the more realistic pulse function with a sine, scaled to the same amount of volume transport per cardiac cycle. The code example below implements the alternative pulse function in the object function `boundary_conditions`. The variable `sin_integration_factor` describes the integral of the first half of a sine.

Python code

```
self.HR = 1.16 # heart rate in beats per second; from 70 bpm
self.HR_inv = 1.0/self.HR
self.SV = 0.27
self.A1 = self.area(1)
self.flow_per_unit_area = self.volume_flow/self.A1
sin_integration_factor = 0.315
self.v_max = self.flow_per_unit_area/sin_integration_factor
```

As before, we have a global function returning the sine as an object:

Python code

```
def get_sine_input_function(V, z_index, HR, HR_inv, v_max):
    v_z = "sin(2*pi*HR*fmod(t,T))*(v_max)"
    vel = ["0.0", "0.0", "0.0"]
    vel[z_index] = v_z
    defaults = {"HR":HR, "v_max":v_max, "T":HR_inv}
    sine = Expression(vel, defaults)
    return sine
```

that is called instead of `get_pulse_input_function` in the function `boundary_conditions`:

Python code

```
self.g1 = get_sine_input_function(V, self.z_index, self.factor, self.A,
                                  self.HR_inv, self.HR, self.b, self.f1)
```

The pulse and the sine are sketched in Figure 24.2. Both functions are marked at the points of interest: maximum systolic flow, around the transition from systole to diastole and the (first, local) minimum. Results for sinusoidal boundary conditions are shown in Figure 24.5 The shape of the flow profile is similar at every time step, only the magnitudes change. No bidirectional flow was discovered in the transition from positive to negative flow. Compared to the results received by the more realistic pulse function, the velocity profile generated from sinusoidal boundaries is more homogeneous over the cross section.

24.3.5 Example 3: cord shape and position

According to Loth et al. (2001); Alperin et al. (2006), the present flow is inertia dominated, meaning that the shape of the cross section should not influence the pressure gradient. Changing the length of

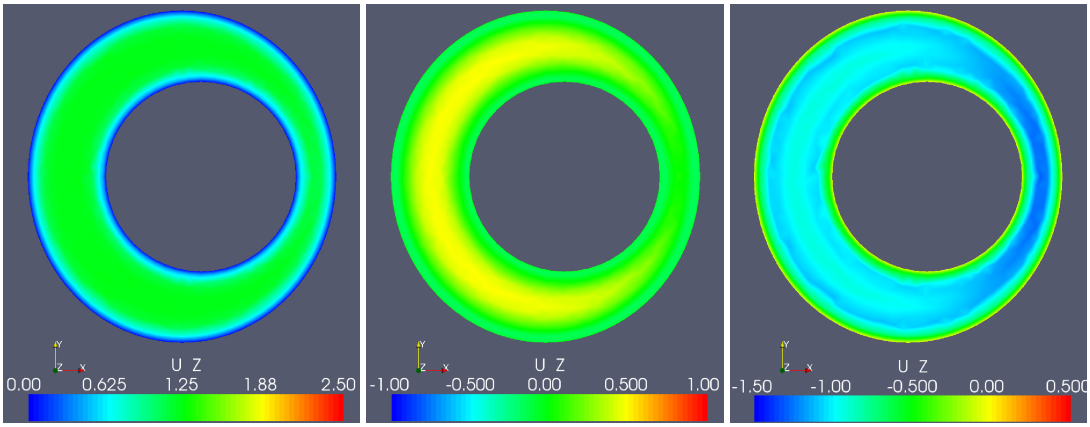


Figure 24.5: The case with circular cord and the boundary condition based on a sine function. The pictures show the velocity in z-direction as response to a sine boundary condition for the time steps $t = 0.2, 0.4, 0.6$.

vectors describing the ellipse from

Code

```
c = 0.4;
d = 0.4;
```

to

Code

```
c = 0.32;
d = 0.5;
```

transforms the cross section of the inner cylinder to an elliptic shape with preserved area. The simulation results are collected in Figure 24.6. The geometry differences between the two cases gave different flow profiles but no differences in the pressure gradient.

A further perturbation of the SAS cross sections was achieved by changing the displacement of the elliptic cord center from

Code

```
move = 0.08;
```

to

Code

```
move = 0.16;
```

Also for this case the pressure field was identical, with some variations in the flow profiles, see Figure 24.7.

24.3.6 Example 4: cord with syrinx

Syrinxes expand the cord so that it occupies more space of the spinal SAS. Increasing the cord radius from 4 mm to 5 mm [1] decreases the cross sectional area by almost one third to 0.64 cm^2. The resulting flow is shown in Figure 24.8. Apart from the increased velocities, we see bidirectional flow already at

[1] Similar to setting the variables c and d in the geo-file to 0.5.

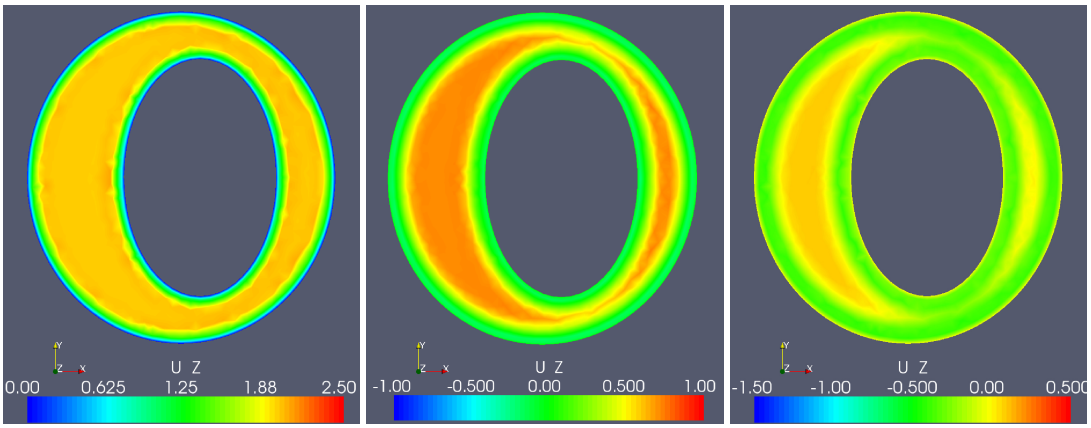


Figure 24.6: The case with an elliptic cord and the boundary condition based on the pressure pulse in the heart. The pictures show the velocity in z-direction for the non-symmetric pulse at the time steps $t = 0.07s, 0.18s, 0.25s$.

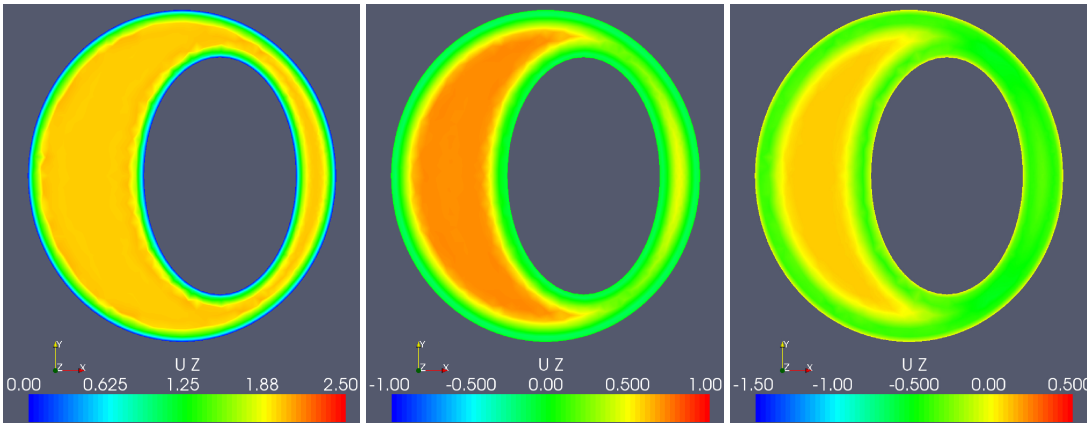


Figure 24.7: The case with a translated elliptic cord. The pictures show the velocity in z-direction for the non-symmetric pulse at the time steps $t = 0.07s, 0.18s, 0.25s$.

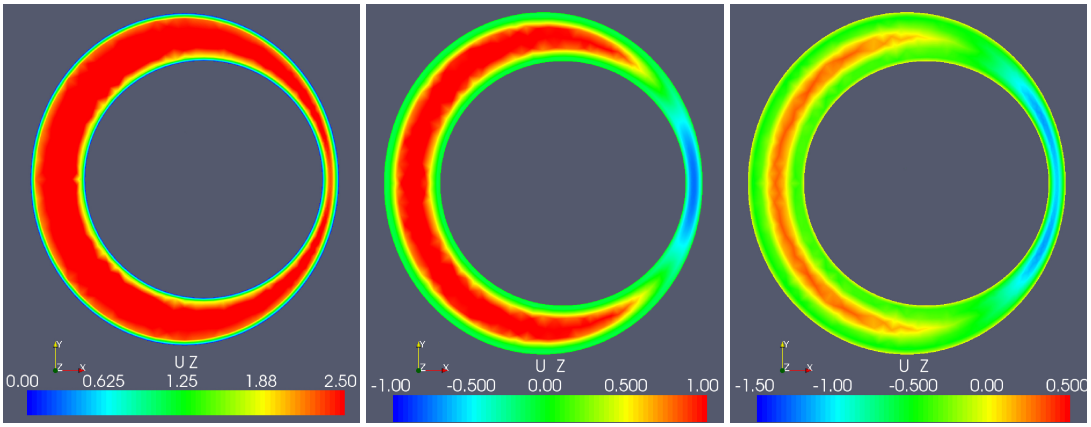


Figure 24.8: The case with an enlarged cord diameter and the boundary conditions based on the pressure pulse. The pictures show the velocity in z-direction for the non-symmetric pulse at the time steps $t = 0.07s, 0.18s, 0.25s$.

Problem	D , in cm	$\mathbf{v}_{max}$ in cm/s	Re	We
Example 1	0.54	2.3	177	17
Example 2	0.54	0.92	70	17
Example 4	0.45	3.2	205	14

Table 24.3: Characteristic values for the examples 1, 2 and 3. Here, the characteristic length is $D = \sqrt{(A/\pi)}$, where A is the inflow and outflow area. Furthermore, $\mathbf{v}_{max}$ is defined as the maximal velocity at inflow and outflow boundary.

t = 0.18 s and at t = 0.25 s as before. The fact that diastolic back-flow is visible at t = 0.18 s, shows that the pulse with its increased amplitude travels faster.

Comparing Reynolds and Womersley numbers shows a clear difference for the above described examples 1, 2 and 3. Example 2 is marked by a clearly lower maximum velocity at inflow and outflow boundary that leads to a rather low Reynolds number. Due to the different inflow and outflow area, example 4 has a lower Womersley number, leading to an elevated maximum velocity at the boundary and clearly increased Reynolds number. These numbers help to quantify the changes introduced by variations in the model. For the chosen model, the shape of the pulse function at the boundary as well as the cross sectional area have great influence on the simulation results. Comparison of Reynolds numbers for different scenarios can be found in Table 24.3.

24.4 Conclusions

In this chapter, we have presented the use of FEniCS to simulate CSF flow in various idealized geometries representing the spinal cord and the surrounding subarachnoid space. We have further quantified the effect of abnormal geometries and boundary conditions in terms of pressure and flow deviations. From our simulations, it seems that pressure instabilities are quickly damped out under realistic Reynolds and Womersley numbers. These instabilities travel less than 1 cm and it seems unlikely that Chiari induced pressure instabilities will produce cysts several centimeters further down in the spinal canal. We have observed that the velocity changes quite a bit with varying shape and position of the cord. The pressure does, however, not change much. The size of the cross section area does have an impact on the pressure, as expected. Finally, we observed that the pressure computed using the G2 method differed significantly from the pressure computed by Chorin and IPCS.

25 *Improved Boussinesq equations for surface water waves*

By Nuno D. Lopes, Pedro J. S. Pereira and Luís Trabucho

The main motivation of this work is the implementation of a general finite element solver for some of the improved Boussinesq models. Here, we use an extension of the model proposed by Zhao et al. (2004) to investigate the behavior of surface water waves. The equations in this model do not contain spatial derivatives with an order higher than two. Some effects like energy dissipation and wave generation by natural phenomena or external physical mechanisms are also included. As a consequence, some modified dispersion relations are derived for this extended model. A matrix-based linear stability analysis of the proposed model is presented. It is shown that this model is robust with respect to instabilities related to steep bottom gradients.

25.1 Overview

The FEniCS project, via DOLFIN, UFL and FFC, provides good technical and scientific support for the implementation of large scale industrial models based on the finite element method. Specifically, all the finite element matrices and vectors are automatically generated and assembled by DOLFIN and FFC, directly from the variational formulation of the problem which is declared using UFL. Moreover, DOLFIN provides a user friendly interface for the libraries needed to solve the finite element system of equations.

Numerical implementation of Boussinesq equations goes back to the works of Peregrine (1967) and Wu (1981), and later by the development of improved dispersion characteristics (see, e.g., Madsen et al. (1991); Nwogu (1993); Chen and Liu (1994) as well as Beji and Nadaoka (1996)).

We implement a solver for some of the Boussinesq type systems to model the evolution of surface water waves in a variable depth seabed. This type of model is used, for instance, in harbor simulation (see Figure 25.1 for an example of a standard harbor), tsunami generation and wave propagation as well as in coastal dynamics.

In Section 25.2, we begin by describing the DOLFWAVE application which is a FEniCS based application for the simulation of surface water waves (see `https://launchpad.net/dolfwave`).

The governing equations for surface water waves are presented in Section 25.3. From these equations different types of models can be derived. There are several Boussinesq models and some of the most widely used are those based on the wave surface Elevation and horizontal Velocities formulation (BEV) (see, e.g., Walkley and Berzins (2002), Woo and Liu (2004a) as well as Woo and Liu (2004b) for finite element discretizations of BEV models). However, we only consider the wave surface Elevation and velocity Potential (BEP) formulation (see, e.g., Langtangen and Pedersen (1998) for a finite element discretization of a BEP model). Thus, the number of unknowns is reduced from five (the three velocity components, the pressure and the wave surface elevation) in the BEV models to

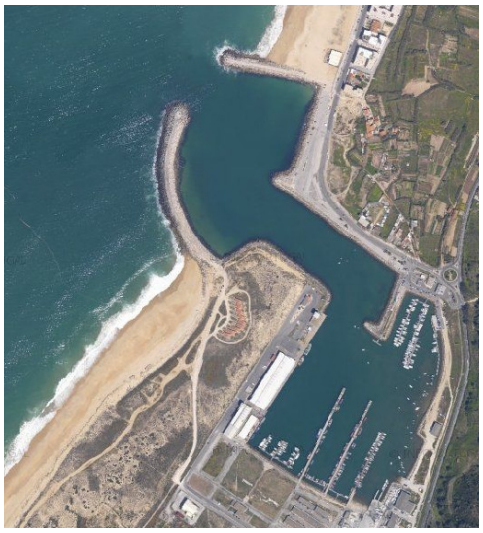


Figure 25.1: Nazaré's harbor, Portugal.

three (the velocity potential, the pressure and the wave surface elevation) in the BEP models. Two different types of BEP models are taken into account:

1. a standard model containing sixth-order spatial derivatives;

2. the model proposed by Zhao et al. (2004) (ZTC), containing only first and second-order spatial derivatives.

A standard technique is used in order to derive the Boussinesq-type model mentioned in 1. In the subsequent sections, only the ZTC-type model is considered. Note that these two models are complemented with some extra terms, due to the inclusion of effects like energy dissipation and wave generation by moving an impermeable bottom or using a source function.

An important characteristic of the extended ZTC model, including dissipative effects, is presented in Section 25.4, namely, the dispersion relation.

Section 25.5 is dedicated to the numerical methods used in the discretization of the variational formulation. The discretization of the spatial variables is accomplished with Lagrange P_1 or P_2 finite elements (see Chapter 3) whereas the time integration is implemented using Runge–Kutta and predictor-corrector algorithms.

In Sections 25.6 and 25.7, we describe several types of wave generation, absorption and reflection mechanisms. Initial conditions for a solitary wave and a periodic wave induced by Dirichlet boundary conditions are also presented. Moreover, the extended ZTC model includes a source function to generate surface water waves, as proposed in Wei et al. (1999). Total reflective walls are modelled by standard zero Neumann conditions for the wave surface elevation and velocity potential. The wave energy absorption is simulated using sponge layers.

In Section 25.8, we use a matrix-based analysis in order to study some stability properties of the linearized ZTC model in one horizontal dimension and with a time-independent bathymetry. The standard potential model with depth averaged velocity potential investigated by Løvholt and Pedersen (2009) is also used here as a reference for comparison.

In Section 25.9, the extended ZTC equations are used to model four different physical problems: the evolution of solitary waves passing through submerged bars with different geometries, the evolution of a Gaussian hump in a square basin, the evolution of a periodic wave in a harbor geometry like that one represented in Figure 25.1 and the generation of a wave due to an object moving on a horizontal bottom. We also use the first numerical test to illustrate the usage of the DOLFWAVE application.

Other solvers, mostly based on finite difference methods, have been proposed in the literature, such like FUNWAVE (see Kirby (1998)), COULWAVE (see Lynett and Liu (2004)) and the global Boussinesq solver by Pedersen and Løvholt (2008). FUNWAVE is based on the fully nonlinear and BEV equations by Wei and Kirby (1995). Wave generation by source function, wave breaking, bottom friction, treatment of moving shorelines, subgrid turbulent mixing, totally reflective walls and sponge layers are included in this model. The third-order spatial derivative equations in this model are discretized using a fourth-order finite difference scheme. Specifically, for time integration, a composite fourth-order Adams–Bashforth–Moulton scheme (third-order Adams–Bashforth predictor step and a fourth-order Adams–Moulton corrector-step) is used. Moreover, a fourth-order Shapiro type filter is applied to remove short length waves. The main cause of instabilities in nonlinear shallow water computations is often due to high nonlinearity in shallow water. The instabilities appear through fast growing short wavelengths which eventually cause the blowup of the model.

COULWAVE possesses essentially the main features of FUNWAVE plus the inclusion of wave generation due to a moving bottom. Moreover, COULWAVE is based on a multilayer third-order spatial derivative BEV model. This leads to improved dispersion relations and nonlinear properties when compared with FUNWAVE. However, the number of primary unknowns in this two horizontal dimensional model with n-layers increases from 3 to $2n + 1$. Thus, the computational time is also increased, accordingly.

The global Boussinesq solver by Pedersen and Løvholt (2008) is based on BEV and BEP models, although the BEV versions are preferred by the authors. In Løvholt and Pedersen (2009) several BEV and BEP models are studied regarding linear stability properties. It was shown that the tested BEV formulations are less prone to instabilities due to steep depth gradients than some BEP ones. This solver includes Coriolis effect and the modification on arc lengths by the curvature of the Earth. The main goal of this solver is to treat, efficiently and in a robust way, large scale ocean waves such as tsunamis. The shoreline runup, breaking waves or generation of waves by moving bottoms are not yet included. Only standard nonlinear terms are considered and finite differences methods are used to discretize the model, specifically C-grid is implemented for the spatial discretization and a leap-frog scheme is used for the time stepping.

25.2 DOLFWAVE

The main goal of DOLFWAVE is to provide a framework for the analysis, development and computation of models for surface water waves, based on finite element methods. Algorithms for shoreline runup/rundown, numerical filters or effects like wave breaking or bottom friction are not yet included in the application, however they are planned for future implementation.

We have already implemented solvers for the following cases:

1. Shallow water wave models for unidirectional long waves in one horizontal dimension;
2. Boussinesq-type models for moderately long waves with small amplitude in shallow water.

The shallow water wave models implemented and mentioned in 1 are the following:

- The Korteweg–de Vries (KdV) model which consists of a weakly nonlinear and dispersive third-order partial differential equation for the wave surface elevation. The discretization of the spatial variable is accomplished using a continuous/discontinuous finite element method with Lagrange P_2 elements;
- The Benjamin–Bona–Mahony model, also known as the regularized long-wave (RLW) model, which is an improvement of the KdV model regarding the dispersive properties. The equation in the RLW model contains only second-order spatial derivatives. The discretization of the

spatial variable is accomplished using continuous finite element methods with Lagrange P_1 or P_2 elements.

For the Boussinesq-type models we considered the following cases:

- The extended ZTC model which is based on a system of two second-order partial differential equations for the wave surface elevation and a velocity potential. The discretization of the spatial variables is accomplished using continuous finite element methods with Lagrange P_1 or P_2 elements;
- An extension of the model by Chen and Liu (1994) in order to include dissipative effects, several forms of wave generation and improved dispersive properties. This model is based on a system of two fourth-order partial differential equations for the wave surface elevation and a velocity potential. The discretization of the spatial variables is accomplished using the continuous/discontinuous finite element method with Lagrange P_2 elements (see Lopes et al. (2011)).

A predictor-corrector scheme with an initialization provided by an explicit Runge–Kutta method is used for the time integration of the equations. These schemes are easier to implement and require smaller computational times than the implicit ones. However, they are more prone to numerical instabilities and in general require smaller time steps.

We use UFL for the declaration of the finite element discretization of the variational forms related to the models mentioned above (see the UFL form files in the following directories of the DOLFWAVE code tree: `dolfwave/src/1hd1sforms`, `dolfwave/src/1hdforms` and `dolfwave/src/2hdforms`). These files are compiled using FFC to generate the C++ code of the finite element discretization of the variational forms (see Chapter 17). DOLFWAVE is based on the C++ interface of DOLFIN 1.0 to assemble and solve all the systems of equations related with the finite element method.

All the DOLFWAVE code is available for download at `https://launchpad.net/dolfwave`. Some tools for the generation of the C++ code for boundary conditions and source functions are included. Scripts for visualization and data analysis are also part of the application. The Xd3d post-processor is used in some cases (see `http://www.cmap.polytechnique.fr/~jouve/xd3d/`). DOLFWAVE has a large number of demos covering all the implemented models (see `dolfwave/demo`). Different physical effects are illustrated. All the numerical examples in this work are included in the demos.

25.3 Model derivation

We consider the following set of equations for the irrotational flow of an incompressible and inviscid fluid:

$$\frac{\partial u}{\partial t} + u \cdot \nabla u = -\nabla \left(\frac{P}{\rho} + g\, z \right), \tag{25.1a}$$

$$\nabla \times u = 0, \tag{25.1b}$$

$$\nabla \cdot u = 0, \tag{25.1c}$$

where u is the velocity vector field of the fluid, P the pressure, g the gravitational acceleration, ρ the mass per unit volume, t the time and the differential operator $\nabla = \left(\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z} \right)$. A Cartesian coordinate system is adopted with the horizontal x and y-axes on the still water plane and the z-axis pointing vertically upwards (see Figure 25.2). The fluid domain is bounded by the bottom seabed at $z = -h(x, y, t)$ and the free water surface at $z = \eta(x, y, t)$. In Figure 25.2, L, A and H are the

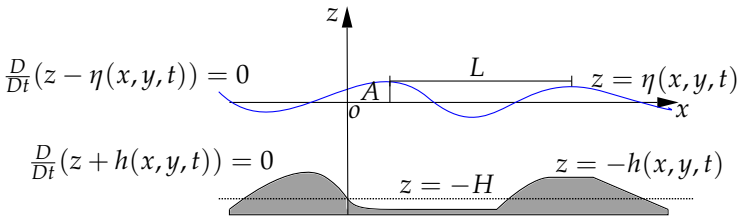


Figure 25.2: Cross-section of the water wave domain.

characteristic wave length, wave amplitude and depth, respectively. Note that the material time derivative is denoted by $\frac{D}{Dt}$.

From the irrotational assumption (see (25.1b)), we can introduce a velocity potential function, $\phi(x,y,z,t)$ to obtain Bernoulli's equation:

$$\frac{\partial\phi}{\partial t}+\frac{1}{2}\nabla\phi\cdot\nabla\phi+\frac{P}{\rho}+g\,z=f(t), \tag{25.2}$$

where $u=\nabla\phi(x,y,z,t)$ and $f(t)$ stands for an arbitrary function of integration. Note that we can remove $f(t)$ from equation (25.2) if ϕ is redefined by $\phi+\int f(t)\,\mathrm{d}t$. From the incompressibility condition (see (25.1c)) the velocity potential satisfies Laplace's equation:

$$\nabla^2\phi+\frac{\partial^2\phi}{\partial z^2}=0, \tag{25.3}$$

where, from now on, ∇ denotes the horizontal gradient operator given by $\nabla=\left(\frac{\partial}{\partial x},\frac{\partial}{\partial y}\right)$. To close this problem, the following boundary conditions must be satisfied:

1. the kinematic boundary condition for the free water surface:

$$\frac{\partial\phi}{\partial z}=\frac{\partial\eta}{\partial t}+\nabla\phi\cdot\nabla\eta,\quad z=\eta; \tag{25.4}$$

2. the kinematic boundary condition for the impermeable sea bottom:

$$\frac{\partial\phi}{\partial z}+(\nabla\phi\cdot\nabla h)=-\frac{\partial h}{\partial t},\quad z=-h; \tag{25.5}$$

3. the dynamic boundary condition for the free water surface:

$$\frac{\partial\phi}{\partial t}+g\eta+\frac{1}{2}\left(|\nabla\phi|^2+\left(\frac{\partial\phi}{\partial z}\right)^2\right)+D(\phi)=0,\quad z=\eta, \tag{25.6}$$

where $D(\phi)$ is a dissipative term (see, e.g., the work by Dutykh and Dias (2007)). We assume that this dissipative term is of the following form:

$$D(\phi)=\nu\frac{\partial^2\phi}{\partial z^2}, \tag{25.7}$$

with $\nu=\bar{\mu}/\rho$ and $\bar{\mu}$ an eddy-viscosity coefficient. Note that a non-dissipative model means that there is no energy loss. This is not acceptable from a physical point of view, since any real flow is accompanied by energy dissipation.

Using Laplace's equation (see (25.3)) it is possible to rewrite (25.7) as $D(\phi)=-\nu\nabla^2\phi$. Throughout the literature, analogous terms were added to the kinematic and dynamic conditions to absorb the

wave energy near the boundaries. These terms are related with the sponge or damping layers and, as we will see later, they can be used to modify the dispersion relations.

A more detailed description of the above equations is found in the reference book on waves by Whitham (1974), or in the more recent book by Johnson (1997).

25.3.1 Standard models

In this subsection, we present a generic Boussinesq system using the velocity potential formulation. To transform equations (25.2)– (25.7) in a dimensionless form, the following scales are introduced (see Figure 25.2):

$$(x', y') = \frac{1}{L}(x, y), \quad z' = \frac{z}{H}, \quad t' = \frac{t\sqrt{gH}}{L}, \quad \eta' = \frac{\eta}{A}, \quad \phi' = \frac{H\phi}{AL\sqrt{gH}}, \quad h' = \frac{h}{H}, \tag{25.8}$$

together with the small parameters

$$\mu = \frac{H}{L}, \quad \varepsilon = \frac{A}{H}. \tag{25.9}$$

In the last equation, μ is called the long wave parameter and ε the small amplitude wave parameter. Note that ε is related with the nonlinear terms and μ with the dispersive terms. For simplicity, in what follows, we drop the prime notation.

The Boussinesq approach consists of reducing a 3D problem to a 2D one. This may be accomplished by expanding the velocity potential in a Taylor power series in terms of z. Using Laplace's equation, in a dimensionless form, we can obtain the following expression for the velocity potential:

$$\phi(x, y, z, t) = \sum_{n=0}^{+\infty} \left((-1)^n \frac{z^{2n}}{(2n)!} \mu^{2n} \nabla^{2n} \phi_0(x, y, t) + (-1)^n \frac{z^{2n+1}}{(2n+1)!} \mu^{2n} \nabla^{2n} \phi_1(x, y, t) \right), \tag{25.10}$$

with

$$\phi_0 = \phi \mid_{z=0}, \quad \phi_1 = \left(\frac{\partial \phi}{\partial z} \right) \mid_{z=0}. \tag{25.11}$$

From asymptotic expansions, successive approximation techniques and the kinematic boundary condition for the sea bottom, it is possible to write ϕ_1 in terms of ϕ_0 (see Chen and Liu (1994) and Zhao et al. (2004)). In this work, without loss of generality, we assume that the dispersive and nonlinear terms are related by the following equation:

$$\frac{\varepsilon}{\mu^2} = O(1) \text{ with } \mu < 1 \text{ and } \varepsilon < 1. \tag{25.12}$$

Note that the Ursell number is defined by $U_r = \varepsilon/\mu^2$ and plays a central role in deciding the choice of approximations which correspond to very different physics. The regime of weakly nonlinear, small amplitude and moderately long waves in shallow water is characterized by (25.12) $\left(O(\mu^2) = O(\varepsilon)\right.$; that is, $\left.\frac{H^2}{L^2} \sim \frac{A}{H}\right)$. Boussinesq equations account for the effects of nonlinearity ε and dispersion μ^2 to the leading order. When $\varepsilon \gg \mu^2$, they reduce to the Airy equations. When $\varepsilon \ll \mu^2$ they reduce to the linearized approximation with weak dispersion. Finally, if we assume that $\varepsilon \to 0$ and $\mu^2 \to 0$, the classical linearized wave equation is obtained.

A sixth-order spatial derivative model is obtained if ϕ_1 is expanded in terms of ϕ_0 and all terms up to $O(\mu^8)$ are retained. Thus, the asymptotic kinematic and dynamic boundary conditions for the

free water surface are rewritten as follows [1]:

$$\frac{\partial \eta}{\partial t} + \varepsilon \nabla \cdot (\eta \nabla \phi_0) - \frac{1}{\mu^2}\phi_1 + \frac{\varepsilon^2}{2}\nabla \cdot (\eta^2 \nabla \phi_1) = O(\mu^6), \tag{25.13a}$$

$$\frac{\partial \phi_0}{\partial t} + \varepsilon \eta \frac{\partial \phi_1}{\partial t} + \eta + \frac{\varepsilon}{2}|\nabla \phi_0|^2 + \varepsilon^2 \nabla \phi_0 \cdot \eta \nabla \phi_1 - \varepsilon^2 \eta \nabla^2 \phi_0 \phi_1 + \frac{\varepsilon}{2\mu^2}\phi_1^2 + D(\phi_0, \phi_1) = O(\mu^6), \tag{25.13b}$$

where ϕ_1 is given by:

$$\begin{aligned}\phi_1 = &-\mu^2 \nabla \cdot (h \nabla \phi_0) + \frac{\mu^4}{6}\nabla \cdot \left(h^3 \nabla^3 \phi_0\right) - \frac{\mu^4}{2}\nabla \cdot \left(h^2 \nabla^2 \cdot (h \nabla \phi_0)\right) \\ &- \frac{\mu^6}{120}\nabla \cdot \left(h^5 \nabla^5 \phi_0\right) + \frac{\mu^6}{24}\nabla \cdot \left(h^4 \nabla^4 \cdot (h \nabla \phi_0)\right) + \frac{\mu^6}{12}\nabla \cdot \left(h^2 \nabla^2 \cdot \left(h^3 \nabla^3 \phi_0\right)\right) \\ &- \frac{\mu^6}{4}\nabla \cdot \left(h^2 \nabla^2 \cdot \left(h^2 \nabla^2 \cdot (h \nabla \phi_0)\right)\right) - \frac{\mu^2}{\varepsilon}\frac{\partial h}{\partial t} - \frac{\mu^2}{\varepsilon}\frac{\mu^2}{2}\nabla \cdot \left(h^2 \nabla \frac{\partial - h}{\partial t}\right) \\ &+ \frac{\mu^2}{\varepsilon}\frac{\mu^4}{24}\nabla \cdot \left(h^4 \nabla^3 \frac{\partial h}{\partial t}\right) - \frac{\mu^2}{\varepsilon}\frac{\mu^4}{4}\nabla \cdot \left(h^2 \nabla^2 \left(h^2 \nabla \frac{\partial h}{\partial t}\right)\right) + O(\mu^8).\end{aligned} \tag{25.14}$$

To obtain equation (25.14), we assume that $\dfrac{\partial h}{\partial t} = O(\varepsilon)$ (see Dutykh and Dias (2007)).

25.3.2 Second-order spatial derivative model

The second-order spatial derivative equations are obtained, essentially, via the slowly varying bottom assumption. In particular, only $O(h, \nabla h)$ terms are retained. Also, only $O(\varepsilon)$ nonlinear terms are admitted. In fact, the extended ZTC model is written retaining only $O(\varepsilon, \mu^4)$ terms.

Under these conditions, (25.13) and (25.14) lead to:

$$\frac{\partial \eta}{\partial t} + \varepsilon \nabla \cdot (\eta \nabla \phi_0) - \frac{1}{\mu^2}\phi_1 = O(\mu^6), \tag{25.15a}$$

$$\frac{\partial \phi_0}{\partial t} + \eta + \frac{\varepsilon}{2}|\nabla \phi_0|^2 - \nu^* \varepsilon \nabla^2 \phi_0 = O(\mu^6), \tag{25.15b}$$

where $\nu^* = \nu\sqrt{H}/(AL\sqrt{g})$ and

$$\begin{aligned}\phi_1 = &-\mu^2 \nabla \cdot (h \nabla \phi_0) + \frac{\mu^4}{6}\nabla \cdot \left(h^3 \nabla^3 \phi_0\right) - \frac{\mu^4}{2}\nabla \cdot \left(h^2 \nabla^2 \cdot (h \nabla \phi_0)\right) \\ &- \frac{2\mu^6}{15}h^5 \nabla^6 \phi_0 - 2\mu^6 h^4 \nabla h \cdot \nabla^5 \phi_0 - \frac{\mu^2}{\varepsilon}\frac{\partial h}{\partial t} + O(\mu^8).\end{aligned} \tag{25.16}$$

Thus, these extended equations are written as follows:

$$\frac{\partial \eta}{\partial t} + \nabla \cdot [(h + \varepsilon\eta)\nabla \Phi] - \frac{\mu^2}{2}\nabla \cdot [h^2 \nabla \frac{\partial \eta}{\partial t}] + \frac{\mu^2}{6}h^2 \nabla^2 \frac{\partial \eta}{\partial t} - \frac{\mu^2}{15}\nabla \cdot [h \nabla (h \frac{\partial \eta}{\partial t})] = -\frac{1}{\varepsilon}\frac{\partial h}{\partial t}, \tag{25.17a}$$

$$\frac{\partial \Phi}{\partial t} + \frac{\varepsilon}{2}|\nabla \Phi|^2 + \eta - \frac{\mu^2}{15}h \nabla \cdot (h \nabla \eta) - \nu^* \varepsilon \nabla^2 \Phi = 0, \tag{25.17b}$$

[1] Note that D is, now, a dimensionless function.

where Φ is the transformed velocity potential given by:

$$\Phi = \phi_0 + \mu^2 \frac{h}{15} \nabla \cdot (h \nabla \phi_0). \tag{25.18}$$

In terms of the dimensional variables, equations (25.17) become:

$$\frac{\partial \eta}{\partial t} + \nabla \cdot [(h+\eta)\nabla\Phi] - \frac{1}{2}\nabla \cdot [h^2 \nabla \frac{\partial \eta}{\partial t}] + \frac{1}{6} h^2 \nabla^2 \frac{\partial \eta}{\partial t} - \frac{1}{15} \nabla \cdot [h \nabla (h \frac{\partial \eta}{\partial t})] = -\frac{\partial h}{\partial t}, \tag{25.19a}$$

$$\frac{\partial \Phi}{\partial t} + \frac{1}{2}|\nabla\Phi|^2 + g\eta - \frac{1}{15} g h \nabla \cdot (h \nabla \eta) - \nu \nabla^2 \Phi = 0, \tag{25.19b}$$

whereas equation (25.18) is rewritten as follows:

$$\Phi = \phi_0 + \frac{h}{15} \nabla \cdot (h \nabla \phi_0). \tag{25.20}$$

In this context, the use of the transformed velocity potential has two main advantages (see Zhao et al. (2004)):

1. the spatial derivative order is reduced to 2;

2. linear dispersion characteristics, analogous to the BEP model proposed by Chen and Liu (1994) and the BEV model developed by Nwogu (1993), are obtained. The latter models contain fourth and third-order spatial derivatives, respectively.

25.4 Linear dispersion relation

One of the most important properties of a water wave model is described by the linear dispersion relation. From this relation we can deduce the phase velocity, group velocity and the linear shoaling.

The dispersion relation of a linearized water wave model should be in good agreement with the one provided by the linear wave theory of Airy.

In this section, we follow the work by Dutykh and Dias (2007). Moreover, we only present a generalized version of the dispersion relation for the extended ZTC model with the dissipative term mentioned above. We can also include other damping terms, which are usually used in the sponge layers.

For simplicity, a one horizontal dimensional model is considered. To obtain the dispersion relation, a standard test wave is assumed:

$$\eta(x,t) = a\, e^{i(kx-\omega t)}, \tag{25.21a}$$

$$\Phi(x,t) = -b\, i\, e^{i(kx-\omega t)}, \tag{25.21b}$$

where a is the wave amplitude, b the potential magnitude, $k = 2\pi/L$ the wave number and ω the angular frequency. This wave, described by equations (25.21), is the solution of the linearized extended ZTC model, with a constant depth bottom and an extra dissipative term, if the following equation is satisfied:

$$\omega^2 - ghk^2 \frac{1 + \frac{1}{15}(kh)^2}{1 + \frac{2}{5}(kh)^2} + i\nu\omega k^2 = 0. \tag{25.22}$$

The dispersion relation given by the last equation is accurate up to $O((kh)^4)$ or $O(\mu^4)$ when compared

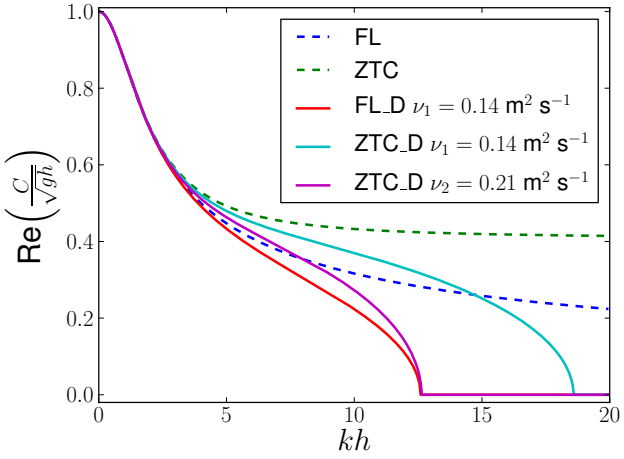


Figure 25.3: Positive part of $Re\left(C/\sqrt{gh}\right)$ as a function of kh for several models.

with the Padé approximant of order $[2/2]$ of the following equation:

$$\omega^2 - ghk^2\frac{\tanh(kh)}{kh} + i\nu\omega k^2 = 0. \tag{25.23}$$

In fact, equation (25.23) is the dispersion relation of the full linear problem.

From (25.22), the phase velocity, $C = w/k$, for this dissipative and extended ZTC model is given by:

$$C = -\frac{i\nu k}{2} \pm \sqrt{-\left(\frac{\nu k}{2}\right)^2 + gh\frac{(1+\frac{1}{15}(kh)^2)}{(1+\frac{2}{5}(kh)^2)}}. \tag{25.24}$$

In Figure 25.3, we can see the positive real part of $\left(C/\sqrt{gh}\right)$ as a function of kh for the following models: full linear theory (FL), Zhao et al. (ZTC), full linear theory with a dissipative model (FL_D) and the improved ZTC model with the dissipative term (ZTC_D).

From Figure 25.3, we can also see that these two dissipative models admit critical wave numbers k_1 and k_2, such that the positive part of $Re\left(C/\sqrt{gh}\right)$ is zero for $k \geqslant k_1$ and $k \geqslant k_2$, respectively. We can optimize the value of ν in the ZTC_D model in order to have $k_1 = k_2$. From (25.23), $Re\left(C/\sqrt{gh}\right)$ is zero for

$$k_1^3 = 4g\frac{\tanh(k_1 h)}{\nu^2}. \tag{25.25}$$

Thus, we can obtain the values of k_1, in the FL_D model, for which short waves no longer propagate for fixed h and $\nu = \nu_1$ values. Considering now the real part of (25.24) equal to zero, we have

$$\nu^2 = 4\frac{gh}{k^2}\left(\frac{1+\frac{1}{15}(kh)^2}{1+\frac{2}{5}(kh)^2}\right). \tag{25.26}$$

Therefore, inserting the previous value of k_1 into (25.26) we obtain the corresponding value of $\nu = \nu_2$, in the ZTC_D model, for which the same type of waves do not propagate. As in Dutykh and Dias (2007) we choose $\nu_1 = 0.14\,\mathrm{m}^2\,\mathrm{s}^{-1}$. In Figure 25.3, we can see that if $\nu_1 = 0.14\ \mathrm{m}^2\,\mathrm{s}^{-1}$ in the FL_D

model and $\nu_2 = 0.21\ \mathtt{m}^2\,\mathtt{s}^{-1}$ in the ZTC_D model, $k_1 = k_2 = 12.6\,\mathtt{m}^{-1}$ for $h = 1\,\mathtt{m}$. In this case the time decay of the solutions in the ZTC_D model is more accentuated than in the FL_D model. Some instabilities generated by short length waves can be eliminated optimizing the viscosity values as shown above.

In general, to improve the dispersion relation we can also use other transformations like (25.20), or evaluate the velocity potential at $z = -\sigma h$ ($\sigma \in [0,1]$) instead of $z = 0$ (see Bingham et al. (2008), Madsen and Agnon (2003) and Madsen et al. (2003)).

25.5 Numerical methods

We start this section by noting that a detailed description of the implemented numerical methods referred bellow can be found in the work of Lopes (2007).

For simplicity, we only consider the system described by equations (25.19) restricted to a stationary bottom and without dissipative or extra source terms.

The model variational formulation is written as follows:

$$\begin{aligned}
&\int_\Omega \frac{\partial \eta}{\partial t}\vartheta_1 \,\mathrm{d}x\,\mathrm{d}y + \frac{1}{2}\int_\Omega h^2 \nabla\left(\frac{\partial \eta}{\partial t}\right)\cdot\nabla\vartheta_1 \,\mathrm{d}x\,\mathrm{d}y - \frac{1}{6}\int_\Omega \nabla\left(\frac{\partial \eta}{\partial t}\right)\cdot\nabla(h^2\vartheta_1)\,\mathrm{d}x\,\mathrm{d}y \\
&\quad + \frac{1}{15}\int_\Omega h\nabla\left(h\frac{\partial \eta}{\partial t}\right)\cdot\nabla\vartheta_1\,\mathrm{d}x\,\mathrm{d}y - \frac{1}{15}\int_\Gamma h\frac{\partial h}{\partial n}\frac{\partial \eta}{\partial t}\vartheta_1\,\mathrm{d}s \qquad (25.27a)\\
&\quad = \int_\Omega (h+\eta)\nabla\Phi\cdot\nabla\vartheta_1\,\mathrm{d}x\,\mathrm{d}y - \int_\Gamma (h+\eta)\frac{\partial\Phi}{\partial n}\vartheta_1\,\mathrm{d}s + \frac{2}{5}\int_\Gamma h^2\frac{\partial}{\partial t}\left(\frac{\partial \eta}{\partial n}\right)\vartheta_1\,\mathrm{d}s,\\
&\int_\Omega \frac{\partial\Phi}{\partial t}\vartheta_2\,\mathrm{d}x\,\mathrm{d}y = -\frac{1}{2}\int_\Omega |\nabla\Phi|^2\vartheta_2\,\mathrm{d}x\,\mathrm{d}y - g\int_\Omega \eta\,\vartheta_2\,\mathrm{d}x\,\mathrm{d}y \\
&\qquad\qquad - \frac{g}{15}\int_\Omega h\nabla\eta\cdot\nabla(h\vartheta_2)\,\mathrm{d}x\,\mathrm{d}y + \frac{g}{15}\int_\Gamma h^2\frac{\partial\eta}{\partial n}\vartheta_2\,\mathrm{d}s, \qquad (25.27b)
\end{aligned}$$

where the unknown functions η and Φ are the wave surface elevation and the transformed velocity potential, whereas ϑ_1 and ϑ_2 are the test functions defined in appropriate spaces.

We use a predictor-corrector scheme with an initialization provided by an explicit Runge–Kutta method for the time integration. In the DOLFWAVE code these routines are implemented in the PredCorr and RungeKutta classes (see `dolfwave/src/predictorcorrector` and `dolfwave/src/rungekutta`).

Note that the discretization of equations (25.27) can be written in the following form:

$$M\dot{U} = F(t,U), \qquad (25.28)$$

where $\dot{U}$ and U refer to $\left(\frac{\partial\eta}{\partial t}, \frac{\partial\Phi}{\partial t}\right)$ and (η,Φ), respectively. The coefficient matrix M is given by the left-hand sides of (25.27), whereas the known vector F is related with the right-hand sides of the same equations. In this way, the fourth-order Adams-Bashforth-Moulton method can be written as follows:

$$MU_{n+1}^{(0)} = MU_n + \frac{\Delta t}{24}[55F(t_n,U_n) - 59F(t_{n-1},U_{n-1}) + 37F(t_{n-2},U_{n-2}) - 9F(t_{n-3},U_{n-3})], \qquad (25.29a)$$

$$MU_{n+1}^{(1)} = MU_n + \frac{\Delta t}{24}[9F(t_{n+1},U_{n+1}^{(0)}) + 19F(t_n,U_n) - 5F(t_{n-1},U_{n-1}) + F(t_{n-2},U_{n-2})], \qquad (25.29b)$$

where Δt is the time step, $t_n = n\Delta t$ ($n \in \mathbb{N}$) and U_n is U evaluated at t_n. The predicted and corrected values of U_n are denoted by $U_n^{(0)}$ and $U_n^{(1)}$, respectively. The corrector-step equation (25.29b) can be iterated as function of a predefined error between consecutive time steps. For more details see, e.g., Hairer and Wanner (1991a) or Lambert (1991).

The UFL form file (see `dolfwave/src/2hdforms/Zhao.ufl`) for the declaration of the spatial discretization of (25.27) using Lagrange P_1 elements (see Chapter 3) and including dissipative and source terms is presented below.

Python code

```
P = FiniteElement("Lagrange",triangle,1) # Linear Lagrange element in triangles
Th = P*P # Product space for basis functions

# eta_t: time derivative of the surface elevation
# phi_t: time derivative of the velocity potential
(eta_t,phi_t) = TrialFunctions(Th)

# p: test function for eta_t
# q: test function for phi_t
(p, q) = TestFunctions(Th)

eta = Coefficient(P) # Surface elevation
phi = Coefficient(P) # Velocity potential

h = Coefficient(P) # Depth function
g = Constant(triangle) # Gravity acceleration

# Several types of sponge layers are considered
sp_eta = Coefficient(P) # Viscous frequency coefficient of eta
sp_lap_eta = Coefficient(P) # Viscosity coefficient of Laplacian of eta
sp_phi = Coefficient(P) # Viscous frequency coefficient of phi
sp_lap_phi = Coefficient(P) # Viscosity coefficient of Laplacian of phi

# Source function for the surface elevation equation
srceta = Coefficient(P)

# Normal Vector for boundary contributions
n = P.cell().n

# Bilinear form declaration for M
# Contribution from the surface elevation equation
a0 = eta_t*p*dx
a1 = (1./2.)*inner(h*h*grad(eta_t),grad(p))*dx
a2 = -(1./6.0)*inner(grad(eta_t),grad(h*h*p))*dx
a3 = (1./15.0)*inner(h*grad(h*eta_t),grad(p))*dx
a4 = -(1./15.)*h*inner(grad(h),n)*eta_t*p*ds # Boundary contribution

# Contribution from the velocity potential equation
a5 = (phi_t*q)*dx

# a: bilinear form
# See "dolfwave/src/formsfactory/bilinearforminit.cpp"
a = a0+a1+a2+a3+a4+a5

# Linear Variational form declaration for F(t,U)
# Contribution from the surface elevation equation
l0 = inner(((h+eta))*grad(phi0),grad(p))*dx

# Contribution from the velocity potential equation
l1 = -(1./2.)*inner(grad(phi0),grad(phi0))*q*dx
l2 = -g*(eta*q)*dx
l3 = -g*(1.0/15.0)*inner(h*grad(eta),grad(h*q))*dx

# Sponge layers contributions
l4 = -sp_eta*eta*p*dx-sp_lap_eta*inner(grad(eta),grad(p))*dx
```

```
l5 = -sp_phi*phi0*q*dx-sp_lap_phi*inner(grad(phi0),grad(q))*dx

# Source function for the surface elevation equation
l6=srceta*p*dx

# L: linear form
# See "dolfwave/src/formsfactory/linearforminit.cpp"
L = l0+l1+l2+l3+l4+l5+l6
```

Some wave generation mechanisms as well as reflective walls and sponge layers are discussed in sections 25.6 and 25.7, respectively.

25.6 Wave generation

In this section, some of the physical mechanisms responsible for inducing surface water waves are presented. We note that the moving bottom approach is useful for wave generation due to seismic activities. However, some physical applications are associated with other wave generation mechanisms. For simplicity, we only consider mechanisms to generate surface water waves along the x direction.

25.6.1 Initial conditions

The simplest way of inducing a wave into a certain domain is to consider an appropriate initial condition. A useful and typical case is to assume a solitary wave given by:

$$\eta(x,t) = a_1 \operatorname{sech}^2(kx-\omega t) + a_2 \operatorname{sech}^4(kx-\omega t) \quad \text{at } t = 0\,\mathrm{s}, \tag{25.30}$$

$$u(x,t) = a_3 \operatorname{sech}^2(kx-\omega t) \quad \text{at } t = 0\,\mathrm{s}, \tag{25.31}$$

where the parameters a_1 and a_2 are the wave amplitudes and a_3 is the magnitude of the velocity in the x direction. Since we use a potential formulation, Φ is given by:

$$\Phi(x,t) = -\frac{2a_3\, e^{2\omega t}}{k\,(e^{2\omega t}+e^{2kx})} + K_1(t) \quad \text{at } t = 0\,\mathrm{s}, \tag{25.32}$$

where $K_1(t)$ is a generic time dependent function of integration. In fact, in order to satisfy the solution of equation (25.19b) $K_1(t)$ is specified as a constant.

We remark that the above solitary wave given by (25.30) and (25.31), but for all time t, was presented as a solution of the extended Nwogu's Boussinesq model in Walkley (1999) and Wei and Kirby (1995).

25.6.2 Incident wave

For time dependent wave generation, it is possible to consider waves induced by a boundary condition. This requires that the wave surface elevation and the velocity potential must satisfy appropriate boundary conditions, e.g., Dirichlet or Neumann conditions.

The simplest case is to consider a periodic wave given by:

$$\eta(x,t) = a\sin(kx-\omega t) \tag{25.33}$$

$$\Phi(x,t) = -\frac{c}{k}\cos(kx-\omega t) + K_2(t), \tag{25.34}$$

where c is the wave velocity magnitude and $K_2(t)$ is a time dependent function of integration. This function $K_2(t)$ must satisfy the initial condition of the problem. Note that the parameters a, c, k and ω are not arbitrary. Specifically, k and ω should be related by the dispersion equation (25.22) (with no dissipative effects) while c is given by the following expression:

$$\frac{c}{k} = \frac{a\omega}{hk^2}\left(1 + \frac{2}{5}(kh)^2\right) \tag{25.35}$$

We can also consider the superposition of water waves as solutions of the full linear problem with a constant depth.

25.6.3 Source function

In the work by Wei et al. (1999), a source function for the generation of surface water waves was derived. This source function was obtained, using Fourier transform and Green's functions, to solve the linearized and nonhomogeneous equations of the Peregrine (1967) and Nwogu (1993) models. This mathematical procedure can also be adapted here to deduce the source function.

We consider a monochromatic Gaussian wave generated by the following source function:

$$S(x,t) = D^* \exp(-\beta(x - x_s)^2)\cos(\omega t), \tag{25.36}$$

with D^* given by:

$$D^* = \frac{\sqrt{\beta}}{\omega\sqrt{\pi}} a \exp(\frac{k^2}{4\beta})\frac{2}{15}h^3k^3g. \tag{25.37}$$

In the above expressions x_s is the center line of the source function and β is a parameter associated with the width of the generation band (see Wei et al. (1999)). Note that $S(x,t)$ should be added to the right-hand side of equation (25.19a). A DOLFWAVE demo code for an example of wave generation using this source function is available at `dolfwave/demo/2HD/srcFunction`.

25.7 Reflective walls and sponge layers

Besides the incident wave boundaries where the wave profiles are given, we must close the system with appropriate boundary conditions. We consider two more types of boundaries:

1. full reflective boundaries;
2. sponge layers.

The first case is modelled by the following equations:

$$\frac{\partial\Phi}{\partial n} = 0 \quad \text{and} \quad \frac{\partial\eta}{\partial n} = 0 \quad \text{on } \Gamma, \tag{25.38}$$

where n is the outward unit vector normal to the boundary Γ of the domain Ω.

Regarding the second case, we consider equations (25.38) and an extra artificial term, often called sponge or damping layer, given by $\nu\nabla^2\Phi$ (see equation (25.19b)), acting in a neighborhood of the boundary Γ. In this way, the reflected energy can be controlled. Moreover, we can prevent unwanted wave reflections and avoid complex wave interactions. It is also possible to simulate effects like energy dissipation by wave breaking.

In fact, a sponge layer is a subset Ω_S of Ω where some extra viscosity term is added. As mentioned above, the system of equations can incorporate several extra damping terms, like that one provided by

the inclusion of a dissipative model. Thus, the viscosity coefficient ν can be described by a function of the following form:

$$\nu(x,y) = \begin{cases} 0, & (x,y) \notin \Omega_S, \\ n_1 \dfrac{\exp\left(\dfrac{d_{\Omega_S} - d(x,y)}{d_{\Omega_S}}\right)^{n_2} - 1}{\exp(1) - 1}, & (x,y) \in \Omega_S, \end{cases} \tag{25.39}$$

where n_1 and n_2 are, in general, experimental parameters, d_{Ω_S} is the sponge-layer diameter and $d(x,y)$ stands for a distance function between a point (x,y) and the intersection of Γ with the boundary of Ω_S (see, e.g., Walkley (1999)).

25.8 Linear stability analysis

In this section, we use a matrix-based analysis in order to study some stability properties of the linearized ZTC model in one horizontal dimension and with a time-independent bathymetry. We follow the procedures outlined in Løvholt and Pedersen (2009) applied to the finite element discretization associated to the spatial variable. Only uniform meshes are considered in this stability analysis. The standard potential model with depth averaged velocity potential investigated by Løvholt and Pedersen (2009) is also used here as a reference for comparison. For both models, full reflective boundary conditions are considered.

We start by assuming a separated solution of the form

$$\eta(x,t) = e^{i\omega t}\hat{\eta}(x), \quad \Phi(x,t) = e^{i\omega t}\hat{\Phi}(x), \tag{25.40}$$

where ω denotes the angular frequency which may be real or complex. In the linearized ZTC equations this separation will simply result in the substitution of $\frac{\partial \eta}{\partial t}$ by $i\omega\hat{\eta}$ and $\frac{\partial \Phi}{\partial t}$ by $i\omega\hat{\Phi}$. For the spatially discretized and linearized ZTC equations we replace ω by $\hat{\omega} = \frac{2}{\Delta t}\sin\left(\frac{\omega \Delta t}{2}\right)$ where Δt is the time-step (see Løvholt and Pedersen (2009)). Replacing η and Φ in (25.27) by their finite element approximations we obtain an eigenvalue problem of the form

$$(K - i\hat{\omega}M)U = 0, \tag{25.41}$$

where K is the stiffness matrix related to the right-hand sides of equations (25.27) and M is the mass matrix given by the left-hand sides of the same equations. This problem is solved using the DOLFIN interface for the SLEPc libraries. The DOLFWAVE demo code for this eigenvalue problem is available at `dolfwave/demo/stability`.

We remark that in a constant depth bathymetry (25.41) takes the simplified form:

$$\frac{H}{\Delta x}\hat{\Phi}_1 - \frac{H}{\Delta x}\hat{\Phi}_2 - i\hat{\omega}\left[\left(\frac{\Delta x}{3} + \frac{2}{5}\frac{H^2}{\Delta x^2}\right)\hat{\eta}_1 + \left(\frac{\Delta x}{6} - \frac{2}{5}\frac{H^2}{\Delta x^2}\right)\hat{\eta}_2\right] = 0, \tag{25.42a}$$

$$\left(-g\frac{\Delta x}{3} - g\frac{H^2}{15\Delta x}\right)\hat{\eta}_1 + \left(-g\frac{\Delta x}{6} + g\frac{H^2}{15\Delta x}\right)\hat{\eta}_2 - i\hat{\omega}\left(\frac{\Delta x}{3}\hat{\Phi}_1 + \frac{\Delta x}{6}\hat{\Phi}_2\right) = 0, \tag{25.42b}$$

$$\begin{aligned} &-\frac{H}{\Delta x}\left(\hat{\Phi}_{j-1} + \hat{\Phi}_{j+1}\right) + \frac{2H}{\Delta x}\hat{\Phi}_j \\ &\quad - i\hat{\omega}\left[2\left(\frac{\Delta x}{3} + \frac{2}{5}\frac{H^2}{\Delta x^2}\right)\hat{\eta}_i + \left(\frac{\Delta x}{6} - \frac{2}{5}\frac{H^2}{\Delta x^2}\right)\left(\hat{\eta}_{j-1} + \hat{\eta}_{j+1}\right)\right] = 0, \quad (2 \leqslant j \leqslant n-2) \end{aligned} \tag{25.42c}$$

$$\begin{aligned} &-2\left(g\frac{\Delta x}{3} + g\frac{H^2}{15\Delta x}\right)\hat{\eta}_j + \left(-g\frac{\Delta x}{6} + g\frac{H^2}{15\Delta x}\right)\left(\hat{\eta}_{j-1} + \hat{\eta}_{j+1}\right) \\ &\quad - i\hat{\omega}\left[2\frac{\Delta x}{3}\hat{\Phi}_j + \frac{\Delta x}{6}\left(\hat{\Phi}_{j-1} + \hat{\Phi}_{j+1}\right)\right] = 0, \quad (2 \leqslant j \leqslant n-2) \end{aligned} \tag{25.42d}$$

$$\frac{H}{\Delta x}\hat{\Phi}_n - \frac{H}{\Delta x}\hat{\Phi}_{n-1} - i\hat{\omega}\left[\left(\frac{\Delta x}{3} + \frac{2}{5}\frac{H^2}{\Delta x^2}\right)\hat{\eta}_n + \left(\frac{\Delta x}{6} - \frac{2}{5}\frac{H^2}{\Delta x^2}\right)\hat{\eta}_{n-1}\right] = 0, \tag{25.42e}$$

$$\left(-g\frac{\Delta x}{3} - g\frac{H^2}{15\Delta x}\right)\hat{\eta}_n + \left(-g\frac{\Delta x}{6} + g\frac{H^2}{15\Delta x}\right)\hat{\eta}_{n-1} - i\hat{\omega}\left(\frac{\Delta x}{3}\hat{\Phi}_n + \frac{\Delta x}{6}\hat{\Phi}_{n-1}\right) = 0, \tag{25.42f}$$

where $\Delta x = x_{j+1} - x_j$ ($j = 1, \ldots, n-1$) is the uniform mesh size and $\hat{\eta}_j$ as well as $\hat{\Phi}_j$ stand for $\hat{\eta}(x_j)$ and $\hat{\Phi}(x_j)$ ($j = 1, \ldots, n$), respectively. We can show that

$$\hat{\omega} = \pm\sqrt{g\left(\frac{3H}{\Delta x^2}\right)\left(\frac{5\Delta x^2 + H^2}{5\Delta x^2 + 6H^2}\right)} \tag{25.43}$$

are always eigenvalues of the constant depth problem. This allows us to conclude that, for this case, the accuracy of the eigenvalue solver is 10^{-11} and that the spectral radius goes to infinity as the mesh size approaches zero.

As mentioned in Løvholt and Pedersen (2009), instabilities associated with steep bottoms may occur for some BEV and BEP models. For instance, it was shown that the standard potential model used here for comparison is very prone to such instabilities. From equations (25.40) and (25.41), unstable wave modes may appear when eigenvalues $\hat{\omega}$ are of the following types:

1. when $\hat{\omega}$ is a pure imaginary number the solutions grow or decay exponentially without propagation;

2. when $\hat{\omega}$ is a complex number the solutions grow or decay exponentially and propagate;

3. when a real solution is found for $\hat{\omega}$ but yielding $\frac{1}{2}\Delta t|\hat{\omega}| > 1$ and ω complex. This corresponds to a CFL criterion but this kind of instability may anyhow be avoided for a sufficiently small Δt.

For the stability tests, we consider the geometries in Figure 25.4 with $l = 2$ m, $l = 1$ m and $l = 0.5$ m. In all the cases we test 1300 pairs of $(h_m, \Delta x)$ with h_m and $\Delta x \in]0, 1]$ (m). In Figs. 25.5–25.7, we can see the unstable wave modes $(h_m, \Delta x)$ of the ZTC (red circles) and standard potential (blue points) models for the spike (left panels) and shelf (right panels) geometries with $l = 2$ m, $l = 1$ m and $l = 0.5$ m. We only present $(h_m, \Delta x)$ related to the eigenvalues with imaginary part (exponential growth/decay rate) at least of order 10^{-5} s^{-1}. We remark that in the ZTC model, we observe at most 10 unstable wave modes of type I with growth rates smaller than $|Im(\hat{\omega})| = 3 \times 10^{-5}$ s^{-1}.

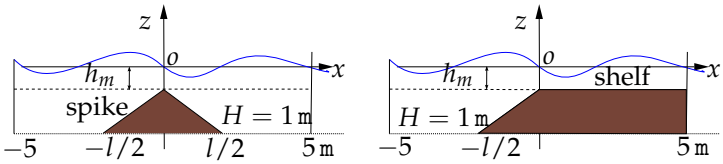


Figure 25.4: Spike and shelf geometries for the impermeable sea bottom.

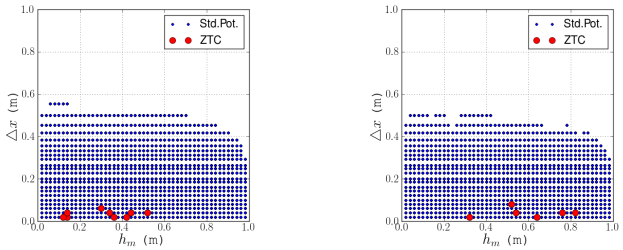


Figure 25.5: Unstable wave modes of the ZTC (red circles) and standard potential (blue points) models for the spike (left panel) and shelf (right panel) geometries with $l = 2$ m.

In Figs. 25.8–25.10, we present the eigenvalues for the standard potential (upper panels) and ZTC (lower panels) models, for the spike (left panels) and shelf (right panels) geometries, with $l = 2$ m and $\Delta x = 0.02$ m, $l = 1$ m and $\Delta x = 0.1$ m as well as $l = 0.5$ m and $\Delta x = 0.25$ m. The spectrum depends on h_m and the eigenvalues are plotted with different colors (from red to blue) to accentuate that dependence ($h_m \approx 1$ m, $h_m \approx 0.5$ m and $h_m \approx 0.02$ m denoted by red, green and blue circles, respectively).

As in Løvholt and Pedersen (2009), we find instabilities of type I and II for the standard potential model, specially for steep bottom gradients and finer meshes. We also observe that as l increases so does the number of unstable wave modes. Moreover, a steep gradient increases the growth rate of the unstable solutions for the standard potential model. In contrast, for the ZTC model all the growth rates are limited to $|Im(\hat{\omega})| = 3 \times 10^{-5}$ s^{-1}. It was shown in Løvholt and Pedersen (2009) that a lower bound growth rate of $|Im(\hat{\omega})| = 10^{-5}$ s^{-1} does not influence the numerical results in most real problems, even when steep bottom gradients occur as in tsunami simulations.

From Figures 25.5–25.10, we can conclude that the ZTC model is very robust in terms of the instabilities depending on l, depth gradients and mesh discretizations.

In the next section, we also test the weakly nonlinear ZTC model in order to verify its robustness with respect to instabilities.

25.9 Model validation and numerical applications

To validate the model we consider two benchmark tests. Additionally, the wave propagation in a harbor as well as the generation of a wave due to a time dependent moving bottom are also investigated.

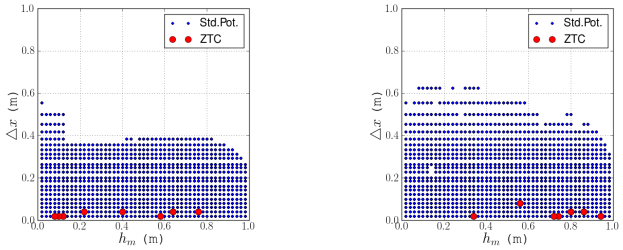


Figure 25.6: Unstable wave modes of the ZTC (red circles) and standard potential (blue points) models for the spike (left panel) and shelf (right panel) geometries with $l = 1$ m.

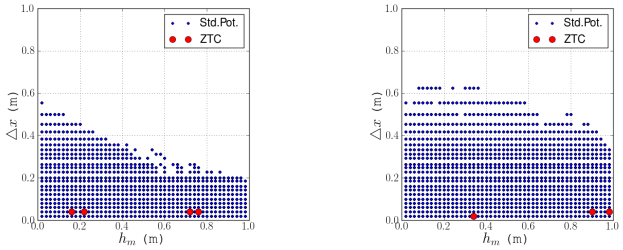


Figure 25.7: Unstable wave modes of the ZTC (red circles) and standard potential (blue points) models for the spike (left panel) and shelf (right panel) geometries with $l = 0.5$ m.

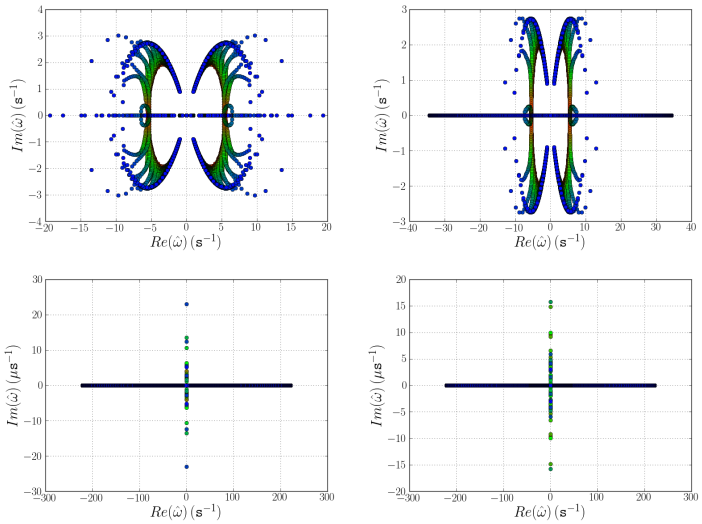


Figure 25.8: Eigenvalue spectrum for the standard potential (upper panels) and ZTC (lower panels) models for the spike (left panels) and shelf (right panels) geometries, with $l = 2$ m, $\Delta x = 0.02$ m and h_m from 1 m (red circles) to 0.02 m (blue circles).

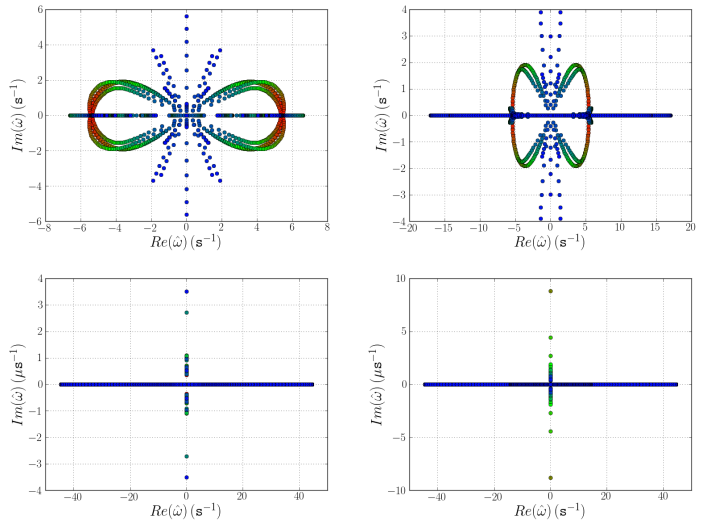


Figure 25.9: Eigenvalue spectrum for the standard potential (upper panels) and ZTC (lower panels) models for the spike (left panels) and shelf (right panels) geometries, with $l = 1$ m, $\Delta x = 0.1$ m and h_m from 1 m (red circles) to 0.02 m (blue circles).

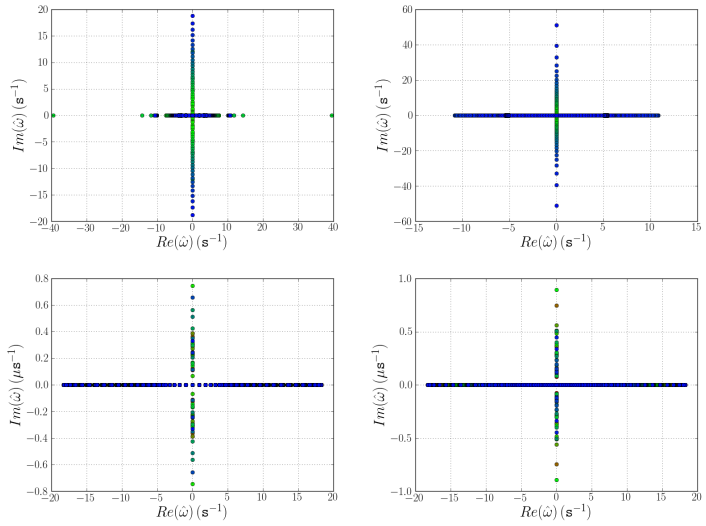


Figure 25.10: Eigenvalue spectrum for the standard potential (upper panels) and ZTC (lower panels) models for the spike (left panels) and shelf (right panels) geometries, with $l = 0.5$ m, $\Delta x = 0.25$ m and h_m from 1 m (red circles) to 0.02 m (blue circles).

25.9.1 *Solitary wave over submerged bars*

In this subsection, we simulate the propagation of a solitary wave passing through trapezoidal and triangular submerged bars. Moreover, the solutions obtained by the DOLFWAVE (ZTC/BEP) solver are compared with those provided by a FEniCS independent numerical code to solve the Nwogu's BEV equations (see `dolfwave/tools/Nwogu`). Specifically, a finite element discretization of these Nwogu's equations is considered (see Walkley (1999)) together with an implicit Radau IIA-type Runge–Kutta scheme for the time integration (see Hairer and Wanner (1991b)).

We start now the description of the problem along with the DOLFWAVE code used to solve it. This C++ code should start with the inclusion of the DOLFWAVE library.

C++ code

```
#include <dolfwave.h>
using namespace dolfin::dolfwave;
```

The initial condition for the wave surface elevation is given by (25.30) and implemented as follows:

C++ code

```
class ElevationInit : public Expression
{
  void eval(Array<double> & values,const Array<double> & x) const
  {
    // Wave parameters (see Walkley [1999])
    double c=sqrt(1.025), H=0.4;
    double ca=-0.4, cb=ca+1.0/3.0;
    double center=-5.0;
    double a1=(H/3.0)*(sqr(c)-1)/(cb-ca*sqr(c));
    double a2=-(H/2.0)*sqr((sqr(c)-1)/c)*(cb+2.0*ca*sqr(c))/(cb-ca*sqr(c));
    double k=(1.0/(2.0*H))*sqrt((sqr(c)-1)/(cb-ca*sqr(c)));
    values[0]=a1/sqr(cosh(k*(x[0]-center)))+a2/sqr(sqr(cosh(k*(x[0]-center))));
  }
};
```

Moreover, the initial condition for the velocity potential is defined by (25.32) and implemented using the following code:

C++ code

```
class PotentialInit :  public Expression
{
  void eval(Array<double> & values,const Array<double> & x) const
  { //Wave parameters
    double c=sqrt(1.025), H=0.4;
    double ca=-0.4, cb=ca+1.0/3.0;
    double center=-5.0;
    double a3=sqrt(H*g_e)*(sqr(c)-1)/c;
    double k=(1.0/(2.0*H))*sqrt((sqr(c)-1)/(cb-ca*sqr(c)));
    double cnst=4.0*a3/(2.0*k*(1+exp(2.0*k*(-25.)))); //Constant of integration
    values[0]=-4.0*a3/(2.0*k*(1+exp(2.0*k*(x[0]-center))))+cnst;
  }
};
```

Te trapezoidal sea bottom h (m) is described by the following continuous and piecewise differentiable function:

$$h(x) = \begin{cases} 0.4 & \text{if} \quad -25 \leqslant x \leqslant 6 \\ -0.05x + 0.7 & \text{if} \quad 6 < x \leqslant 12 \\ 0.1 & \text{if} \quad 12 < x \leqslant 14 \\ 0.1x - 1.3 & \text{if} \quad 14 < x \leqslant 17 \\ 0.4 & \text{if} \quad 17 < x \leqslant 25 \end{cases} \quad (\mathrm{m}) \tag{25.44}$$

which is implemented by:

C++ code

```
class Depth :  public Expression
{
  void eval(Array<double> & values,const Array<double> & x) const
  {
    double retrn = 0.0;
    if (x[0] <= 6.0)
      retrn=0.4;
    else if (x[0] <= 12.0)
      retrn=-0.05*x[0]+0.7;
    else if (x[0] <= 14.0)
      retrn=0.1;
    else if(x[0] <= 17.0)
      retrn = 0.1*x[0]-1.3;
    else
      retrn = 0.4;
    values[0] = retrn;
  }
};
```

The main code starts with the creation of an object of the Dolfwave class by calling its constructor. Here, we simulate the wave propagation during 25 s using a time step of 0.001 s. The UFL form file used in this problem, which is identified by "Zhao_1D", corresponds to the one horizontal dimensional version of (25.27). We use a LU solver provided by the PETSc algebra backend (see Chapter 1). Here we use Viper for previewing the numerical solutions, which are saved in the "output" directory using a simple ASCII format denoted by "xyz".

C++ code

```
int main( )
{
  Dolfwave dw(25000 /*Number of steps*/,
```

```
            0.001 /*Time step*/,
            100 /*Gap for saving the solutions*/,
            "Zhao_1D" /*Variational form identifier*/,
            "LU_P" /*Linear solver type*/,
            "viper" /*Preview program*/,
            "output" /*Output directory*/,
            "xyz" /*File output format*/);
```

The spatial domain used in the case of the trapezoidal submerged bar is the interval $[-25, 25]$ (m) which is discretized using 201 nodes.

C++ code

```
Interval mesh(201,-25,25);
```

Now all the known functions are initialized. Sponge layers or source functions are not used here.

C++ code

```
Depth depth; // Depth function
ElevationInit eta_init; // Initial condition for the surface elevation
PotentialInit phi_init;  // Initial condition for the velocity potential
Constant zero(0.0); // Sponge layers and source function are 0
```

From the bilinear and linear forms of the variational formulation, all the finite element matrices and vectors are created.

C++ code

```
dw.SpacesFunctionsVectorsInit(mesh); // Initialization of the spaces, functions and vectors
dw.BilinearFormInit(mesh,depth); // Initialization of the bilinear form "a"
dw.MatricesAssemble( ); // Initialization of the system matrices
dw.LinearFormsInit(depth, zero, zero, zero, zero, zero); // Initialization of the linear form "L"
dw.InitialCondition(eta_init,phi_init); // Setting the initial conditions
```

We only need to make an initial factorization for the LU solver, since the system matrix does not depend on time.

C++ code

```
dw.LUFactorization(true); // Reuse the LU factorization throughout the time integration routines
```

The output of the symmetric of the depth function is given by:

C++ code

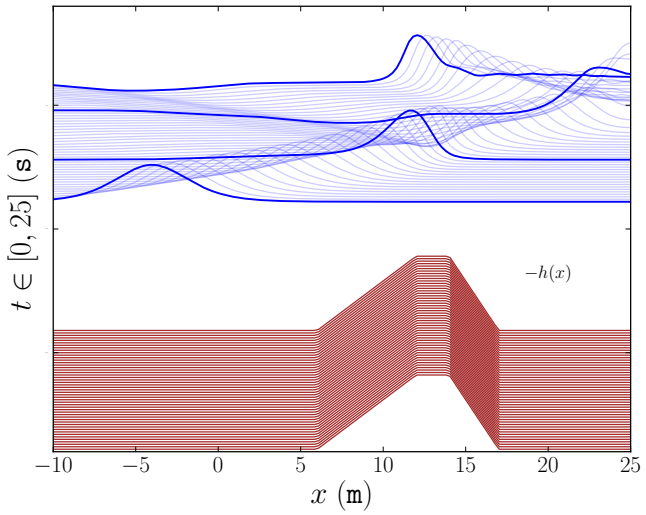
```
dw.DepthPlot(mesh,depth,true); // Plot the symmetric of the depth function h
```

Now, the time integration routines are used. The Adams–Bashforth–Moulton method described by equations (25.29) is initialized by a fourth-order explicit Runge–Kutta scheme.

C++ code

```
dw.RKInit("exp4");    // Choose the explicit 4th-order Runge-Kutta for initialization
dw.RKSolve( );        // Use the Runge-Kutta for the 3 initial steps
dw.PCInit(mesh,true); // Initialization of the predictor-corrector with multi-step corrector

// Advance in time with the predictor-corrector scheme
for(dolfin::uint i=4; i<dw.MaxSteps+1;i++)
{
  dw.PCSolve( ); // Adams-Bashforth-Moulton method
  if (!(i%dw.WriteGap)) // Save and preview the surf. elevation with a gap of 100 iterations
```

Figure 25.11: Detailed view of a wave passing through a trapezoidal submerged bar using the ZTC/BEP model implemented in DOLFWAVE




```
  dw.Plot(mesh, true /*eta preview*/, false /*phi preview*/,
          true /*eta save*/, false /*phi save*/);
}
```

In this case only the wave surface elevation is saved and previewed using the Plot function. The solution provided by this code, for the trapezoidal submerged bar with $x \in [-10, 25]$ (m), is given in Figure 25.11. We remark that the shoaling effect over the trapezoidal submerged bar is clearly observed, both for the incident and reflected waves.

From Figure 25.12, we can compare the solutions provided by the two independent models. We remark that (25.30) and (25.31) are used for the correspondent initial conditions of Nwogu's equations. In spite of the fact that the solutions come from different models and discretizations, a good agreement is achieved. These solutions also compare well with those provided by another model in the DOLFWAVE application (see also `dolfwave/demo/1HD/submergedbar`).

In the numerical simulation of a solitary wave passing through the triangular submerged bar, we investigate the nonlinear effects and the influence of h_m (see Figure 25.13) for the weakly nonlinear ZTC/BEP and Nwogu's BEV models. From Figure 25.14 we can conclude that these two models compare well in the case of the triangular submerged bar with $h_m = 0.1$ m. Even though both models compare well for the triangular submerged bar with a smaller value of $h_m = 0.04$ m, the Nwogu's BEV model presents small amplitude and high frequency oscillations after the interaction with the bar (see Figure 25.15). As the value of h_m is decreased the Nwogu's model becomes unstable (see Figure 25.16). In Figure 25.16 the blowup of the solution provided by the Nwogu's model is observed for $h_m = 0.02$ m. We remark that the reference values of ε at the point $(15, -h_m)$ (m) are $\varepsilon = 0.1$, $\varepsilon = 0.25$ and $\varepsilon = 0.5$ for $h_m = 0.1$ m, $h_m = 0.04$ m and $h_m = 0.02$ m, respectively. Although, $\varepsilon = 0.5$ is clearly out of the range of validity of both ZTC/BEP and Nwogu's BEV models, the first one seems more robust when dealing with the nonlinear effects. These stability properties of the ZTC/BEP model are also observed in numerical tests involving grid refinements.

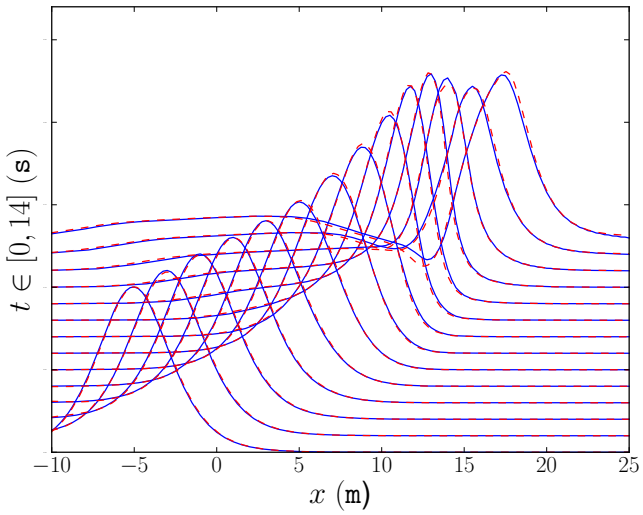


Figure 25.12: Detailed comparison of a wave passing through the trapezoidal submerged bar, simulated by DOLFWAVE using the ZTC/BEP (red dashed line) and Nwogu's BEV (blue solid line) models, for $x \in [-10, 25]$ (m) and $t \in [0, 14]$ (s).

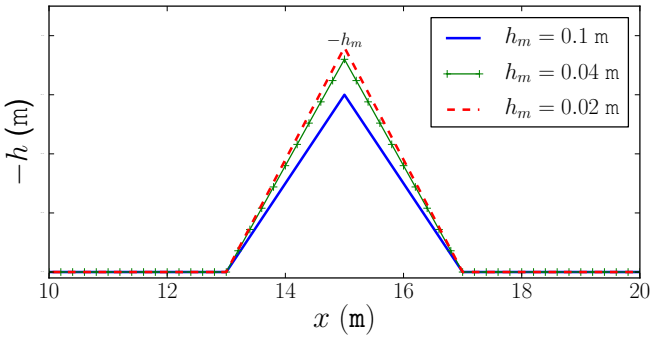


Figure 25.13: Sketch of the three sea bottoms with triangular configurations of height $h_m = 0.1$ m and $\varepsilon = 0.1$ (blue solid line), $h_m = 0.04$ m and $\varepsilon = 0.25$ (green line with plus markers) as well as $h_m = 0.02$ m and $\varepsilon = 0.5$ (red dashed line).

Figure 25.14: The detailed comparison of a wave passing through the triangular submerged bar with $h_m = 0.1$ m, $\varepsilon = 0.1$, $x \in [-25, 50]$ (m) and $t \in [0, 25]$ (s). These solutions are provided by DOLFWAVE using the ZTC/BEP (red dashed line) and Nwogu's BEV (blue solid line) models.

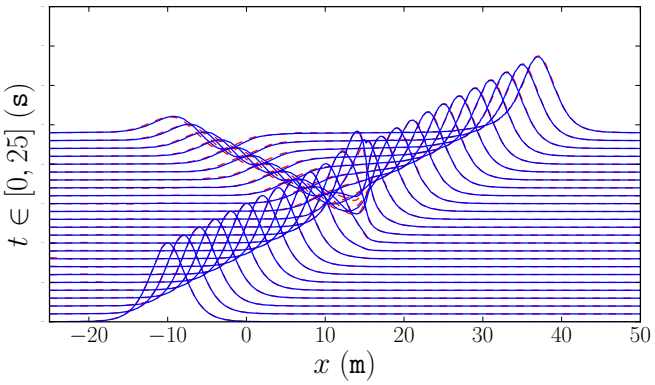
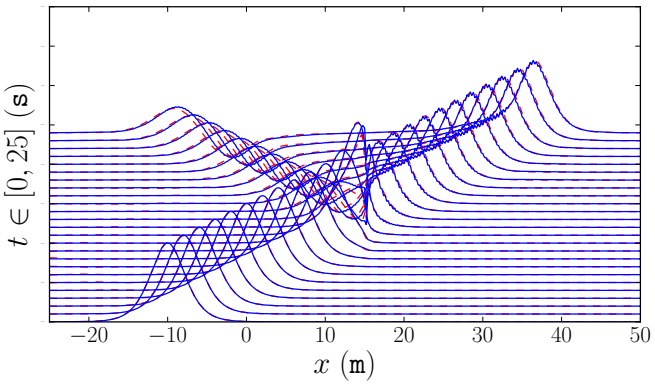


Figure 25.15: The detailed comparison of a wave passing through the triangular submerged bar with $h_m = 0.04$ m, $\varepsilon = 0.25$, $x \in [-25, 50]$ (m) and $t \in [0, 25]$ (s). These solutions are provided by DOLFWAVE using the ZTC/BEP (red dashed line) and Nwogu's BEV (blue solid line) models. Small amplitude and high frequency oscillations are observed in the Nwogu's BEV model solutions.



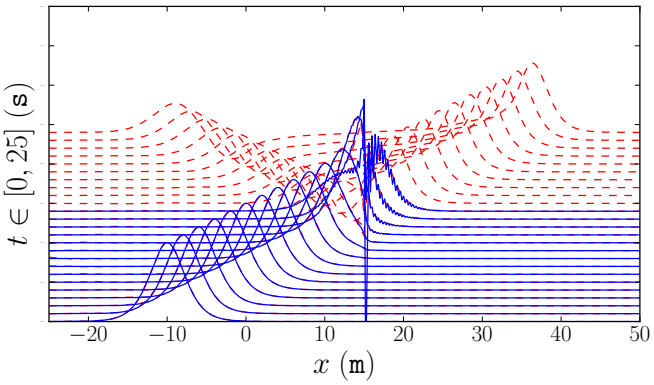


Figure 25.16: The detailed comparison of a wave passing through the triangular submerged bar with $h_m = 0.02$ m, $\varepsilon = 0.5$, $x \in [-25, 50]$ (m) and $t \in [0, 25]$ (s). These solutions are provided by DOLFWAVE using the ZTC/BEP (red dashed line) and Nwogu's BEV (blue solid line) models. The blowup of the solution provided by the Nwogu's BEV model is observed.

25.9.2 A Gaussian hump in a square basin

Here, we simulate the evolution of a Gaussian hump in a square basin. Analogous tests are available in the literature (see, e.g., Wei and Kirby (1995) and Woo and Liu (2004a)). The computational domain is a square of 10×10 m^2 which is discretized using triangular unstructured meshes. Moreover, we provide grid refinement tests to ensure convergence and accuracy. Reflective wall boundary conditions are applied (see (25.38)) and no sponge layers are considered. As initial conditions, we have

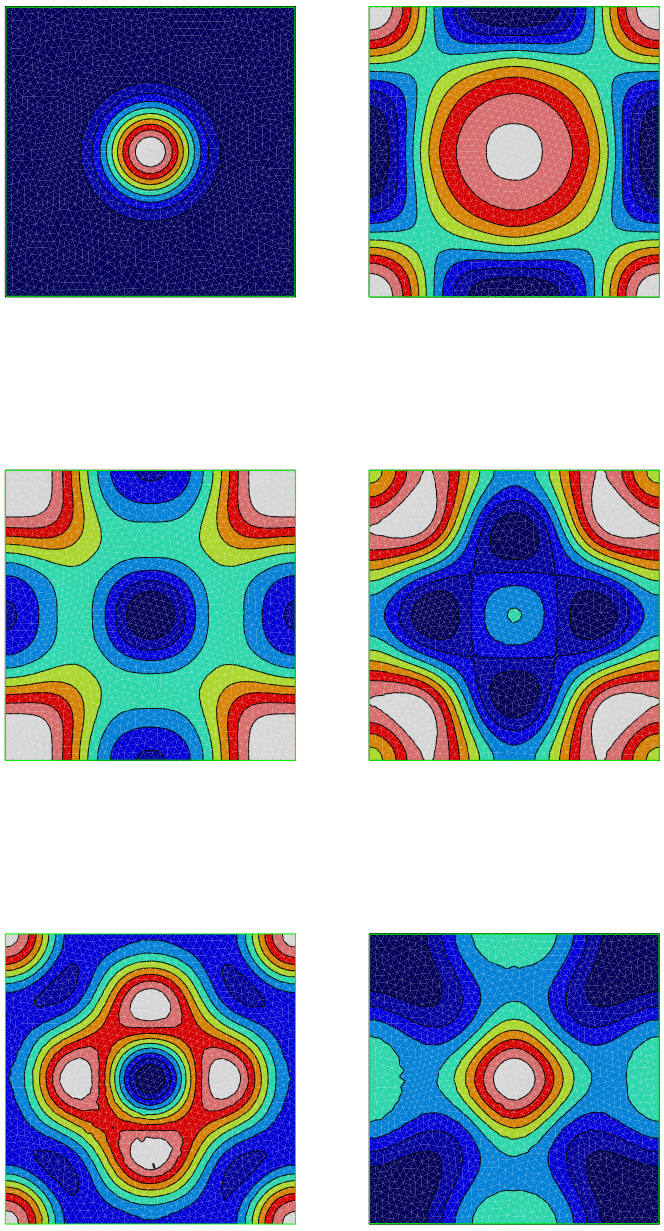
$$\begin{cases} \eta(x,y,0) = 0.1\, e^{-0.4\left((x-5)^2+(y-5)^2\right)} \quad (\mathrm{m}), \\ \Phi(x,y,0) = 0 \quad (\mathrm{m}^2\,\mathrm{s}^{-1}). \end{cases} \tag{25.45}$$

A constant depth $h = 0.5$ m is considered. These initial conditions and see bottom are considered in the FUNWAVE manual (see Kirby (1998)). Even though we do not know the exact solutions of the nonlinear equations, the symmetric characteristics of the problem should result in symmetric surface elevation profiles. These symmetric properties are conserved in the numerical solutions provided by DOLFWAVE even for nonsymmetric unstructured meshes. As an example, we show in Figure 25.17 the isovalues of the wave surface elevation for a mesh with 1873 nodes and $t = 0$ s, $t = 10$ s, $t = 20$ s, $t = 30$ s, $t \approx 40$ s as well as $t \approx 50$ s. Moreover, the volume conservation condition is satisfied with a neglectable error.

In Figure 25.18, we show the time history of the wave surface elevation for the central point $P_0 = (5, 5)$ (m) and for the corner point $P_1 = (0, 0)$ (m), using meshes with 2815, 1364 and 706 nodes. These results are in agreement with those presented in the FUNWAVE manual. A slight phase shift is only observed for $t > 40$ (s). A detailed view of the time history of the wave surface elevation for the central point $P_0 = (5, 5)$ (m) using meshes with 5049, 3964, 2815, 1873, 1364 and 706 nodes is shown in Figure 25.19. A slight discrepancy is only observed among the coarser mesh with 706 nodes and all the finer ones.

In the following table we compare the relative l^2-error (%) among the coarser meshes and the finer one with 5094 nodes, for $t \in [0, 30]$ (s) at the points $P_0 = (5, 5)$ (m) and $P_1 = (0, 0)$ (m), using unstructured meshes with 706, 1324, 1873, 2815 and 3964 nodes.

Figure 25.17: The isovalues of the wave surface elevation η (m) at the time $t = 0$ s, $t = 10$ s, $t = 20$ s, $t = 30$ s, $t \approx 40$ s and $t \approx 50$ s.



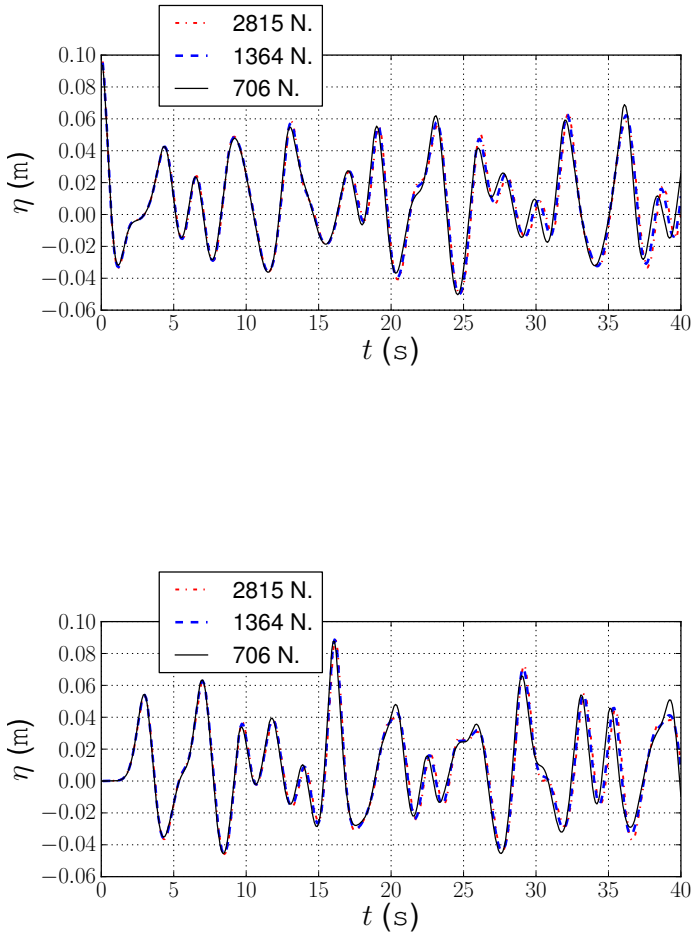


Figure 25.18: The time history of the wave surface elevation η (m) at $P_0 = (5,5)$ (m) (upper panel) and $P_1 = (0,0)$ (m) (lower panel), using unstructured meshes with 2815, 1364 and 706 nodes.

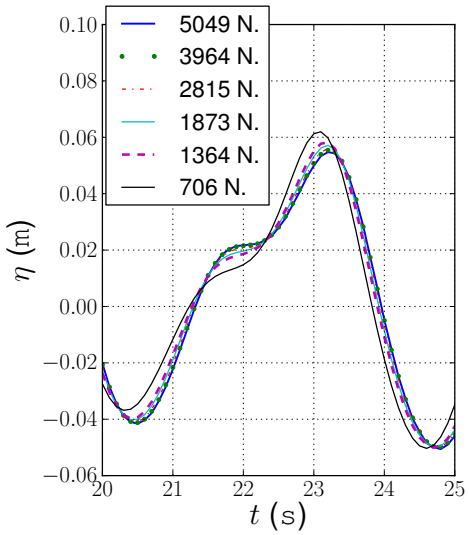


Figure 25.19: A detailed view of the wave surface elevation η (m) for $t \in [20,25]$ (s) at $P_0 = (5,5)$ (m) using several unstructured meshes.

Mesh	$P_0 = (5,5)$ (m)	$P_1 = (0,0)$ (m)
706	6.6%	5.7%
1324	1.2%	1.1%
1873	0.5%	0.5%
2815	0.2%	0.1%
3964	0.1%	0.02%

As the number of mesh nodes is increased this error approaches zero, which gives a good indication of convergence for a certain time range.

25.9.3 Harbor

In this subsection, we present some numerical results about the propagation of surface water waves in a harbor with a geometry similar to that one of Figure 25.1. The finite element discretization of equations (25.27) is declared in the UFL form file given in section 25.5. The DOLFWAVE demo code for this example is available at `dolfwave/demo/2HD/harbor`.

The color scale used in Figs. 25.21–25.24 is presented in Figure 25.20. A schematic description of the fluid domain, namely the bottom profile and the sponge layer can be seen in Figs. 25.21 and 25.22, respectively. Note that a piecewise linear bathymetry is considered. Sponge layers of the type $\nu\nabla^2\Phi$ with the viscosity coefficients given by equation (25.39) are used to absorb the wave energy at the outflow region and to avoid strong interaction between incident and reflected waves in the harbor entrance. A monochromatic periodic wave is introduced at the indicated boundary (Dirichlet BC) in Figure 25.22. This is achieved by considering waves induced by a periodic Dirichlet boundary condition, described by the equations (25.33) and (25.34), with the following characteristics:

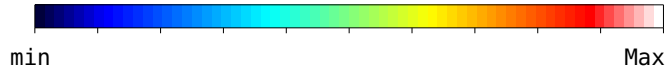


Figure 25.20: Color scale.

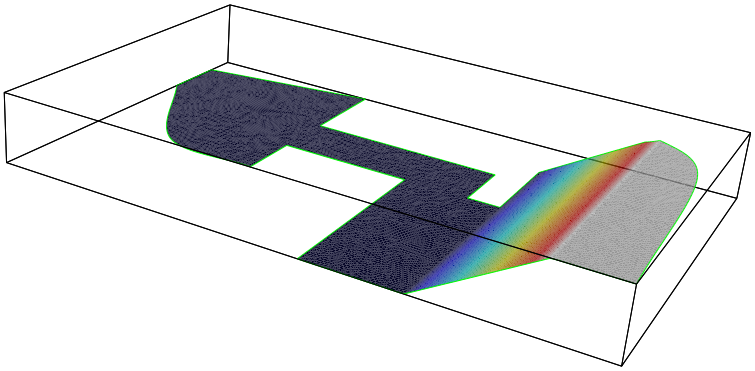


Figure 25.21: Impermeable bottom [Max $= -5.316\,\mathtt{m}$, min $= -13.716\,\mathtt{m}$].

a	wave amplitude	$0.25\,\mathtt{m}$
ω	wave angular frequency	$0.64715\,\mathtt{s}^{-1}$
p	wave period	$4.06614\,\mathtt{s}$
k	wave number	$0.06185\,\mathtt{m}^{-1}$
L	wave length	$101.59474\,\mathtt{m}$
b	wave potential magnitude	$3.97151\,\mathtt{m}^2\mathtt{s}^{-1}$
c	wave velocity magnitude	$0.24562\,\mathtt{m}\,\mathtt{s}^{-1}$
ε	small amplitude parameter	0.01823
μ	long wave parameter	0.13501

Full reflective walls are assumed as boundary conditions in all domain boundary except in the harbor entrance. In Figure 25.23 a snapshot of the wave surface elevation is shown at the time $t_s = 137\,\mathtt{s}$.

A zoom of the image, which describes the physical potential $\phi_0(x, y)$ and velocity vector field in the still water plane, is given in the neighborhood of the point $P_3 = (255, -75)\,(\mathtt{m})$ at t_s (see Figure 25.24). The Figs. 25.25 and 25.26 represent the wave surface elevation and water speed as a function of the time, at the points $P_1 = (-350, 150)\,(\mathtt{m})$, $P_2 = (-125, 60)\,(\mathtt{m})$ and P_3.

From these numerical results, we can conclude that the interaction between incident and reflected waves, near the harbor entrance, can generate waves with amplitudes that almost take the triple value of the incident wave amplitude. We can also observe an analogous behavior for velocities. Note that no mechanism for releasing energy of the reflected waves throughout the incident wave boundary is considered.

25.9.4 Object moving on a horizontal bottom

A wave generated by an object moving on a horizontal bottom with a constant speed is simulated here. The declaration of the finite element discretization of (25.27) is that one described in Section 25.5.

The spatial numerical domain is a rectangular basin of $12.5 \times 6\,\mathtt{m}^2$ discretized with a symmetric uniform mesh with 2100 elements. Full reflective boundary conditions are only considered here. The

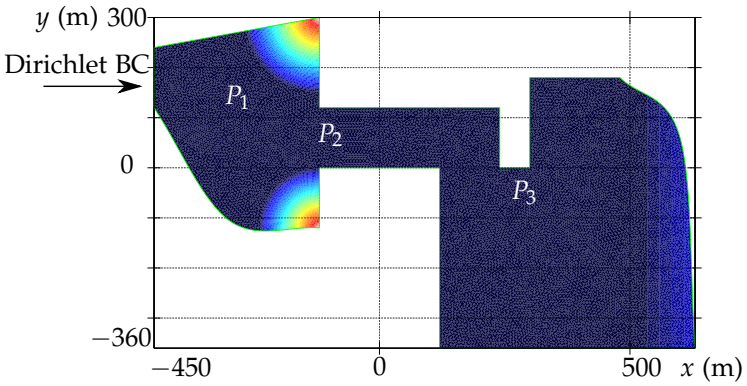


Figure 25.22: Sponge layer (viscosity $\nu(x,y)$) [$\texttt{Max} \approx 0.1\,\texttt{m}^2\texttt{s}^{-1}, \texttt{min} = 0\,\texttt{m}^2\texttt{s}^{-1}$].

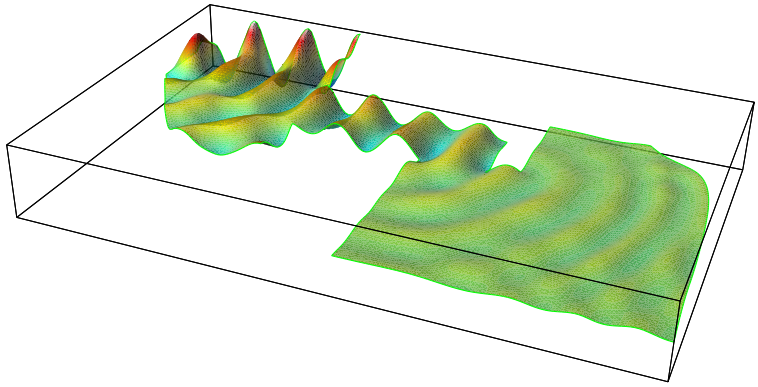


Figure 25.23: Wave surface elevation [$\texttt{Max} \approx 0.63\,\texttt{m}, \texttt{min} \approx -0.73\,\texttt{m}$].

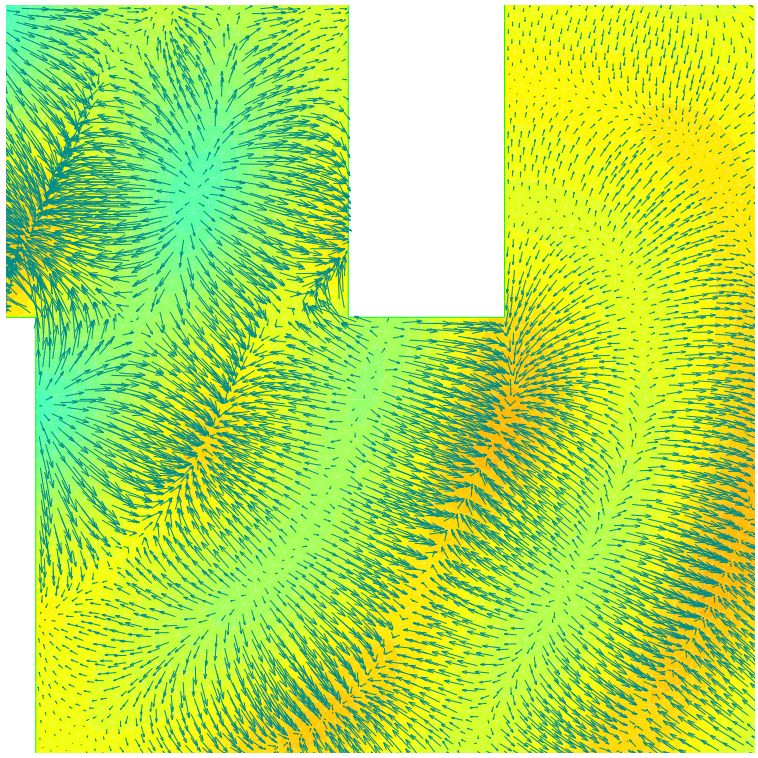


Figure 25.24: Velocity vector field at $z = 0$ and potential $\phi_0(x,y,t_s)$ near P_3. Potential values in Ω: [$\texttt{Max} \approx 14.2\,\texttt{m}^2\texttt{s}^{-1}, \texttt{min} \approx -12.8\,\texttt{m}^2\texttt{s}^{-1}$].

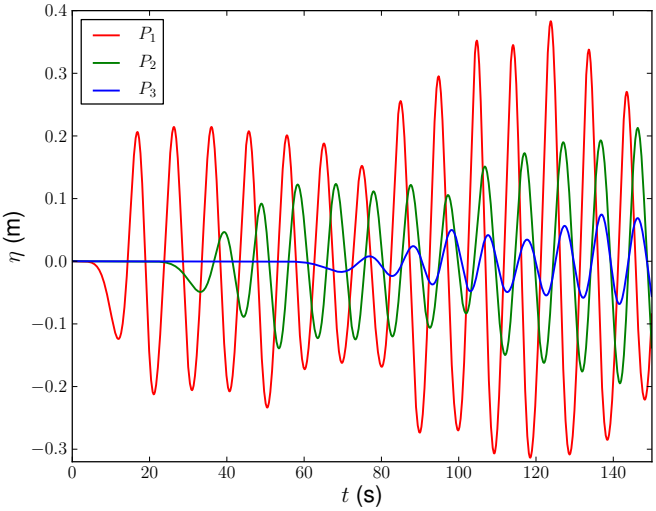


Figure 25.25: Wave surface elevation at P_1, P_2 and P_3 [$\texttt{Max} \approx 0.4\,\texttt{m}, \texttt{min} \approx -0.31\,\texttt{m}$].

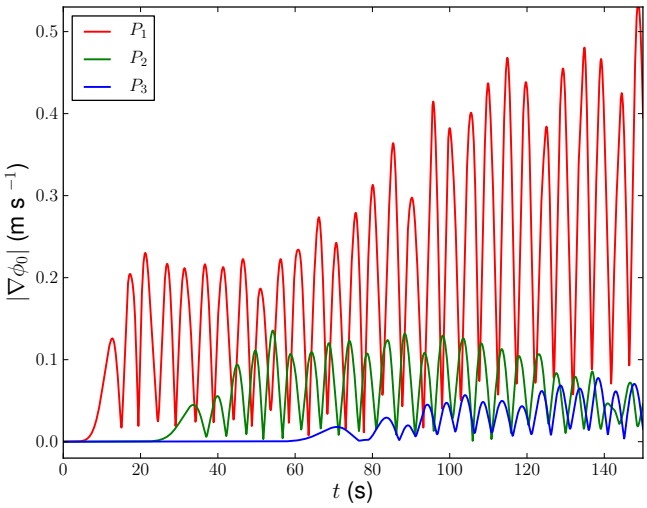


Figure 25.26: Water speed at P_1, P_2 and P_3 [$\texttt{Max} \approx 0.53\,\texttt{m}\,\texttt{s}^{-1}, \texttt{min} = 0\,\texttt{m}\,\texttt{s}^{-1}$].

Figure 25.27: The impermeable bottom $-h(x,y,t)$ (m) at the time $t_0 = 0$ s. [$\texttt{Max} = -0.405\,\texttt{m}, \texttt{min} = -0.45\,\texttt{m}$]

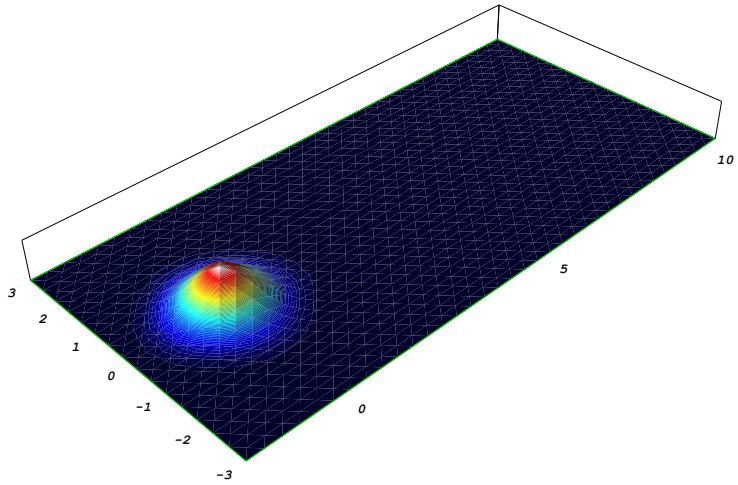
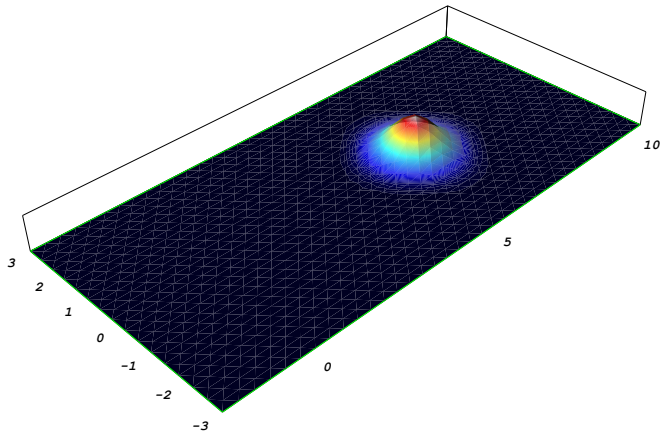


Figure 25.28: The impermeable bottom $-h(x,y,t)$ (m) at the time $t_2 = 6$ s. [$\texttt{Max} = -0.405\,\texttt{m}, \texttt{min} = -0.45\,\texttt{m}$]



moving bottom $h\,(\texttt{m})$ with a constant speed $S_0 = 1\,\texttt{m}\,\texttt{s}^{-1}$ is defined by

$$h(x,y,t) = 0.45 - \frac{\Delta h}{(1+\tanh(1))^4}\bar{X}(x,t)\bar{Y}(y) \tag{25.46}$$

with

$$\bar{X}(x,t) = (1+\tanh(2(x-x_l(t))))(1-\tanh(2(x-x_r(t)))), \tag{25.47}$$

$$\bar{Y}(y) = (1+\tanh(2y+1)))(1-\tanh(2y-1)), \tag{25.48}$$

$$x_l(t) = x_c(t) - \frac{1}{2}, \quad x_r(t) = x_c(t) + \frac{1}{2}, \quad x_c(t) = x_0 + S_0 t, \tag{25.49}$$

where $x_0 = 0\,\texttt{m}$ and $\Delta h = 0.045\,\texttt{m}$ is the maximum thickness of the slide (see Figs. 25.27–25.28).

A time step of $\Delta t = 0.0005\,\texttt{s}$ is considered. In Figs. 25.29–25.32, we show four snapshots of the wave surface elevation provided by the extended ZTC model at the time $t_0 = 1\,\texttt{s}$, $t_1 = 3\,\texttt{s}$, $t_2 = 4.5\,\texttt{s}$ and $t_3 = 6\,\texttt{s}$. Note that we also use here the color scale presented in Figure 25.20.

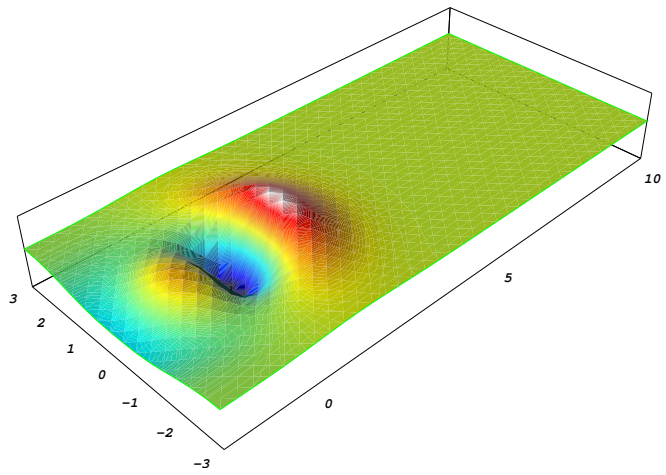


Figure 25.29: The wave surface elevation η (m) at the time $t_0 = 1$ s. [$\texttt{Max} \approx 0.007\,\texttt{m}, \texttt{min} \approx -0.010\,\texttt{m}$]

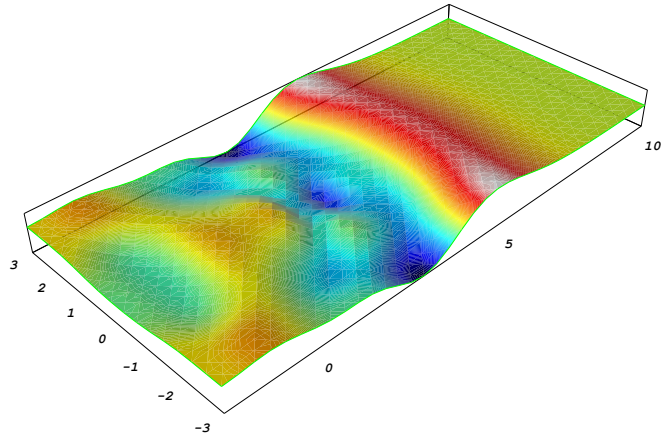


Figure 25.30: The wave surface elevation η (m) at the time $t_1 = 3$ s. [$\texttt{Max} \approx 0.004\,\texttt{m}, \texttt{min} \approx -0.006\,\texttt{m}$]

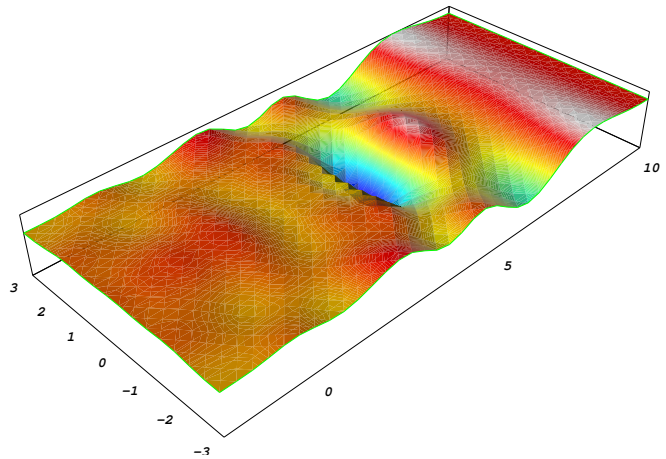
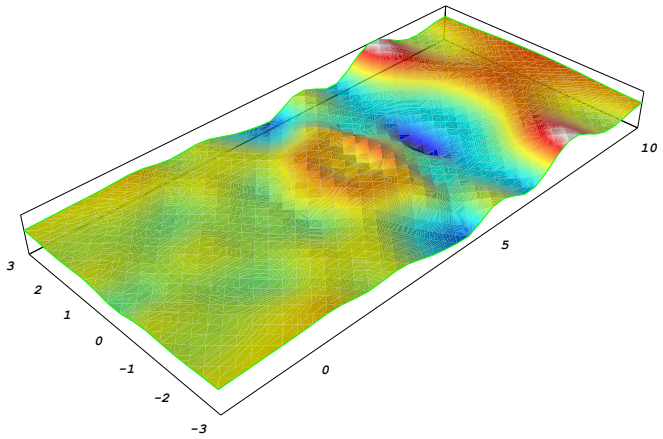


Figure 25.31: The wave surface elevation η (m) at the time $t_2 = 4.5$ s. [$\texttt{Max} \approx 0.004\,\texttt{m}, \texttt{min} \approx -0.011\,\texttt{m}$]

Figure 25.32: The wave surface elevation η (m) at the time $t_3 = 6$ s. [$\texttt{Max} \approx 0.004\,\texttt{m}, \texttt{min} \approx -0.006\,\texttt{m}$]



We refer that the bottom function given by (25.46)–(25.49) is not piecewise linear. In fact, the spatial derivative functions of any order obtained from h are nonzero. Although the extended ZTC model is based on a slowly varying bottom assumption (only $O(h, \nabla h)$ terms are admitted), a good agreement among the solutions presented here with those provided by other models is achieved (see `dolfwave/demo/2HD/hLandslide`). These other models include $O(h, \nabla h, \nabla^2 h)$ terms (see Lopes et al. (2011)).

25.10 Conclusions

As far as we know, the finite element method is not often applied in surface water wave models based on the BEP formulation. In general, finite difference methods are preferred, since they could be easily applied to equations containing spatial derivatives with order higher than 2. On the other hand, they are not appropriate for the treatment of complex geometries, like those of harbors, for instance.

In this work, we extend the BEP model of Zhao et al. (2004) in order to include dissipative effects, as well as, several types of wave generation mechanisms, namely, by moving an impermeable bottom or by the inclusion of a source function. Moreover, we study the influence of a dissipative term in the linear dispersive properties, specifically, in the phase velocity. We show the existence of some cutoff values for the wave number such that the short length waves do not propagate. From a matrix-based linear stability analysis, we can also conclude that the ZTC/BEP model is not prone to instabilities of the type I and II when steep bottom gradients occur or small spatial grid increments are required. On the other hand, the standard potential model with depth averaged velocity potential displays instabilities for certain combinations of the parameters l, h_m and Δx. The eigenvalue spectra of this model exhibit interesting structures putting in evidence high growth rates leading to unstable solutions. Thus, this model should only be applied when gentle bottom variations occur. Since we use the same finite element discretization for both ZTC and standard potential models, some of the unstable wave modes inherent to the latter one may be intrinsic to the partial differential equations and not to the numerical schemes.

The extended ZTC equations are used to model four different physical problems: the evolution of a solitary wave passing through a submerged bar; the evolution of a Gaussian hump in a square basin; the evolution of a periodic wave in a harbor and the generation of a wave due to an object moving on a horizontal bottom. These equations are discretized using Lagrange P_1 elements and a predictor-corrector scheme with an initialization provided by an explicit Runge–Kutta method for the time integration.

In the first physical problem, we can conclude that the numerical model is also stable when there is an interaction between the incident and reflected waves over the submerged bar as well as in one of the domain walls. The shoaling effect over the submerged bar is clearly observed, both for the incident and reflected waves. We compare the solutions of the weakly nonlinear ZTC/BEP and Nwogu/BEV models, for a spike type submerged bar. For the employed finite element discretizations, we observe that the ZTC model is less prone to instabilities than Nwogu's model.

In the second test, the evolution of a Gaussian hump in a square basin is simulated. We obtain a good agreement among the solutions of the ZTC numerical model and those provided in the FUNWAVE manual. Moreover, we perform grid refinement tests to ensure convergence and accuracy.

In the harbor problem, we remark that the interaction between incident and reflected waves, near the harbor entrance, can generate waves with amplitudes and velocities that almost take the triple values of those observed in the incident waves.

In the last numerical example, we refer that the front wave generated by the moving object travels faster than the object. In this way, a subcritical velocity regime associated with the moving object is observed. A good agreement among the numerical solutions presented here with those provided by other models is achieved (see `dolfwave/demo`). From these numerical tests we can conclude that the FEniCS packages, namely DOLFIN, UFL and FFC, are appropriate to model surface water waves, leading to efficient and robust algorithms.

Note that strong nonlinear effects, e.g., negative amplitudes extending below the sea floor, may cause instabilities in a nonlinear wave model. These effects will almost certainly be encountered for instance for a tsunami inundating a shallow sloping beach. Drying and wetting schemes for the treatment of these problems are not yet implemented in DOLFWAVE. Consequently, this type of instabilities will most likely show up when running the proposed model with high amplitude waves and small depth bottoms.

Surface water wave problems are associated with Boussinesq-type governing equations, which require high order spatial derivatives. A first approach to a fourth-order spatial derivative model, using a continuous/discontinuous Galerkin finite element method, can be found in Lopes et al. (2011).

We have been developing DOLFWAVE which is a FEniCS based application for surface water wave models. This package already includes some models with equations containing spatial derivatives of order 4. The current state of the work, along with several numerical simulations, can be found at `http://ptmat.fc.ul.pt/~ndl` and `https://launchpad.net/dolfwave`.

26 *Applications in solid mechanics*

By Kristian B. Ølgaard and Garth N. Wells

Problems in solid mechanics constitute perhaps the largest field of application of finite element methods. The vast majority of solid mechanics problems involve the standard momentum balance equation, posed in a Lagrangian setting, with different models distinguished by the choice of nonlinear or linearized kinematics, and the constitutive model for determining the stress. For some common models, the constitutive relationships are rather complex. This chapter addresses a number of canonical solid mechanics models in the context of automated modeling, and focuses on some pertinent issues that arise due to the nature of the constitutive models. The solution of equations with second-order time derivatives, which characterizes dynamic problems, is also considered.

26.1 Background

We present in this chapter the solution of a collection of common solid mechanics problems using automated code generation techniques. For users familiar with traditional development techniques for solid mechanics problems, it is often not evident how the automation techniques established with the FEniCS Project should be applied to solid mechanics problems. The traditional development approach to solid mechanics problems, and traditional finite element codes, places a strong emphasis on the implementation of constitutive models at the quadrature point level. Automated methods, on the other hand, tend to stress more heavily the governing balance equations. Widely used finite element codes for solid mechanics applications provide application programming interfaces (APIs) for users to implement their own constitutive models. The interface supplies kinematic and history data, and the user code computes the stress tensor, and when required also the linearization of the stress. Users of such libraries will typically not be exposed to code development other than via the constitutive model API.

The purpose of this chapter is to illustrate how problems of relevance in solid mechanics can be solved using automation tools. We consider the common problems of linearized elasticity, plasticity, hyperelasticity and elastic wave propagation. Topics that we address via these problems include 'off-line' computation of stress updates, linearization of problems with off-line stress updates, automatic differentiation and time stepping for problems with second-order time derivatives. The presentation starts with the relevant governing equations and some constitutive models, followed by a summary of a commonly used time stepping method. We then address the important issue of solution and linearization of problems in which the governing equation is expressed in terms of the stress tensor (rather than explicitly in terms of the displacement field, or derivatives of the displacement field), and the stress tensor is computed via a separate algorithm. These topics are then followed by a number of examples that demonstrate implementation approaches. Finally, two future extensions of the FEniCS framework that are particular interesting with respect to solid mechanics problems are summarized.

This chapter does not set out to provide a comprehensive treatment of solid mechanics problems. It

addresses a number of the most frequently encountered issues when applying automated techniques to solid mechanics problems. It should be clear from the considered examples how a wider range of common solid mechanics problems can be tackled using automated modeling.

26.2 Governing equations

26.2.1 Preliminaries

We will consider problems posed on a polygonal domain $\Omega \subset \mathbb{R}^d$, where $1 \leq d \leq 3$. The boundary of Ω, denoted by $\partial\Omega$, is decomposed into regions Γ_D and Γ_N such that $\Gamma_D \cup \Gamma_N = \partial\Omega$ and $\Gamma_D \cap \Gamma_N = \emptyset$. The outward unit normal vector on $\partial\Omega$ will be denoted by n. For time-dependent problems, we will consider a time interval of interest $I = (0, T]$ and let superimposed dots denote time derivatives. We will use Ω to denote the current configuration of a solid body; that is, the domain Ω depends on the displacement field. It is sometimes convenient to also define a reference domain $\Omega_0 \subset \mathbb{R}^d$ that remains fixed. For convenience, we will consider cases in which Ω and Ω_0 coincide at time $t = 0$. To indicate boundaries, outward unit normal vectors, and other quantities relative to Ω_0, the subscript '0' will be used. When considering linearized kinematics, the domains Ω and Ω_0 are both fixed and coincide at all times t. A triangulation of the domain Ω will be denoted by $\mathcal{T}_h$, and a triangulation of the domain Ω_0 will be denoted by $\mathcal{T}_0$. A finite element cell will be denoted by $T \in \mathcal{T}_h$.

The governing equations for the different models will be formulated in the common framework of: find $u \in V$ such that

$$F(u; w) = 0 \quad \forall\, w \in V, \tag{26.1}$$

where $F : V \times V \to \mathbb{R}$ is linear in w and V is a suitable function space. If F is also linear in u, then F can be expressed as

$$F(u; w) := a(u, w) - L(w), \tag{26.2}$$

where $a : V \times V \to \mathbb{R}$ is linear in u and in w, and $L : V \to \mathbb{R}$ is linear in w. For this case, the problem can be cast in the canonical setting of: find $u \in V$ such that

$$a(u, w) = L(w) \quad \forall\, w \in V. \tag{26.3}$$

For nonlinear problems, a Newton method is typically employed to solve (26.1). Linearizing F about $u = u_0$ leads to a bilinear form,

$$a(du, w) := DF_{du}\,(u_0; w) = \left.\frac{dF\,(u_0 + \epsilon du; w)}{d\epsilon}\right|_{\epsilon=0}, \tag{26.4}$$

and a linear form is given by:

$$L(w) := F(u_0, w). \tag{26.5}$$

Using the definitions of a and L in (26.4) and (26.5), respectively, a Newton step involves solving a problem of the type in (26.3), followed by the correction $u_0 \leftarrow u_0 - du$. The process is repeated until (26.1) is satisfied to within a specified tolerance.

26.2.2 Balance of momentum

The standard balance of linear momentum problem for the body Ω reads:

$$\rho\ddot{u} - \nabla \cdot \sigma = b \quad \text{in } \Omega \times I, \tag{26.6}$$

$$u = g \quad \text{on } \Gamma_D \times I, \tag{26.7}$$

$$\sigma n = h \quad \text{on } \Gamma_N \times I, \tag{26.8}$$

$$u(x,0) = u_0 \quad \text{in } \Omega, \tag{26.9}$$

$$\dot{u}(x,0) = v_0 \quad \text{in } \Omega, \tag{26.10}$$

where $\rho : \Omega \times I \to \mathbb{R}$ is the mass density, $u : \Omega \times I \to \mathbb{R}^d$ is the displacement field, $\sigma : \Omega \times I \to \mathbb{R}^d \times \mathbb{R}^d$ is the symmetric Cauchy stress tensor, $b : \Omega \times I \to \mathbb{R}^d$ is a body force, $g : \Omega \times I \to \mathbb{R}^d$ is a prescribed boundary displacement, $h : \Omega \times I \to \mathbb{R}^d$ is a prescribed boundary traction, $u_0 : \Omega \to \mathbb{R}^d$ is the initial displacement and $v_0 : \Omega \to \mathbb{R}^d$ is the initial velocity. To complete the boundary value problem, a constitutive model that relates σ to u is required.

To develop finite element models, it is necessary to cast the momentum balance equation in a weak form by multiplying the balance equation (26.6) by a weight function w and integrating. It is possible to formulate a space-time method by considering a weight function that depends on space and time, and then integrating over $\Omega \times I$. However, it is far more common in solid mechanics applications to consider a weight function that depends on spatial position only and to apply finite difference methods to deal with time derivatives. Following this approach, at a time $t \in I$ we multiply (26.6) by a function w (w is assumed to satisfy $w = 0$ on Γ_D) and integrate over Ω:

$$\int_\Omega \rho\ddot{u} \cdot w \, \mathrm{d}x - \int_\Omega (\nabla \cdot \sigma) \cdot w \, \mathrm{d}x - \int_\Omega b \cdot w \, \mathrm{d}x = 0. \tag{26.11}$$

Applying integration by parts, using the divergence theorem and inserting the boundary condition (26.8), we obtain:

$$F := \int_\Omega \rho\ddot{u} \cdot w \, \mathrm{d}x + \int_\Omega \sigma : \nabla w \, \mathrm{d}x - \int_{\Gamma_N} h \cdot w \, \mathrm{d}s - \int_\Omega b \cdot w \, \mathrm{d}x = 0. \tag{26.12}$$

In this section, the momentum balance equation has been presented on the current configuration Ω. It can also be posed on the fixed reference domain Ω_0 via a pull-back operation. For the particular presentation that we will use in this chapter for geometrically nonlinear models details of the pull-back will not be needed.

26.2.3 Potential energy minimization

An alternative approach to solving static problems (problems without an inertia term) is to consider the minimization of potential energy. This approach leads to the same governing equation when applied to a standard problem, but may be a preferable framework for problems that are naturally posed in terms of stored energy densities and for which external forcing terms are conservative (see Holzapfel (2000, p. 159) for an explanation of conservative loading), and for problems that involve coupled physical phenomena that are best described energetically.

Consider a system for which the total potential energy Π associated with a body can be expressed as

$$\Pi = \Pi_{\text{int}} + \Pi_{\text{ext}}. \tag{26.13}$$

We will consider an internal potential energy functional of the form

$$\Pi_{\text{int}} = \int_{\Omega_0} \Psi_0 \left(u\right) \, \mathrm{d}x, \tag{26.14}$$

where Ψ_0 is the stored strain energy density, and an external potential energy functional of the form

$$\Pi_{\text{ext}} = -\int_{\Omega_0} b_0 \cdot u \, \mathrm{d}x - \int_{\Gamma_{0,N}} h_0 \cdot u \, \mathrm{d}s. \tag{26.15}$$

It is the form of the stored energy density function Ψ_0 that defines a particular constitutive model. For later convenience, the potential energy terms have been presented on the reference domain Ω_0.

A stable solution u to (26.13) minimizes the potential energy:

$$\min_{u \in V} \Pi, \tag{26.16}$$

where V is a suitably defined function space. Minimization of Π corresponds to the directional derivative of Π being zero for all possible variations of u. Therefore, minimization of Π corresponds to solving (26.1) with

$$F(u; w) := D_w \Pi \left(u\right) = \left. \frac{d\Pi \left(u + \epsilon w\right)}{d\epsilon} \right|_{\epsilon=0}. \tag{26.17}$$

For suitable definitions of the stress tensor, it is straightforward to show that minimizing Π is equivalent to solving the balance of momentum problem, for the static case.

26.3 Constitutive models

A constitutive model describes the relationship between stress and deformation. The stress can be defined explicitly in terms of primal functions, it can be implicitly defined via stored energy density functions, or it can be defined as the solution to a secondary problem. The constitutive model can be either linear or nonlinear. In the following sections we present examples of these cases in the form of linearized elasticity, plasticity and hyperelasticity. The expressions for the stress or stored energy density presented in this section can be inserted into the balance equations or the minimization framework in the preceding section to yield a governing equation.

26.3.1 Linearized elasticity

For linearized elasticity, the stress tensor as a function of the strain tensor for an isotropic, homogeneous material is given by

$$\sigma = 2\mu\varepsilon + \lambda \mathrm{tr}(\varepsilon) I, \tag{26.18}$$

where $\varepsilon = \left(\nabla u + (\nabla u)^{\mathrm{T}}\right)/2$ is the strain tensor, μ and λ are the Lamé parameters, and I is the second-order identity tensor. The relationship between the stress and the strain can also be expressed as

$$\sigma = \mathcal{C} : \varepsilon, \tag{26.19}$$

where

$$\mathcal{C}_{ijkl} = \mu \left(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}\right) + \lambda \delta_{ij}\delta_{kl}, \tag{26.20}$$

and δ_{ij} is the Kronecker-Delta.

26.3.2 Flow theory of plasticity

We consider the standard flow theory model of plasticity, and present only the background necessary to support the examples that we will present. In depth coverage can be found in many textbooks, such as Lubliner (2008).

For a geometrically linear plasticity problem, the stress tensor is computed by

$$\sigma = \mathcal{C} : \varepsilon^{\mathrm{e}}, \tag{26.21}$$

where ε^{e} is the elastic part of the strain tensor. It is assumed that the strain tensor can be decomposed additively into elastic and plastic parts:

$$\varepsilon = \varepsilon^{\mathrm{e}} + \varepsilon^{\mathrm{p}}. \tag{26.22}$$

If ε^{e} can be determined, then the stress can be computed.

The stress tensor in classical plasticity models must satisfy the yield criterion:

$$f\left(\sigma, \varepsilon^{\mathrm{p}}, \kappa\right) := \phi\left(\sigma, q_{\mathrm{kin}}\left(\varepsilon^{\mathrm{p}}\right)\right) - q_{\mathrm{iso}}\left(\kappa\right) - \sigma_{\mathrm{y}} \leqslant 0, \tag{26.23}$$

where $\phi\left(\sigma, q_{\mathrm{kin}}\left(\varepsilon^{\mathrm{p}}\right)\right)$ is a scalar effective stress measure, q_{kin} is a stress-like internal variable used to model kinematic hardening, q_{iso} is a scalar stress-like term used to model isotropic hardening, κ is a scalar internal variable and σ_{y} is the initial scalar yield stress. For the commonly adopted von Mises model (also known as J_2-flow) with linear isotropic hardening, ϕ and q_{iso} read:

$$\phi\left(\sigma\right) = \sqrt{\frac{3}{2} s_{ij} s_{ij}}, \tag{26.24}$$

$$q_{\mathrm{iso}}\left(\kappa\right) = H\kappa, \tag{26.25}$$

where $s_{ij} = \sigma_{ij} - \sigma_{kk}\delta_{ij}/3$ is the deviatoric stress and the constant scalar $H > 0$ is a hardening parameter.

In the flow theory of plasticity, the plastic strain rate is given by:

$$\dot{\varepsilon}^{\mathrm{p}} = \dot{\lambda} \frac{\partial g}{\partial \sigma}, \tag{26.26}$$

where $\dot{\lambda}$ is the rate of the plastic multiplier and the scalar g is known as the plastic potential. In the case of associative plastic flow, $g = f$. The term $\dot{\lambda}$ determines the magnitude of the plastic strain rate, and the direction is given by $\partial g / \partial \sigma$. For isotropic strain-hardening, it is usual to set

$$\dot{\kappa} = \sqrt{\frac{2}{3} \dot{\varepsilon}^{\mathrm{p}}_{ij} \dot{\varepsilon}^{\mathrm{p}}_{ij}}, \tag{26.27}$$

which for associative von Mises plasticity implies that $\dot{\kappa} = \dot{\lambda}$.

A feature of the flow theory of plasticity is that the constitutive model is postulated in a rate form. This requires the application algorithms to compute the stress from increments of the total strain. A discussion of algorithmic aspects on how the stress tensor can be computed from the equations presented in this section is postponed to Section 26.6.2.

26.3.3 Hyperelasticity

Hyperelastic models are characterized by the existence of a stored strain energy density function Ψ_0. The linearized model presented at the start of this section falls with the class of hyperelastic models. Assuming linearized kinematics, the stored energy function

$$\Psi_0 = \frac{\lambda}{2}\left(\operatorname{tr}\varepsilon\right)^2 + \mu\varepsilon : \varepsilon, \tag{26.28}$$

corresponds to the linearized model in (26.18). It is straightforward to show that using this stored energy function in the potential energy minimization approach in (26.17) leads to the same equation as inserting the stress from (26.18) into the weak momentum balance equation (26.12).

More generally, stored energy functions that correspond to nonlinear models can be defined. A wide range of stored energy functions for hyperelastic models have been presented and analyzed in the literature (see, for example, Bonet and Wood (1997) for a selection). In order to present concrete examples, it is necessary to introduce some kinematics, and in particular strain measures. The Green–Lagrange strain tensor E is defined in terms of the deformation gradient $F : \Omega_0 \times I \to \mathbb{R}^d \times \mathbb{R}^d$, and right Cauchy–Green tensor $C : \Omega_0 \times I \to \mathbb{R}^d \times \mathbb{R}^d$:

$$F = I + \nabla u, \tag{26.29}$$

$$C = F^{\mathrm{T}}F, \tag{26.30}$$

$$E = \frac{1}{2}\left(C - I\right), \tag{26.31}$$

where I is the second-order identity tensor. Using E in (26.28) in place of the infinitesimal strain tensor ε, we obtain the following expression for the strain energy density function:

$$\Psi_0 = \frac{\lambda}{2}\left(\operatorname{tr}E\right)^2 + \mu E : E, \tag{26.32}$$

which is known as the St. Venant–Kirchhoff model. Unlike the linearized case, this energy density function is not linear in u (or spatial derivatives of u), which means that when minimizing the total potential energy Π, the resulting equations are nonlinear. Another example of a hyperelastic model is the compressible neo-Hookean model:

$$\Psi_0 = \frac{\mu}{2}(I_C - 3) - \mu \ln J + \frac{\lambda}{2}(\ln J)^2, \tag{26.33}$$

where $I_C = \operatorname{tr}C$ and $J = \det F$.

In most presentations of hyperelastic models, one would proceed from the definition of the stored energy function to the derivation of a stress tensor, and then often to a linearization of the stress for use in a Newton method. This process can be lengthy and tedious. For a range of models, features of the Unified Form Language (UFL, Chapter 17) will permit problems to be posed as energy minimization problems, and it will not be necessary to compute expression for a stress tensor, or its linearization, explicitly. A particular model can then be posed in terms of a particular expression for Ψ_0. It is also possible to follow the momentum balance route, in which case UFL can be used to compute the stress tensor and its linearization automatically from an expression for Ψ_0.

26.4 Time integration

In this chapter we focus on the Newmark family of methods, which are widely used in structural dynamics. It is a direct integration method, in which the equations are evaluated at discrete points in time separated by a time increment Δt. Thus, the time step t_{n+1} is equal to $t_n + \Delta t$. While this chapter addresses the Newmark scheme, it is straightforward to extend the approach (and implementation) to generalized-α methods.

The Newmark relations between displacements, velocities and accelerations at t_n and t_{n+1} read:

$$u_{n+1} = u_n + \Delta t \dot{u}_n + \frac{1}{2}\Delta t^2 \left(2\beta \ddot{u}_{n+1} + (1-2\beta)\, \ddot{u}_{n+1}\right), \tag{26.34}$$

$$\dot{u}_{n+1} = \dot{u}_n + \Delta t \left(\gamma \ddot{u}_{n+1} + (1-\gamma)\, \ddot{u}_n\right), \tag{26.35}$$

where β and γ are parameters. Various well-known schemes are recovered for particular combinations of β and γ. Setting $\beta = 1/4$ and $\gamma = 1/2$ leads to the trapezoidal scheme, and setting $\beta = 0$ and $\gamma = 1/2$ leads to a central difference scheme. For $\beta > 0$, re-arranging (26.34) and using (26.35) leads to:

$$\ddot{u}_{n+1} = \frac{1}{\beta \Delta t^2}\left(u_{n+1} - u_n - \Delta t \dot{u}_n\right) - \left(\frac{1}{2\beta} - 1\right) \ddot{u}_n, \tag{26.36}$$

$$\dot{u}_{n+1} = \frac{\gamma}{\beta \Delta t}\left(u_{n+1} - u_n\right) - \left(\frac{\gamma}{\beta} - 1\right) \dot{u}_n - \Delta t \left(\frac{\gamma}{2\beta} - 1\right) \ddot{u}_n, \tag{26.37}$$

in which u_{n+1} is the only unknown term on the right-hand side.

To solve a time dependent problem, the governing equation can be posed at time t_{n+1},

$$F\left(u_{n+1}; w\right) = 0 \quad \forall\, w \in V, \tag{26.38}$$

with the expressions in (26.36) and (26.37) used for first and second time derivatives of u at time t_{n+1}.

26.5 Linearization issues for complex constitutive models

Solving problems with nonlinear constitutive models, such as plasticity, using Newton's method requires linearization of (26.12). There are two particular issues that deserve attention. The first is that if the stress σ is computed via some algorithm, then proper linearization of F requires linearization of the algorithm for computing the stress, and not linearization of the continuous problem. This point is well known in computational plasticity, and has been extensively studied (Simo and Taylor, 1985). The second issue is that the stress field, and its linearization, will not in general come from a finite element space. Hence, if all functions are assumed to be in a finite element space, or are interpolated in a finite element space, suboptimal convergence of a Newton method will be observed.

26.5.1 Consistency of linearization

To illustrate the second issue raised in the preceding paragraph, we consider the representation that the FEniCS Form Compiler (FFC, Chapter 11) would generate for a simple model problem, and linearize this representation. We then consider how FFC would represent a linearization of the original problem, which turns out not to be consistent with the linearization of the FFC representation of the original problem.

Consider the following one-dimensional problem:

$$F(u;w) := \int_\Omega \sigma w_{,x} \, \mathrm{d}x, \tag{26.39}$$

where the scalar stress σ is a nonlinear function of the strain field $u_{,x}$, and will be computed via a separate algorithm outside of the main forms. We consider a continuous, piecewise quadratic displacement field (and likewise for w), and a strain field that is computed via an L^2-projection onto the space of discontinuous, piecewise linear elements (for the considered spaces, this is equivalent to a direct evaluation of the strain). We also represent the stress σ on the discontinuous, piecewise linear basis. Since the polynomial degree of the integrand is two, (26.39) can be integrated using two Gauss quadrature points on an element $T \in \mathcal{T}_h$:

$$f_{T,i_1} := \sum_{q=1}^{2} \sum_{\alpha=1}^{2} \psi_\alpha^T(x_q) \, \sigma_\alpha \phi_{i_1,x}^T(x_q) \, W_q, \tag{26.40}$$

where q is the integration point index, α is the degree of freedom index for the local basis of σ, ψ^T and ϕ^T denotes the linear and quadratic basis functions on the element T, respectively, and W_q is the quadrature weight at integration point x_q. Note that σ_α is the stress at the element node α.

To apply a Newton method, the Jacobian (linearization) of (26.40) is required. This will be denoted by $A_{T,i}^\star$. To achieve quadratic convergence of a Newton method, the linearization must be exact. The Jacobian of (26.40) is:

$$A_{T,i}^\star := \frac{df_{T,i_1}}{du_{i_2}}, \tag{26.41}$$

where u_{i_2} are the displacement degrees of freedom. In (26.41), only σ_α depends on du_{i_2}, and the linearization of this terms reads:

$$\frac{d\sigma_\alpha}{du_{i_2}} = \frac{d\sigma_\alpha}{d\varepsilon_\alpha} \frac{d\varepsilon_\alpha}{du_{i_2}} = D_\alpha \frac{d\varepsilon_\alpha}{du_{i_2}}, \tag{26.42}$$

where D_α is the tangent. To compute the values of the strain at nodes, ε_α, from the displacement field, the derivative of the displacement field is evaluated at x_α:

$$\varepsilon_\alpha = \sum_{i_2=1}^{3} \phi_{i_2,x}^T(x_\alpha) \, u_{i_2}. \tag{26.43}$$

Inserting (26.42) and (26.43) into (26.41) yields:

$$A_{T,i}^\star = \sum_{q=1}^{2} \sum_{\alpha=1}^{2} \psi_\alpha^T(x_q) D_\alpha \phi_{i_2,x}^T(x_\alpha) \phi_{i_1,x}^T(x_q) W_q. \tag{26.44}$$

This is the exact linearization of (26.40).

We now consider linearization of (26.39), which leads to the bilinear form:

$$a(u,w) = \int_\Omega D u_{,x} \, w_{,x} \, \mathrm{d}x, \tag{26.45}$$

where $D = d\sigma/d\varepsilon$. If D is represented using a discontinuous, piecewise linear basis, and two quadrature points are used to integrate the form (which is exact for this form), the resulting element

matrix is:

$$A_{T,i} = \sum_{q=1}^{2} \sum_{\alpha=1}^{2} \psi_{\alpha}^{T}(x_q) D_{\alpha} \phi_{i_2,x}^{T}(x_q) \phi_{i_1,x}^{T}(x_q) W_q. \tag{26.46}$$

The above representation is what would be produced by FFC.

Equations (26.44) and (26.46) are not identical since $x_q \neq x_\alpha$. As a consequence, the bilinear form in (26.46) is not an exact linearization of (26.39), and a Newton method will therefore exhibit suboptimal convergence. In general, the illustrated problem arises when some coefficients in a form are computed by a nonlinear operation elsewhere, and then interpolated and evaluated at a point that differs from where the coefficients were computed. This situation is different from the use of nonlinear operators in UFL, and compiled by FFC. An example of such an operator is the $\ln J$ term in the neo-Hookean model (26.33) where J will be computed at quadrature points during assembly after which the operator ln is applied to compute $\ln J$.

The linearization issue highlighted in this section is further illustrated in the following section, as too is a solution that involves the definition of so-called 'quadrature elements'.

26.5.2 Quadrature elements

To introduce the concept of quadrature elements, we first present a model problem that will be used in numerical examples. Given the finite element space

$$V = \left\{ w \in H_0^1(\Omega),\ w \in P_q(T)\ \forall\, T\ \in \mathcal{T}_h \right\}, \tag{26.47}$$

where $\Omega \subset \mathbb{R}$ and $q \geq 1$, the model problem of interest involves: given $f \in V$, find $u \in V$ such that

$$F := \int_{\Omega} \left(1 + u^2\right) u_{,x} w_{,x}\, \mathrm{d}x - \int_{\Omega} f w\, \mathrm{d}x = 0 \quad \forall\, w \in V. \tag{26.48}$$

Solving this problem via Newton's method involves solving a series of linear problems with

$$L\left(w\right) := \int_{\Omega} \left(1 + u_n^2\right) u_{n,x} w_{n,x}\, \mathrm{d}x - \int_{\Omega} f w\, \mathrm{d}x, \tag{26.49}$$

$$a\left(du_{n+1}, w\right) := \int_{\Omega} \left(1 + u_n^2\right) du_{n+1,x} w_{,x}\, \mathrm{d}x + \int_{\Omega} 2 u_n u_{n,x} du_{n+1} w_{,x}\, \mathrm{d}x, \tag{26.50}$$

with the update $u_n \leftarrow u_n - du_{n+1}$. To draw an analogy with complex constitutive models, we rephrase the above as:

$$L\left(w\right) := \int_{\Omega} \sigma_n w_{,x}\, \mathrm{d}x - \int_{\Omega} f w\, \mathrm{d}x, \tag{26.51}$$

$$a\left(du_{n+1}, w\right) := \int_{\Omega} C_n du_{n+1,x} w_{,x}\, \mathrm{d}x + \int_{\Omega} 2 u_n u_{n,x} du_{n+1} w_{,x}\, \mathrm{d}x, \tag{26.52}$$

where $\sigma_n = \left(1 + u_n^2\right) u_{n,x}$ and $C_n = \left(1 + u_n^2\right)$. The forms now resemble those for a plasticity problem where, σ is the 'stress', C is the 'tangent' and $u_{,x}$ is the 'strain'.

Similar to a plasticity problem, we wish to compute σ and C 'off-line', and to supply σ and C as functions in a space Q to the forms used in the Newton solution process. To access $u_{n,x}$ for use off-line, an approach is to perform an L^2-projection of the derivative of u onto a space Q. For the example in question, we will also project $1 + u^2$ onto Q. A natural choice would be to make Q one polynomial order less that V and discontinuous across cell facets. However, following this approach leads to a convergence rate for a Newton solver that is less than the expected quadratic rate. The reason for this is that the linearization that follows from this process is not consistent with the problem being solved

as explained in the previous section.

To resolve this issue within the context of UFL and FFC, the concept of *quadrature elements* has been developed. This special type of element is used to represent 'functions' that can only be evaluated at particular points (quadrature points), and cannot be differentiated. In the remainder of this section we present some key features of the quadrature element and a demonstration of its use for the model problem considered above. A quadrature element is declared in UFL by:

Python code

```
element = FiniteElement("Quadrature", tetrahedron, q)
```

where `q` is the polynomial degree that the underlying quadrature rule will be able to integrate exactly. The declaration of a quadrature element is similar to the declaration of any other element in UFL and it can be used as such, with some limitations. Note, however, the subtle difference that the element order does not refer to the polynomial degree of the finite element shape functions, but instead relates to the quadrature scheme. For 'sufficient' integration of a second-order polynomial in three dimensions, FFC will use four quadrature points per cell. FFC interprets the quadrature points of the quadrature element as degrees of freedom where the value of a shape function for a degree of freedom is equal to one at the quadrature point and zero otherwise. This has the implication that a function that is defined on a quadrature element can only be evaluated at quadrature points. Furthermore, it is not possible to take derivatives of functions defined on a quadrature element.

Before demonstrating the importance of quadrature elements when computing terms off-line, we illustrate a simple usage of a quadrature element. Consider the bilinear form for a mass matrix weighted by a coefficient f that is defined on a quadrature element:

$$a\left(u,w\right)=\int_{\Omega}fuw\,\mathrm{d}x. \tag{26.53}$$

If the test and trial functions w and u come from a space of linear Lagrange functions, the polynomial degree of their product is two. This means that the coefficient f should be defined as:

Python code

```
ElementQ = FiniteElement("Quadrature", tetrahedron, 2)
f = Coefficient(ElementQ)
```

to ensure appropriate integration of the form in (26.53). The reason for this is that the quadrature element in the form dictates the quadrature scheme that FFC will use for the numerical integration since the quadrature element, as described above, only have nonzero values at points that coincide with the underlying quadrature scheme of the quadrature element. Thus, if the degree of `ElementQ` is set to one, the form will be integrated using only one integration point, since one point is enough to integrate a linear polynomial exactly, and as a result the form is under integrated. If quadratic Lagrange elements are used for w and u, the polynomial degree of the integrand is four, therefore the declaration for the coefficient f should be changed to:

Python code

```
ElementQ = FiniteElement("Quadrature", tetrahedron, 4)
f = Coefficient(ElementQ)
```

The DOLFIN code for solving the nonlinear model problem with a source term $f=x^2-4$, and Dirichlet boundary conditions $u=1$ at $x=0$, continuous quadratic elements for V, and quadrature elements of degree two for Q is shown in Figure 26.1. The relative residual norm after each iteration of the Newton solver for four different combinations of spaces V and Q is shown in Table 26.1. Continuous, discontinuous and quadrature elements are denoted by CG_q, DG_q and Q_q respectively

Iteration	CG_1/DG_0	CG_1/Q_1	CG_2/DG_1	CG_2/Q_2
1	1.114e+00	1.101e+00	1.398e+00	1.388e+00
2	2.161e-01	2.319e-01	2.979e-01	2.691e-01
3	3.206e-03	3.908e-03	2.300e-02	6.119e-03
4	7.918e-07	7.843e-07	1.187e-03	1.490e-06
5	9.696e-14	3.662e-14	2.656e-05	1.242e-13
6			5.888e-07	
7			1.317e-08	
8			2.963e-10	

Table 26.1: Computed relative residual norms after each iteration of the Newton solver for the nonlinear model problem using different elements for V and Q. Quadratic convergence is observed when using quadrature elements, and when using piecewise constant functions for Q, which coincides with a one-point quadrature element. The presented results are computed using the code in Figure 26.1.

where q refers to the polynomial degree as discussed previously. It is clear from the table that using quadratic elements for V requires the use of quadrature elements in order to ensure quadratic convergence of the Newton solver.

26.6 Implementation and examples

We present in this section implementation examples that correspond to the afore presented models. Where feasible, complete solvers are presented. When this is not feasible, relevant code extracts are presented. Python examples are preferred due the compactness of the code extracts, however, in the case of plasticity efficiency demands a C++ implementation. It is possible in the future that an efficient Python interface for plasticity problems will be made available via just-in-time compilation.

The examples are chosen to highlight some implementation aspects that are typical for solid mechanics applications. In the code extracts, we do not provide commentary on generic aspects, such as the creation of meshes, application of boundary conditions and the solution of linear systems. For an explanation of such aspects in the code examples, we refer to Chapters 1 and 10.

26.6.1 Linearized elasticity

This example is particularly simple since the stress can be expressed as a straightforward function of the displacement field, and the expression for the stress in (26.18) can be inserted directly into (26.12). For the steady case (inertia terms are ignored), a complete solver for a linearized elasticity problem is presented in Figure 26.2. The solver in Figure 26.2 is for a simulation on a unit cube with a source term $b = (1, 0, 0)$ and $u = 0$ on $\partial\Omega$. A continuous, piecewise quadratic finite element space is used. The expressiveness of the UFL input means that the expressions for `sigma` and `F` in Figure 26.2 resemble closely the mathematical expressions used in the text for σ and F. We have presented this problem in Figure 26.2 in terms of F to unify our presentation of linear and nonlinear equations, and used the UFL functions `lhs` and `rhs` to automatically extract the bilinear and linear forms, respectively, from `F`.

26.6.2 Plasticity

The computation of the stress tensor, and its linearization, for the model outlined in Section 26.3.2 in a displacement-driven finite element model is rather involved. A method of computing point-wise a stress tensor that satisfies (26.23) from the strain, strain increment and history variables is known as a 'return mapping algorithm'. Return mapping strategies are discussed in detail in Simo and Hughes

Python code

```
from dolfin import *

# Sub domain for Dirichlet boundary condition
class DirichletBoundary(SubDomain):
    def inside(self, x, on_boundary):
        return abs(x[0] - 0.0) < DOLFIN_EPS and on_boundary

# Class for interfacing with the Newton solver
class NonlinearModelProblem(NonlinearProblem):
    def __init__(self, a, L, u, C, S, Q, bc):
        NonlinearProblem.__init__(self)
        self.a, self.L = a, L
        self.u, self.C, self.S, self.Q, self.bc = u, C, S, Q,
             bc

    def F(self, b, x):
        assemble(self.L, tensor=b)
        self.bc.apply(b, x)

    def J(self, A, x):
        assemble(self.a, tensor=A)
        self.bc.apply(A)

    def form(self, A, b, x):
        C = project((1.0 + self.u**2), self.Q)
        self.C.vector()[:] = C.vector()

        S = project(Dx(self.u, 0), self.Q)
        self.S.vector()[:] = S.vector()
        self.S.vector()[:] = self.S.vector()*self.C.vector()

# Create mesh and define function spaces
mesh = UnitInterval(8)
V = FunctionSpace(mesh, "Lagrange", 2)
Q = FunctionSpace(mesh, "Q", 2)

# Define boundary condition
bc = DirichletBC(V, Constant(1.0), DirichletBoundary())

# Define source and functions
f = Expression("x[0]*x[0] - 4")
u, C, S = Function(V), Function(Q), Function(Q)

# Define variational problems
w  = TestFunction(V)
du = TrialFunction(V)
L  = S*Dx(w, 0)*dx - f*w*dx
a  = C*Dx(du, 0)*Dx(w, 0)*dx + 2*u*Dx(u, 0)*du*Dx(w, 0)*dx

# Create nonlinear problem, solver and solve
problem = NonlinearModelProblem(a, L, u, C, S, Q, bc)
solver  = NewtonSolver()
solver.solve(problem, u.vector())
```

Figure 26.1: DOLFIN implementation for the nonlinear model problem in (26.48) with 'off-line' computation of terms used in the variational forms.

Figure 26.2: DOLFIN Python solver for a simple linearized elasticity problem on a unit cube.

Python code

```
from dolfin import *

# Create mesh
mesh = UnitCube(8, 8, 8)

# Create function space
V = VectorFunctionSpace(mesh, "Lagrange", 2)

# Create test and trial functions, and source term
u, w = TrialFunction(V), TestFunction(V)
b = Constant((1.0, 0.0, 0.0))

# Elasticity parameters
E, nu = 10.0, 0.3
mu, lmbda = E/(2.0*(1.0 + nu)), E*nu/((1.0 + nu)*(1.0 -
    2.0*nu))

# Stress
sigma = 2*mu*sym(grad(u)) +
    lmbda*tr(grad(u))*Identity(w.cell().d)

# Governing balance equation
F = inner(sigma, grad(w))*dx - dot(b, w)*dx

# Extract bilinear and linear forms from F
a, L = lhs(F), rhs(F)

# Dirichlet boundary condition on entire boundary
c  = Constant((0.0, 0.0, 0.0))
bc = DirichletBC(V, c, DomainBoundary())

# Set up PDE and solve
u = Function(V)
problem = LinearVariationalProblem(a, L, u, bcs=bc)
solver = LinearVariationalSolver(problem)
solver.parameters["symmetric"] = True
solver.solve()
```

(1998). A widely used return mapping approach, the 'closest-point projection', is summarized below for a plasticity model with linear isotropic hardening.

From (26.21) and (26.22) the stress at the end of a strain increment reads:

$$\sigma_{n+1} = \mathcal{C} : (\varepsilon_{n+1} - \varepsilon^{\mathrm{p}}_{n+1}). \tag{26.54}$$

Therefore, given ε_{n+1}, it is necessary to determine the plastic strain $\varepsilon^{\mathrm{p}}_{n+1}$ in order to compute the stress. In a closest-point projection method the increment in plastic strain is computed from:

$$\varepsilon^{\mathrm{p}}_{n+1} - \varepsilon^{\mathrm{p}}_{n} = \Delta\lambda \frac{\partial g\left(\sigma_{n+1}\right)}{\partial \sigma}, \tag{26.55}$$

where g is the plastic potential function and $\Delta\lambda = \lambda_{n+1} - \lambda_n$. Since $\partial_\sigma g$ is evaluated at σ_{n+1}, (26.54) and (26.55) constitute as system of coupled equations with unknowns $\Delta\lambda$ and σ_{n+1}. In general, the system is nonlinear. To obtain a solution, Newton's method is employed as follows, with k denoting the iteration number. First, a 'trial stress' is computed:

$$\sigma_{\mathrm{trial}} = \mathcal{C} : (\varepsilon_{n+1} - \varepsilon^{\mathrm{p}}_{n}). \tag{26.56}$$

Subtracting (26.56) from (26.54) and inserting (26.55), the following equation is obtained:

$$R_{n+1} := \sigma_{n+1} - \sigma_{\mathrm{trial}} + \Delta\lambda \mathcal{C} : \frac{\partial g\left(\sigma_{n+1}\right)}{\partial \sigma} = 0, \tag{26.57}$$

where R_{n+1} is the 'stress residual'. During the Newton iterations this residual is driven towards zero. If the trial stress in (26.56) leads to satisfaction of the yield criterion in (26.23), then σ_{trial} is the new stress and the Newton procedure is terminated. Otherwise, the Newton increment of $\Delta\lambda$ is computed from:

$$d\lambda_k = \frac{f_k - R_k : Q_k : \partial_\sigma f_k}{\partial_\sigma f_k : \Xi_k : \partial_\sigma g_k + h}, \tag{26.58}$$

where $Q = \left[I + \Delta\lambda \mathcal{C} : \partial^2_{\sigma\sigma} g\right]^{-1}$, $\Xi = Q : \mathcal{C}$ and h is a hardening parameter, which for the von Mises model with linear hardening is equal to H (the constant hardening parameter). The stress increment is computed from:

$$\Delta\sigma_k = \left[-d\lambda_k \mathcal{C} : \partial_\sigma g_k - R_k\right] : Q_k, \tag{26.59}$$

after which the increment of the plastic multiplier and the stresses for the next iteration can be computed:

$$\Delta\lambda_{k+1} = \Delta\lambda_k + d\lambda_k, \tag{26.60}$$

$$\sigma_{k+1} = \sigma_k + \Delta\sigma_k. \tag{26.61}$$

The yield criterion is then evaluated again using the updated values, and the procedure continues until the yield criterion is satisfied to within a prescribed tolerance. Note that to start the procedure $\Delta\lambda_0 = 0$ and $\sigma_0 = \sigma_{\mathrm{trial}}$. After convergence is achieved, the consistent tangent can be computed:

$$C_{\mathrm{tan}} = \Xi - \frac{\Xi : \partial_\sigma g \otimes \partial_\sigma f : \Xi}{\partial_\sigma f : \Xi : \partial_\sigma g + h}, \tag{26.62}$$

which is used when assembling the global Jacobian (stiffness matrix). The return mapping algorithm is applied at all quadrature points.

The closest-point return mapping algorithm described above is common to a range of plasticity

Figure 26.3: `PlasticityModel` public interface defined by the plasticity library. Users are required to supply implementations for at least the pure virtual functions. These functions describe the plasticity model.

C++ code

```
class PlasticityModel
{
public:

  /// Constructor
  PlasticityModel(double E, double nu);

  /// Return hardening parameter
  virtual double hardening_parameter(double eps_eq) const;

  /// Equivalent plastic strain
  virtual double kappa(double eps_eq, const arma::vec& stress,
                       double lambda_dot) const;

  /// Value of yield function f
  virtual double f(const arma::vec& stress,
                   double equivalent_plastic_strain) const =
                        0;

  /// First derivative of f with respect to sigma
  virtual void df(arma::vec& df_dsigma,
                  const arma::vec& stress) const = 0;

  /// First derivative of g with respect to sigma
  virtual void dg(arma::vec& dg_dsigma,
                  const arma::vec& stress) const;

  /// Second derivative of g with respect to sigma
  virtual void ddg(arma::mat& ddg_ddsigma,
                   const arma::vec& stress) const = 0;

};
```

models that are defined by the form of the functions f and g. The process can be generalized for models with more complicated hardening behavior. To aid the implementation of different models, a return mapping algorithm and support for quadrature point level history parameters is provided by the FEniCS Plasticity library (`https://launchpad.net/fenics-plasticity`). The library adopts a polymorphic design, with the base class `PlasticityModel` providing an interface for users to implement, and thereby supply functions for f, $\partial_\sigma f$, $\partial_\sigma g$, and $\partial_{\sigma\sigma} g$. Figure 26.3 shows the `PlasticityModel` class public interface. Supplied with details of f (and possibly g), the library can compute stress updates and linearizations using the closest-point projection method.

Computational efficiency is important in the return mapping algorithm as the stress and its linearization are computed at all quadrature points at each global Newton iteration. Therefore, it is necessary to execute the algorithm in C++ rather than in Python. For this reason, the FEniCS Plasticity library provides a C++ interface only at this stage. To reconcile ease and efficiency, it would be possible to use just-in-time compilation for a Python implementation of the `PlasticityModel` interface, just as DOLFIN presently does for the `Expression` class (see Chapter 10).

We now outline a solver based on the FEniCS Plasticity library. Firstly, the UFL input for a formulation in three dimensions using a continuous, piecewise quadratic basis is shown in Figure 26.4. Note that the stress and the linearized tangent are supplied as coefficients to the form as they are computed inside the plasticity library. Symmetry has been exploited to flatten the stress and the tangent terms. Note also in Figure 26.4 that quadrature elements are used for the coefficients `s` and `t`. Recall from Section 26.5 that when constitutive updates are computed outside of the form file care

Python code

```
element  = VectorElement("Lagrange", tetrahedron, 2)
elementT = VectorElement("Quadrature", tetrahedron, 2, 36)
elementS = VectorElement("Quadrature", tetrahedron, 2, 6)

u, w =  TrialFunction(element), TestFunction(element)
b, h = Coefficient(element), Coefficient(element)
t, s = Coefficient(elementT), Coefficient(elementS)

def eps(u):
  return as_vector([u[i].dx(i) for i in range(3)] \
+ [u[i].dx(j) + u[j].dx(i) for i, j in [(0, 1), (0, 2), (1,
    2)]])

def sigma(s):
  return as_matrix([[s[0], s[3], s[4]],
                    [s[3], s[1], s[5]],
                    [s[4], s[5], s[2]]])

def tangent(t):
  return as_matrix([[t[i*6 + j] for j in range(6)] for i in
      range(6)])

a = inner(dot(tangent(t), eps(u)), eps(w))*dx
L = inner(sigma(s), grad(w))*dx - dot(b, w)*dx - dot(h, w)*ds
```

Figure 26.4: Definition of the linear and bilinear variational forms for plasticity expressed using UFL syntax.

must be taken to ensure quadratic convergence of a Newton method. By using quadrature elements in Figure 26.4, it is possible to achieve quadratic convergence during a Newton solve for plasticity problems.

The solver is implemented in C++, and Figure 26.5 shows an extract of the most relevant parts of the solver in the context of plasticity. First, the necessary function spaces are created. `V` is used to define the bilinear and linear forms and the displacement field u, while `Vt` and `Vs` are used for the two coefficient spaces: the consistent tangent and the stress, which enter the bilinear and linear forms of the plasticity problem. The forms defining the plasticity problem are then created and the relevant functions are attached to the forms. Then the object defining the plasticity model is created. The class `VonMises` is a subclass of the `PlasticityModel` class shown in Figure 26.3 and it implements functions for f, $\partial_\sigma f$ and $\partial_{\sigma\sigma} g$. It is constructed with values for the Young's modulus, Poisson's ratio, yield stress and linear hardening parameter. This object can then be passed to the constructor of the `PlasticityProblem` class along with the forms, displacement field u, coefficient functions and boundary conditions. `PlasticityProblem` is a subclass of the DOLFIN class `NonlinearProblem`, which is described in Chapter 10. The `PlasticityProblem` class handles the assembly over cells, loops over cell quadrature points, and variable updates. The `PlasticityProblem` is solved by the `NewtonSolver` like any other `NonlinearProblem` object. After each Newton solver the history variables are updated by calling the `update_variables`, function before proceeding with the next solution increment.

26.6.3 *Hyperelasticity*

We present the construction of a solver for a hyperelastic problem that is phrased as a minimization problem, following the minimization framework that was presented in Section 26.2.3. The compressible neo-Hookean model in (26.33) is adopted. The automatic functional differentiation features of UFL permit the solver code to resemble closely the abstract mathematical presentation. Noteworthy in this approach is that it is not necessary to provide an explicit expression for the stress tensor. Changing model is therefore as simple as redefining the stored energy density function Ψ_0.

Figure 26.5: Code extract for solving a plasticity problem.

C++ code

```
// Create mesh and define function spaces
UnitCube mesh(4, 4, 4);
Plasticity::FunctionSpace V(mesh);
Plasticity::BilinearForm::CoefficientSpace_t Vt(mesh);
Plasticity::LinearForm::CoefficientSpace_s Vs(mesh);

// Create forms and attach functions
Function tangent(Vt);
Plasticity::BilinearForm a(V, V);
a.t = tangent;

Function stress(Vs);
Plasticity::LinearForm L(V);
L.s = stress;

// Displacements
Function u(V);

// Young's modulus and Poisson's ratio
double E = 20000.0; double nu = 0.3;

// Slope of hardening (linear) and hardening parameter
double E_t(0.1*E);
double hardening_parameter = E_t/(1.0 - E_t/E);

// Yield stress
double yield_stress = 200.0;

// Object of class von Mises
fenicsplas::VonMises J2(E, nu, yield_stress,
     hardening_parameter);

// Create PlasticityProblem
fenicsplas::PlasticityProblem nonlinear_problem(a, L, u,
     tangent, stress, bcs, J2);

// Create nonlinear solver
NewtonSolver nonlinear_solver;

// Pseudo time stepping parameters
double t = 0.0; double dt  = 0.005; double T = 0.02;

// Apply load in steps
while (t < T)
{
  // Increment time and solve nonlinear problem
  t += dt;
  nonlinear_solver.solve(nonlinear_problem, u.vector());

  // Update variables for next load step
  nonlinear_problem.update_variables();
}
```

A complete hyperelastic solver is presented in Figure 26.6. It corresponds to a problem posed on a unit cube, and loaded by a body force $b_0 = (0, -0.5, 0)$, and restrained such that $u = (0, 0, 0)$ where $x = 0$. Elsewhere on the boundary the traction $h_0 = (0.1, 0, 0)$ is applied. Continuous, piecewise linear functions for the displacement field are used. The code in Figure 26.6 adopts the same notation used in Sections 26.2.3 and 26.3.3. The problem is posed on the reference domain, and for convenience the subscripts '0' have been dropped in the code.

The solver in Figure 26.6 solves the problem using one Newton step. For problems with stronger nonlinearities, perhaps as a result of greater volumetric or surface forcing terms, it may be necessary to apply a pseudo time-stepping approach and solve the problem in number of Newton increments, or it may be necessary to apply a path following solution method.

26.6.4 Elastodynamics

We present now a linearized elastodynamics problem to illustrate the solution of time-dependent problems. The example is based on the Newmark family of methods presented in Section 26.4. For this example, we consider a viscoelastic model that is a minor extension of the elasticity model in (26.18). For this model, the stress tensor is given by:

$$\sigma = 2\mu\varepsilon + (\lambda \mathrm{tr}(\varepsilon) + \eta \mathrm{tr}(\dot{\varepsilon}))\, I, \tag{26.63}$$

where the constant scalar $\eta \geq 0$ is a viscosity parameter.

A simple, but complete, elastodynamics solver is presented in Figures 26.7 and 26.8. The solver mirrors the notation used in Section 26.4, with expressions for the acceleration, velocity and displacement at time t_n (`a0`, `v0`, `u0`), and expressions for the acceleration and velocity at time t_{n+1} (`a1`, `v1`) in terms of the displacement at t_{n+1} (`u1`) and other fields at time t_n. For simplicity, the source term $b = (0, 0, 0)$. The body is fixed such that $u = (0, 0, 0)$ at $x = 0$ and the initial conditions are $u_0 = v_0 = (0, 0, 0)$. A traction h is applied at $x = 1$ and is increased linearly from zero to one over the first five time steps. Therefore, no forces are acting on the body at $t = 0$ and the initial acceleration is zero. Again, the UFL functions `lhs` and `rhs` have been used to extract the bilinear and linear terms from the form. This is particularly convenient for time-dependent problems since it allows the code implementation to be posed in the same format as is usually adopted in the mathematical presentation, with the equation of interest posed in terms of fields at some point between times t_n and t_{n+1}. The presented solver could be made more efficient by exploiting linearity of the governing equation and thereby re-using the factorization of the system matrix.

26.7 Future developments

In this chapter we have presented a range of solid mechanics problems in the context of automated modeling. The implementation of the models was shown to be relatively straightforward due to the high level of abstraction provided in the FEniCS framework. The presented cases cover a range of typical solid mechanics problems that can currently be solved using FEniCS tools. To broaden the range of problems that can be handled in the FEniCS framework the following two extensions are of particular interest from a solid mechanics viewpoint:

Assembly of forms on manifolds Currently, it is assumed that two-dimensional elements, like triangles, are embedded in $\mathbb{R}^2$ and three-dimensional elements, like tetrahedra, are embedded in $\mathbb{R}^3$. An improvement would be to support two-dimensional elements embedded in $\mathbb{R}^3$ and one-dimensional elements embedded in $\mathbb{R}^2$ or $\mathbb{R}^3$. This would, among other things, provide support for shell and truss problems within the automated framework.

Figure 26.6: Complete DOLFIN solver for the compressible neo-Hookean model, formulated as a minimization problem.

Python code

```python
from dolfin import *

# Optimization options for the form compiler
parameters["form_compiler"]["cpp_optimize"] = True

# Create mesh and define function space
mesh = UnitCube(16, 16, 16)
V = VectorFunctionSpace(mesh, "Lagrange", 1)

def left(x):
    return x[0] < DOLFIN_EPS

# Define Dirichlet boundary (x = 0 or x = 1)
zero = Constant((0.0, 0.0, 0.0))
bc = DirichletBC(V, zero, left)

# Define test and trial functions
du, w = TrialFunction(V), TestFunction(V)

# Displacement from previous iteration
u  = Function(V)
b  = Constant((0.0, -0.5, 0.0))  # Body force per unit mass
h  = Constant((0.1,  0.0, 0.0))  # Traction force on the
    boundary

# Kinematics
I = Identity(V.cell().d)    # Identity tensor
F = I + grad(u)             # Deformation gradient
C = F.T*F                   # Right Cauchy-Green tensor

# Invariants of deformation tensors
Ic, J = tr(C), det(F)

# Elasticity parameters
E, nu = 10.0, 0.3
mu, lmbda = E/(2*(1 + nu)), E*nu/((1 + nu)*(1 - 2*nu))

# Stored strain energy density (compressible neo-Hookean
    model)
Psi = (mu/2)*(Ic - 3) - mu*ln(J) + (lmbda/2)*(ln(J))**2

# Total potential energy
Pi = Psi*dx - dot(b, u)*dx - dot(h, u)*ds

# Compute first variation of Pi (directional derivative about
    u in the direction of v)
F = derivative(Pi, u, w)

# Compute Jacobian of F
dF = derivative(F, u, du)

# Create nonlinear variational problem and solve
problem = NonlinearVariationalProblem(F, u, bcs=bc, J=dF)
solver = NonlinearVariationalSolver(problem)
solver.solve()

# Save solution in VTK format
file = File("displacement.pvd");
file << u;
```

Python code

```
from dolfin import *

# Form compiler options
parameters["form_compiler"]["cpp_optimize"] = True
parameters["form_compiler"]["optimize"] = True

# External load
class Traction(Expression):
    def __init__(self, end):
        Expression.__init__(self)
        self.t = 0.0
        self.end = end

    def eval(self, values, x):
        values[0] = 0.0
        values[1] = 0.0
        if x[0] > 1.0 - DOLFIN_EPS:
            values[0] = self.t/self.end if self.t < self.end
                 else 1.0

    def value_shape(self):
        return (2,)

def update(u, u0, v0, a0, beta, gamma, dt):
    # Get vectors (references)
    u_vec, u0_vec  = u.vector(), u0.vector()
    v0_vec, a0_vec = v0.vector(), a0.vector()

    # Update acceleration and velocity
    a_vec = (1.0/(2.0*beta))*( (u_vec - u0_vec -
         v0_vec*dt)/(0.5*dt*dt) - (1.0-2.0*beta)*a0_vec )

    # v = dt * ((1-gamma)*a0 + gamma*a) + v0
    v_vec = dt*((1.0-gamma)*a0_vec + gamma*a_vec) + v0_vec

    # Update (t(n) <-- t(n+1))
    v0.vector()[:], a0.vector()[:] = v_vec, a_vec
    u0.vector()[:] = u.vector()

# Load mesh and define function space
mesh = UnitSquare(32, 32)

# Define function space
V = VectorFunctionSpace(mesh, "Lagrange", 1)

# Test and trial functions
u1, w = TrialFunction(V), TestFunction(V)

E, nu  = 10.0, 0.3
mu, lmbda = E/(2.0*(1.0 + nu)), E*nu/((1.0 + nu)*(1.0 -
     2.0*nu))

# Mass density and viscous damping coefficient
rho, eta = 1.0, 0.2

# Time stepping parameters
beta, gamma = 0.25, 0.5
dt   = 0.1
t, T = 0.0, 20*dt

# Fields from previous time step (displacement, velocity,
     acceleration)
u0, v0, a0 = Function(V), Function(V), Function(V)
h = Traction(T/4.0)
```

Figure 26.7: Python code for solving for a dynamic problem using an implicit Newmark scheme. Program continues in Figure 26.8.

Figure 26.8: Continuation of Python code extract for solving for a dynamic problem.

Python code

```
# Velocity and acceleration at t_(n+1)
v1 = (gamma/(beta*dt))*(u1 - u0) - (gamma/beta - 1.0)*v0 -
     dt*(gamma/(2.0*beta) - 1.0)*a0
a1 = (1.0/(beta*dt**2))*(u1 - u0 - dt*v0) - (1.0/(2.0*beta) -
     1.0)*a0

# Stress tensor
def sigma(u, v):
    return 2.0*mu*sym(grad(u)) + (lmbda*tr(grad(u)) +
         eta*tr(grad(v)))*Identity(u.cell().d)

# Governing equation
F = (rho*dot(a1, w) + inner(sigma(u1, v1), sym(grad(w))))*dx -
      dot(h, w)*ds

# Extract bilinear and linear forms
a, L = lhs(F), rhs(F)

# Set up boundary condition at left end
zero = Constant((0.0, 0.0))
def left(x):
    return x[0] < DOLFIN_EPS
bc = DirichletBC(V, zero, left)

# Set up PDE, advance in time and solve
u = Function(V)
problem = LinearVariationalProblem(a, L, u, bcs=bc)
solver = LinearVariationalSolver(problem)
# Save solution in VTK format
file = File("displacement.pvd")
while t <= T:
    t += dt
    h.t = t
    solver.solve()
    update(u, u0, v0, a0, beta, gamma, dt)
    file << u
```

Isoparametric elements This issue relates to quadrilateral and hexahedral elements, which are currently not supported, and to elements with higher order mappings that allow curved mesh boundaries to be represented.

27 *A computational framework for nonlinear elasticity*

By Harish Narayanan

Nonlinear elasticity theory plays a fundamental role in modeling the mechanical response of many polymeric and biological materials. Such materials are capable of undergoing finite deformation, and their material response is often characterized by complex, nonlinear constitutive relationships. (See, for example, Holzapfel (2000) and Truesdell and Noll (1965) and the references within for several examples.) Because of these difficulties, predicting the response of arbitrary structures composed of such materials to arbitrary loads requires numerical computation, usually based on the finite element method. The steps involved in the construction of the required finite element algorithms are classical and straightforward in principle, but their application to non-trivial material models are typically tedious and error-prone. Our recent work on an automated computational framework for nonlinear elasticity, CBC.Twist, is an attempt to alleviate this problem.

The focus of this chapter will be to describe the design and implementation of CBC.Twist, as well as providing examples of its use. The goal is to allow researchers to easily pose and solve problems in nonlinear elasticity in a straightforward manner, so that they may focus on higher-level modeling questions without being hindered by specific implementation issues.

What follows is the proposed outline for the chapter.

The chapter begins with a summary of some key results from classical nonlinear elasticity theory. This discussion is used to motivate the design of CBC.Twist, which is a DOLFIN (Logg and Wells, 2010) module written in UFL syntax (Alnæs and Logg, 2009) that closely resembles how the theory is written down on paper. In particular, we will see how one can easily pose sophisticated material models purely at the level of specifying a strain energy function. The discourse will then turn to the primary equation of interest: the balance of linear momentum of a body posed in the reference configuration. A finite element scheme for this equation will then be presented, pointing out how CBC.Twist leverages the automatic linearization capabilities of UFL to implement this scheme in a manner that is independent of the specific material model. The time-stepping schemes that CBC.Twist implements will also be discussed. With this in place, we turn to increasingly complex examples to see how initial- boundary-value problems in nonlinear elasticity can be posed and solved in CBC.Twist using only a few lines of high-level code. The chapter concludes with some remarks on how one can obtain CBC.Twist, along with ideas for its extension.

27.1 Brief overview of nonlinear elasticity theory

The goal of this section is to present an overview of the mathematical theory of nonlinear elasticity, which plays an important role in the design of CBC.Twist. Readers interested in a more comprehensive treatment of the subject are referred to, for example, the classical treatises of Truesdell and Toupin

(1960) and Truesdell and Noll (1965), or more modern works such as Gurtin (1981), Ogden (1997) and Holzapfel (2000).

27.1.1 Posing the question we aim to answer

The theory begins by idealizing the elastic body of interest as an open subset of $\mathbb{R}^{2,3}$ with a piecewise smooth boundary. At a *reference* placement of the body, Ω, points in the body are identified by their reference positions, $X \in \Omega$. The treatment presented in this chapter is posed in terms of fields which are parametrized by reference positions. This is commonly termed the *material* or *Lagrangian* description.

In its most basic terms, the *deformation* of the body over time $t \in [0, T]$ is a sufficiently smooth bijective map $\varphi : \overline{\Omega} \times [0, T] \to \mathbb{R}^{2,3}$, where $\overline{\Omega} := \overline{\Omega \cup \partial\Omega}$ and $\partial\Omega$ is the boundary of Ω. The restrictions on the map ensure that the motion it describes is physical and within the range of applicability of the theory (e.g., disallowing the interpenetration of matter or the formation of cracks). From this map, we can construct the *displacement field*,

$$u(X, t) = \varphi(X, t) - X, \tag{27.1}$$

which represents the displacement of a point in time relative to its reference position.

With this brief background, we are ready to pose the fundamental question that CBC.Twist is designed to answer: Given a body comprised of a specified elastic material, what is the displacement of the body when it is subjected to prescribed:

- *Body forces:* These include forces such as the self-weight of a body, forces on ferromagnetic materials in magnetic fields, etc., which act everywhere in the volume of the body. They are denoted by the vector field $B(X, t)$.
- *Traction forces:* This is the force measured per unit surface area acting on the *Neumann* boundary of the body, $\partial\Omega_{\mathrm{N}}$, and denoted by the vector field $T(X, t)$.
- *Displacement boundary conditions:* These are displacement fields prescribed on the *Dirichlet* boundary of the body, $\partial\Omega_{\mathrm{D}}$.

It is assumed that $\partial\Omega_{\mathrm{N}} \cap \partial\Omega_{\mathrm{D}} = \emptyset$ and $\overline{\partial\Omega_{\mathrm{N}} \cup \partial\Omega_{\mathrm{D}}} = \partial\Omega$. These details are depicted in Figure 27.1.

27.1.2 The basic equation we need to solve

In order to determine the displacement of an elastic body subjected to these specified loads and boundary conditions, we turn to a fundamental law called the *balance of linear momentum*. This is a law which is valid for all materials and must hold for all time. CBC.Twist solves the Lagrangian form of this equation, which is presented below in local form that is pertinent to numerical implementation by the finite element method:

$$\rho \frac{\partial^2 u}{\partial t^2} = \mathrm{Div}(P) + B \text{ in } \Omega, \tag{27.2}$$

where ρ is the *reference density* of the body, P is the *first Piola–Kirchhoff stress tensor*, $\mathrm{Div}(\cdot)$ is the divergence operator and B is the body force per unit volume. Along with (27.2), we have initial conditions $u(X, 0) = u_0(X)$ and $\frac{\partial u}{\partial t}(X, 0) = v_0(X)$ in Ω, and boundary conditions $u(X, t) = g(X, t)$ on $\partial\Omega_{\mathrm{D}}$ and $P(X, t)N(X) = T(X, t)$ on $\partial\Omega_{\mathrm{N}}$. Here, $N(X)$ is the outward normal on the boundary at the point X.

We focus on the balance of linear momentum because, in a continuous sense, the other fundamental balance principles that materials must obey—the balance of mass (continuity equation), balance of

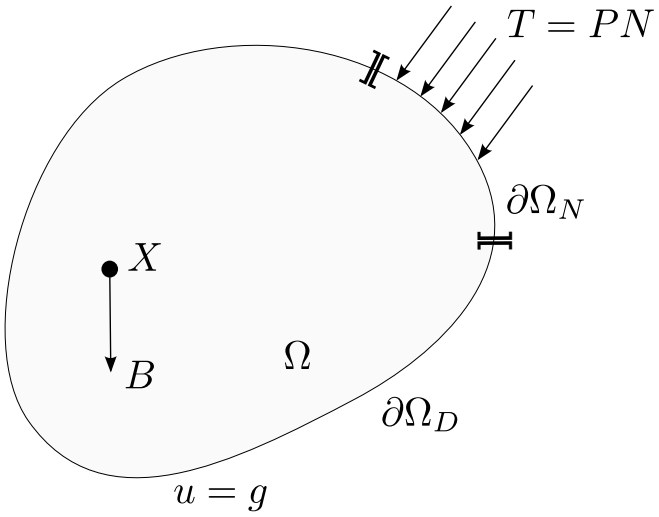


Figure 27.1: An elastic body idealized as a continuum, subjected to body forces, B, surface tractions, T, and prescribed displacement boundary conditions.

angular momentum and balance of energy—are each trivially satisfied[1] in the Lagrangian description by elastic materials with suitably chosen stress responses.

27.1.3 Accounting for different materials

It is important to reiterate that (27.2) is valid for all materials. In order to differentiate between different materials and to characterize their specific mechanical responses, the theory turns to *constitutive relationships*, which are models for describing the real mechanical behavior of matter. In the case of nonlinear elastic (or *hyperelastic*) materials, this description is usually posed in the form of a stress-strain relationship through an objective and frame-indifferent Helmholtz free energy function called the *strain energy function*, ψ. This is an energy defined per unit reference volume and is solely a function of the local *strain measure*. Comprehensive texts on the subject (e.g. Holzapfel (2000)) cover the motivations for defining different forms of strain measures, but in this chapter we just provide the definitions of some of the most common forms. In what follows, $\mathrm{Grad}(\cdot)$ is the gradient operator, and $\mathrm{Tr}(\cdot)$ and $\mathrm{Det}(\cdot)$ are the trace and determinant of $\cdot$, respectively.

In CBC.Twist, each of the forms listed in Table 27.1 have been implemented in the file `kinematics.py` in UFL notation that closely resemble their definitions above. Figure 27.2 presents a section of this file. Notice that it is straightforward to introduce other custom measures as required.

The stress response of isotropic hyperelastic materials (the class of materials CBC.Twist restricts its attention to) can be derived from the scalar-valued strain energy function. In particular, the tensor known as the *second Piola–Kirchhoff stress tensor* is defined using the following *constitutive relationship*:

$$S = F^{-1}\frac{\partial\psi(F)}{\partial F}. \tag{27.3}$$

The second Piola–Kirchhoff stress tensor is related to the first Piola–Kirchhoff stress tensor introduced earlier through the relation, $P = FS$.

[1]It should be noted that the story is not so simple in the context of numerical approximations. For instance, when modeling (nearly) incompressible materials, it is well known that the ill-conditioned stiffness matrix resulting from the conventional Galerkin approximation (discretizing only the displacement field) can result in *volumetric locking*. One can work around this difficulty by resorting to a mixed formulation of the Hu–Washizu type (Simo and Hughes, 1998), but such a formulation is beyond the scope of the current chapter. CBC.Twist can be extended to such a formulation, but for now, we circumvent the problem by restricting our attention to compressible materials.

Infinitesimal strain tensor	$\epsilon = \frac{1}{2}\left(\mathrm{Grad}(u) + \mathrm{Grad}(u)^{\mathrm{T}}\right)$
Deformation gradient	$F = 1 + \mathrm{Grad}(u)$
Right Cauchy–Green tensor	$C = F^{\mathrm{T}}F$
Green–Lagrange strain tensor	$E = \frac{1}{2}\left(C - 1\right)$
Left Cauchy–Green tensor	$b = FF^{\mathrm{T}}$
Euler–Almansi strain tensor	$e = \frac{1}{2}\left(1 - b^{-1}\right)$
Volumetric and isochoric decomposition of C	$\bar{C} = J^{-\frac{2}{3}}C, \quad J = \mathrm{Det}(F)$
Principal invariants of C	$I_1 = \mathrm{Tr}(C)$, $I_2 = \frac{1}{2}\left(I_1^2 - \mathrm{Tr}(C^2)\right)$, $I_3 = \mathrm{Det}(C)$
Principal stretches and directions	$C = \sum_{A=1}^{3} \lambda_A^2 N^A \otimes N^A, \quad \|\|N^A\|\| = 1$

Table 27.1: Definitions of some common strain measures.

Python code

```
# Deformation gradient
def DeformationGradient(u):
    I = SecondOrderIdentity(u)
    return variable(I + Grad(u))

# Determinant of the deformation gradient
def Jacobian(u):
    F = DeformationGradient(u)
    return variable(det(F))

# Right Cauchy-Green tensor
def RightCauchyGreen(u):
    F = DeformationGradient(u)
    return variable(F.T*F)

# Green-Lagrange strain tensor
def GreenLagrangeStrain(u):
    I = SecondOrderIdentity(u)
    C = RightCauchyGreen(u)
    return variable(0.5*(C - I))

# Invariants of an arbitrary tensor, A
def Invariants(A):
    I1 = tr(A)
    I2 = 0.5*(tr(A)**2 - tr(A*A))
    I3 = det(A)
    return [I1, I2, I3]

# Isochoric part of the deformation gradient
def IsochoricDeformationGradient(u):
    F = DeformationGradient(u)
    J = Jacobian(u)
    return variable(J**(-1.0/3.0)*F)

# Isochoric part of the right Cauchy-Green tensor
def IsochoricRightCauchyGreen(u):
    C = RightCauchyGreen(u)
    J = Jacobian(u)
    return variable(J**(-2.0/3.0)*C)
```

Figure 27.2: Samples of how strain measures are implemented in CBC.Twist. Notice that the definitions in the implementation closely resemble the classical forms introduced in Table 27.1.

Figure 27.3: Partial listing of the method that suitably computes the second Piola–Kirchhoff stress tensor based on the chosen strain measure.

Python code

```
def SecondPiolaKirchhoffStress(self, u):

    ...

    if kinematic_measure == "InfinitesimalStrain":
        epsilon = self.epsilon
        S = diff(psi, epsilon)
    elif kinematic_measure == "RightCauchyGreen":
        C = self.C
        S = 2*diff(psi, C)
    elif kinematic_measure == "GreenLagrangeStrain":
        E = self.E
        S = diff(psi, E)
    elif kinematic_measure == "CauchyGreenInvariants":
        I = self.I; C = self.C
        I1 = self.I1; I2 = self.I2; I3 = self.I3
        gamma1 = diff(psi, I1) + I1*diff(psi, I2)
        gamma2 = -diff(psi, I2)
        gamma3 = I3*diff(psi, I3)
        S = 2*(gamma1*I + gamma2*C + gamma3*inv(C))

    ...
    return S
```

As already mentioned, the strain energy function can be posed in equivalent forms in terms of different strain measures. (Again, the interested reader is directed to classical texts to motivate this.) In order to then arrive at the second Piola–Kirchhoff stress tensor, we turn to the chain rule of differentiation. For example,

$$\begin{aligned} S &= 2\frac{\partial\psi(C)}{\partial C} = \frac{\partial\psi(E)}{\partial E} \\ &= 2\left[\left(\frac{\partial\psi(I_1,I_2,I_3)}{\partial I_1} + I_1\frac{\partial\psi(I_1,I_2,I_3)}{\partial I_2}\right)1 - \frac{\partial\psi(I_1,I_2,I_3)}{\partial I_2}C + I_3\frac{\partial\psi(I_1,I_2,I_3)}{\partial I_3}C^{-1}\right] \\ &= \sum_{A=1}^{3}\frac{1}{\lambda_A}\frac{\partial\psi(\lambda_1,\lambda_2,\lambda_3)}{\partial\lambda_A}N^A\otimes N^A = \dots \end{aligned} \tag{27.4}$$

Using definitions such as the ones explicitly provided in (27.4), CBC.Twist computes the second Piola–Kirchhoff stress tensor from the strain energy function by suitably differentiating it with respect to the appropriate strain measure. This allows the user to easily specify material models in terms of each of the strain measures introduced in Table 27.1. The base class for all material models, `MaterialModel`, encapsulates this functionality. The relevant method of this class is provided in Figure 27.3. The implementation relies heavily on the UFL `diff` operator.

The generality of the material model base class allows for the (almost trivial) specification of a large set of models. To see this in practice, let us consider two popular material models,

- the *St. Venant–Kirchhoff* model: $\psi_{\text{SVK}} = \frac{\lambda}{2}\text{Tr}(E)^2 + \mu\text{Tr}(E^2)$, and
- the two term *Mooney–Rivlin* model: $\psi_{\text{MR}} = c_1(I_1 - 3) + c_2(I_2 - 3)$,

and see how they can be specified in CBC.Twist. The relevant blocks of code are shown in Figures 27.4 and 27.5. Clearly, the code simply contains the strain energy function in classical notation, along with some metadata clarifying the number of material parameters and the strain measure the model relies on. The file `material_models.py` contains several other material models, including *linear elasticity,*

Python code

```
class StVenantKirchhoff(MaterialModel)

    def model_info(self):
        self.num_parameters = 2
        self.kinematic_measure = "GreenLagrangeStrain"

    def strain_energy(self, parameters):
        E = self.E
        [mu, lmbda] = parameters
        return lmbda/2*(tr(E)**2) + mu*tr(E*E)
```

Figure 27.4: Definition the strain energy function for a St. Venant–Kirchhoff material.

Python code

```
class MooneyRivlin(MaterialModel)

    def model_info(self):
        self.num_parameters = 2
        self.kinematic_measure = "CauchyGreenInvariants"

    def strain_energy(self, parameters):
        I1 = self.I1
        I2 = self.I2
        [C1, C2] = parameters
        return C1*(I1 - 3) + C2*(I2 - 3)
```

Figure 27.5: Definition the strain energy function for a two term Mooney–Rivlin material.

neo Hookean, *Isihara*, *Biderman*, and *Gent–Thomas* that come pre-implemented in CBC.Twist. (Refer to the article by Marckmann and Verron (2006) comparing several hyperelastic models for rubber-like materials for their definitions.) But the salient point to note here is that it is straightforward to introduce other additional models, and this is a significant feature of CBC.Twist.

27.2 Numerical methods and further implementation details

In the preceding section, we saw the functionality that CBC.Twist provided to easily specify material models to suitably characterize different materials of interest. In this section, we return to the general form of the balance of linear momentum and look at details of a finite element formulation and implementation for this equation. For further details on the treatment that follows, the interested reader is directed to Simo and Hughes (1998).

27.2.1 The finite element formulation of the balance of linear momentum

By taking the dot product of (27.2) with a test function $v \in \hat{\mathrm{V}}$ and integrating over the reference domain and time, we have

$$\int_0^T \int_\Omega \rho \frac{\partial^2 u}{\partial t^2} \cdot v \, \mathrm{d}x \, \mathrm{d}t \; = \int_0^T \int_\Omega \mathrm{Div}(P) \cdot v \, \mathrm{d}x \, \mathrm{d}t \; + \int_0^T \int_\Omega B \cdot v \, \mathrm{d}x \, \mathrm{d}t. \tag{27.5}$$

Noting that the traction vector $T = PN$ on $\partial\Omega_{\mathrm{N}}$ (N being the outward normal on the boundary) and that by definition $v|_{\partial\Omega_{\mathrm{D}}} = 0$, we apply the divergence theorem to arrive at the following weak form of the balance of linear momentum:

Find $u \in \mathrm{V}$, such that $\forall\, v \in \hat{\mathrm{V}}$:

$$\int_0^T \int_\Omega \rho \frac{\partial^2 u}{\partial t^2} \cdot v \,\mathrm{d}x \,\mathrm{d}t \; + \int_0^T \int_\Omega P : \mathrm{Grad}(v) \,\mathrm{d}x \,\mathrm{d}t \; = \int_0^T \int_\Omega B \cdot v \,\mathrm{d}x \,\mathrm{d}t \; + \int_0^T \int_{\partial\Omega_\mathrm{N}} T \cdot v \,\mathrm{d}s \,\mathrm{d}t, \quad (27.6)$$

with initial conditions $u(X,0) = u_0(X)$ and $\frac{\partial u}{\partial t}(X,0) = v_0(X)$ in Ω, and boundary conditions $u(X,t) = g(X,t)$ on $\partial\Omega_\mathrm{D}$.

The finite element formulation implemented in CBC.Twist follows the Galerkin approximation of the above weak form (27.6), by looking for solutions in a finite solution space $\mathrm{V}_h \subset \mathrm{V}$ and allowing for test functions in a finite approximation of the test space $\hat{\mathrm{V}}_h \subset \hat{\mathrm{V}}$.[2]

27.2.2 Implementation of the static form

We consider first the static weak form (dropping the time derivative term) of the balance of linear momentum which reads

$$\int_\Omega P : \mathrm{Grad}(v) \,\mathrm{d}x - \int_\Omega B \cdot v \,\mathrm{d}x - \int_{\partial\Omega_\mathrm{N}} T \cdot v \,\mathrm{d}s = 0. \quad (27.7)$$

Since CBC.Twist provides the necessary functionality to easily compute the first Piola–Kirchhoff stress tensor, P, given a displacement field, u, for arbitrary material models, (27.7) is just a nonlinear functional in terms of u. The automatic differentiation capabilities of UFL[3] make this nonlinear form straightforward to implement, as evidenced by the code listing in Figure 27.6.

This listing provides the relevant section of the static balance of linear momentum solver class, `StaticMomentumBalanceSolver`. The class draws information about the problem (mesh, loading, boundary conditions and form of the stress equation derived from the material model) from the user-specified problem class,[4] and solves the nonlinear momentum balance equation using a Newton solver.

27.2.3 Time-stepping algorithms

CBC.Twist implements two time integration algorithms to solve the weak form of the fully dynamic balance of linear momentum (27.6). The first of these is the so-called CG_1 method (Eriksson et al., 1996). In order to derive this method, (27.6), which is a second order differential equation in time, is rewritten as a system of first order equations. We do this by introducing an additional velocity variable, $w = \frac{\partial u}{\partial t}$. Thus, the weak form now reads:

[2] We now note an inherent advantage in choosing the Lagrangian description in formulating the theory. The fact that the integrals in (27.6), along with the various fields and differential operators, are defined over the fixed domain Ω means that one need not be concerned with the complexity associated with calculations on a moving computational domain when implementing this formulation.

[3] An earlier chapter on UFL (17) provides a detailed look at the capabilities of UFL, as well as insights into how it achieves its functionality. Even so, we note the following differentiation capabilities of UFL because of their pivotal relevance to this work:

- Computing spatial derivatives of fields, which allows for the construction of differential operators such as such as $\mathrm{Grad}(\cdot)$ or $\mathrm{Div}(\cdot)$:
  ```
  df_i = Dx(f, i)
  ```
- Differentiating arbitrary expressions with respect to variables they are functions of:
  ```
  g = variable(cos(cell.x[0]))
  f = exp(g**2)
  h = diff(f, g)
  ```
- Differentiating forms with respect to coefficients of a discrete function, allowing for automatic linearizations of nonlinear variational forms:
  ```
  a = derivative(L, w, u)
  ```

[4] Details of how the user can specify problem details are covered in the following section containing examples of CBC.Twist usage.

Python code

```
# Get the problem mesh
mesh = problem.mesh()

# Define the function space
vector = VectorFunctionSpace(mesh, "Lagrange", 1)

# Test and trial functions
v = TestFunction(vector)
u = Function(vector)
du = TrialFunction(vector)

# Get forces and boundary conditions
B = problem.body_force()
PN = problem.surface_traction()
bcu = problem.boundary_conditions()

# First Piola-Kirchhoff stress tensor based on
# the material model
P = problem.first_pk_stress(u)

# The variational form corresponding to static
# hyperelasticity
L = inner(P, Grad(v))*dx - inner(B, v)*dx - inner(PN, v)*ds
a = derivative(L, u, du)

# Setup and solve problem
problem = NonlinearVariationalProblem(L, u, bcu, a)
solver = NonlinearVariationalSolver(problem)
solver.solve()
```

Figure 27.6: The relevant section of the class `StaticMomentumBalanceSolver`, the solver for the static balance of linear momentum.

Find $(u, w) \in \mathrm{V}$, such that $\forall\, (v, r) \in \hat{\mathrm{V}}$:

$$\begin{aligned} \int_0^T \int_\Omega \rho \frac{\partial w}{\partial t} \cdot v \,\mathrm{d}x \,\mathrm{d}t \; + \int_0^T \int_\Omega P : \mathrm{Grad}(v) \,\mathrm{d}x \,\mathrm{d}t \; &= \int_0^T \int_\Omega B \cdot v \,\mathrm{d}x \,\mathrm{d}t \; + \int_0^T \int_{\partial\Omega_\mathrm{N}} T \cdot v \,\mathrm{d}s \,\mathrm{d}t, \text{ and} \\ \int_0^T \int_\Omega \frac{\partial u}{\partial t} \cdot r \,\mathrm{d}x \,\mathrm{d}t \; &= \int_0^T \int_\Omega w \cdot r \,\mathrm{d}x \,\mathrm{d}t. \end{aligned} \tag{27.8}$$

with initial conditions $u(X, 0) = u_0(X)$ and $w(X, 0) = v_0(X)$ in Ω, and boundary conditions $u(X, t) = g(X, t)$ on $\partial\Omega_\mathrm{D}$.

We now assume that the finite element approximation space V_h is CG_1 (continuous and piecewise linear in time), and $\hat{\mathrm{V}}_h$ is DG_0 (discontinuous and piecewise constant in time). With these assumptions, we arrive at the following scheme:

$$\begin{aligned} \int_\Omega \rho \frac{(w_{n+1} - w_n)}{\Delta t} \cdot v \,\mathrm{d}x \; + \int_\Omega P(u_\mathrm{mid}) : \mathrm{Grad}(v) \,\mathrm{d}x \; &= \int_\Omega B \cdot v \,\mathrm{d}x \; + \int_{\partial\Omega_\mathrm{N}} T \cdot v \,\mathrm{d}s, \text{ and} \\ \int_\Omega \frac{(u_{n+1} - u_n)}{\Delta t} \cdot r \,\mathrm{d}x \; &= \int_\Omega w_\mathrm{mid} \cdot r \,\mathrm{d}x, \end{aligned} \tag{27.9}$$

where $(\cdot)_n$ and $(\cdot)_{n+1}$ are the values of a quantity at the current and subsequent time-step, respectively, and $(\cdot)_\mathrm{mid} = \frac{(\cdot)_n + (\cdot)_{n+1}}{2}$. A section of the CG_1 linear momentum balance solver class is presented in Figure 27.7. The code closely mirrors the scheme defined in (27.9), and results in a mixed system that is solved for using a Newton scheme.

Figure 27.7: Relevant portion of the dynamic balance of linear momentum solver using the CG_1 time-stepping scheme.

Python code

```
class CG1MomentumBalanceSolver(CBCSolver):

    # Define function spaces
    vector = VectorFunctionSpace(mesh, "Lagrange", 1)
    mixed_element = MixedFunctionSpace([vector,
                                        vector])
    V = TestFunction(mixed_element)
    dU = TrialFunction(mixed_element)
    U = Function(mixed_element)
    U0 = Function(mixed_element)

    # Get initial conditions, boundary conditions
    # and body forces
    ...

    # Functions
    v, r = split(V)
    u, w = split(U)
    u0, w0 = split(U0)

    # Evaluate displacements and velocities at
    # mid points
    u_mid = 0.5*(u0 + u)
    w_mid = 0.5*(w0 + w)

    # Get reference density
    rho = problem.reference_density()

    # Piola-Kirchhoff stress tensor based on the
    # material model
    P = problem.first_pk_stress(u_mid)

    # The variational form corresponding to
    # dynamic hyperelasticity
    L = rho*inner(w - w0, v)*dx \
        + dt*inner(P, grad(v))*dx \
        - dt*inner(B, v)*dx\
        + inner(u - u0, r)*dx \
        - dt*inner(w_mid, r)*dx

    # Add contributions to the form from the
    # Neumann boundary conditions
    ...

    a = derivative(L, U, dU)
```

The CG_1 scheme defined in (27.9) is straightforward to derive and implement, and it is second order accurate and energy conserving.[5] But it should also be noted that the mixed system that results from the formulation is computationally expensive and memory intensive, as the number of variables being solved for have doubled due to the introduction of the velocity variable.

CBC.Twist also provides a standard implementation of a finite difference time-stepping algorithm that is commonly used in the computational mechanics community: the Hilber–Hughes–Taylor (HHT) method (Hilber et al., 1977). The stability and dissipative properties of this method in the case of linear problems have been thoroughly discussed in Hughes (1987). In particular, the method contains three parameters α, β and γ which control the accuracy, stability and numerical dissipation of the scheme. The default values for these parameters chosen in CBC.Twist ($\alpha = 1$, $\beta = \frac{1}{4}$ and $\gamma = \frac{1}{2}$) ensure that the method is second order accurate, stable for linear problems and introduces no numerical dissipation.

The method is briefly sketched below. For further details about the scheme itself, or its implementation in CBC.Twist, the interested reader is directed to the previously mentioned papers, and the `MomentumBalanceSolver` class in the file `solution_algorithms.py`.

Given initial conditions $u(X,0) = u_0(X)$ and $\frac{\partial u}{\partial t}(X,0) = v_0(X)$, we can compute the initial acceleration, a_0, from the weak form:

$$\int_\Omega \rho a_0 \cdot v \,\mathrm{d}x \; + \int_\Omega P(u_0) : \mathrm{Grad}(v) \,\mathrm{d}x \; - \int_\Omega B(X,0) \cdot v \,\mathrm{d}x \; - \int_{\partial\Omega_\mathrm{N}} T(X,0) \cdot v \,\mathrm{d}s = 0. \tag{27.10}$$

This provides the complete initial state (u_0, v_0, a_0) of the body. Now, given the solution at time step n, the HHT formulae advance the solution to step $n+1$ as follows. First, we note the following definitions:

$$\begin{aligned} u_{n+1} &= u_n + \Delta t v_n + \Delta t^2 \left[\left(\frac{1}{2} - \beta \right) a_n + \beta a_{n+1} \right] \\ v_{n+1} &= v_n + \Delta t \left[(1-\gamma) a_n + \gamma a_{n+1} \right] \\ u_{n+\alpha} &= (1-\alpha) u_n + \alpha u_{n+1} \\ v_{n+\alpha} &= (1-\alpha) v_n + \alpha v_{n+1} \\ t_{n+\alpha} &= (1-\alpha) t_n + \alpha t_{n+1} \end{aligned} \tag{27.11}$$

Inserting the definitions in (27.11) into the following form of the balance of linear momentum,

$$\int_\Omega \rho a_{n+1} \cdot v \,\mathrm{d}x \; + \int_\Omega P(u_{n+\alpha}) : \mathrm{Grad}(v) \,\mathrm{d}x \; - \int_\Omega B(X, t_{n+\alpha}) \cdot v \,\mathrm{d}x \; - \int_{\partial\Omega_\mathrm{N}} T(X, t_{n+\alpha}) \cdot v \,\mathrm{d}s = 0, \tag{27.12}$$

we can solve for the for the only unknown variable, the acceleration at the next step, a_{n+1}. The acceleration solution to (27.12) is then used in the definitions (27.11) to update to new displacement and velocity values, and the problem is stepped through time.

We close this subsection on time-stepping algorithms with one usage detail pertaining to CBC.Twist. By default, when solving a dynamics problem, CBC.Twist assumes that the user wants to use the HHT method. In case one wants to override this behavior, they can do so by returning `"CG(1)"` in the `time_stepping` method while specifying the problem. Figure 27.11 is an example showing this.

27.3 Examples of CBC.Twist usage

The algorithms discussed thus far serve primarily to explain the computational framework's inner working, and are not at the level at which the user usually interacts with CBC.Twist (unless they

[5] This is demonstrated in Figure 27.13 as part of the second example calculation.

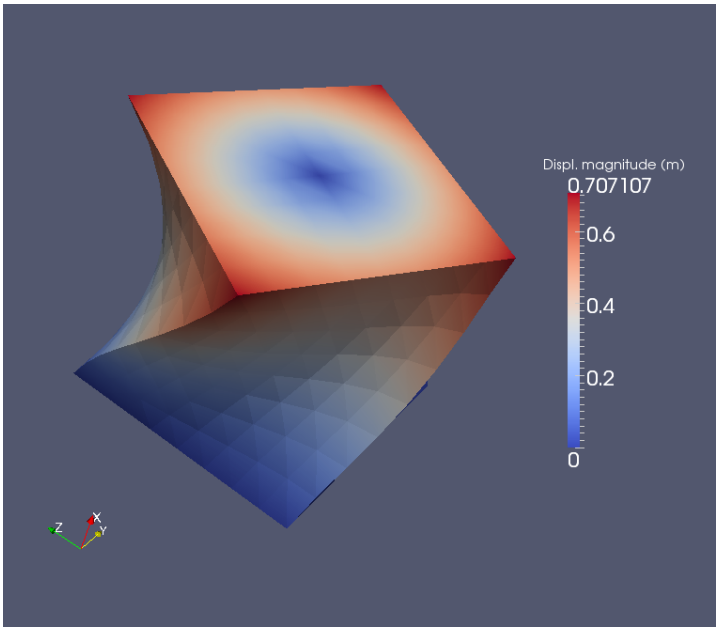


Figure 27.8: A hyperelastic cube twisted by 60 degrees.

are interested in extending it). In practice, the functionality of CBC.Twist is exposed to the user through two primary problem definition classes: `StaticHyperelasticity` and `Hyperelasticity`. These classes reside in `problem_definitions.py`, and contain numerous methods for defining aspects of the nonlinear elasticity problem. As their names suggest, these are respectively used to describe static or dynamic problems in nonlinear elasticity.

Over the course of the following examples, we will see how various problems can be defined in CBC.Twist by suitably deriving from these problem classes and overloading relevant methods.[6] We will also see some results from these calculations. The information defined in the problem classes are internally transferred to the solvers described earlier to actually solve the problem.

27.3.1 The static twisting of a hyperelastic cube

The first problem we are interested in is the twisting of a unit hyperelastic cube (1 m^3). The cube is assumed to be made out of a St. Venant–Kirchhoff material with Lamé's parameters $\mu = 3.8461$ N/m^2 and a spatially varying $\lambda = 5.8x_1 + 5.7(1 - x_1)$ N/m^2. Here, x_1 is the first coordinate of the reference position, X.[7] In order to twist the cube, the face $x_1 = 0$ is held fixed and the opposite face $x_1 = 1$ is rotated 60 degrees using the Dirichlet condition defined in Figure 27.9.

Before getting to the actual specification of the problem in code, we need to import CBC.Twist's functionality:

Python code

```
from cbc.twist import *
```

The problem is completely specified by defining relevant methods in the user-created class `Twist` (see Figure 27.9), which derives from the base class `StaticHyperelasticity`. CBC.Twist only requires relevant methods to be provided, and for the current problem, this includes those that define the

[6] The examples presented in this chapter, along with a few others, reside in the `demos/twist/` folder in CBC.Twist's source repository. They can be run by navigating to this folder and typing `python demo_name.py` on the command-line.

[7] The numerical parameters in this chapter have been arbitrarily chosen for illustration of the framework's use. They do not necessarily correspond to a real material.

Python code

```
class Twist(StaticHyperelasticity):

    def mesh(self):
        n = 8
        return UnitCube(n, n, n)

    def dirichlet_conditions(self):
        clamp = Expression(("0.0", "0.0", "0.0"))
        twist = Expression(("0.0",
        "y0+(x[1]-y0)*cos(theta)-(x[2]-z0)*sin(theta)-x[1]",
        "z0+(x[1]-y0)*sin(theta)+(x[2]-z0)*cos(theta)-x[2]"),
        y0=0.5, z0=0.5, theta=pi/3)
        return [clamp, twist]

    def dirichlet_boundaries(self):
        return ["x[0] == 0.0", "x[0] == 1.0"]

    def material_model(self):
        mu    = 3.8461
        lmbda = Expression("x[0]*5.8+(1-x[0])*5.7")

        material = StVenantKirchhoff([mu, lmbda])
        return material

    def __str__(self):
        return "A cube twisted by 60 degrees"
```

Figure 27.9: Problem definition: The static twisting of a hyperelastic cube.

Python code

```
twist = Twist()
u = twist.solve()
```

Figure 27.10: Solving the posed problem.

computational domain, Dirichlet boundary conditions and material model. The methods are fairly self-explanatory, but the following points are to be noted. Firstly, CBC.Twist supports spatially-varying material parameters. Secondly, Dirichlet boundary conditions are posed in two parts: the conditions themselves, and the corresponding boundaries along which they act.

In order to solve this problem, an instance of the `Twist` class is created and its `solve` method is called (see Figure 27.10). This triggers a Newton solve which exhibits quadratic convergence (see Table 27.2) and results in the displacement field shown in Figure 27.8.

27.3.2 The dynamic release of a twisted cube

In this problem, we release a unit cube (1 m^3) that has previously been twisted. The initial twist was precomputed in a separate calculation involving a traction force on the top surface and the resulting displacement field was stored in the file `twisty.txt`. The release calculation loads this solution as the initial displacement. It fixes the cube (made of a St. Venant–Kirchhoff material with Lamé's parameters $\mu = 3.8461$ N/m^2 and $\lambda = 5.76$ N/m^2) on the bottom surface, and tracks the motion of the cube over 2 s.

The problem is specified in the user-created class `Release`, which derives from `Hyperelasticity`. This example is similar to the previous one, except that since it is a dynamic calculation, it also provides initial conditions, a reference density and information about time-stepping. Again, the methods listed in Figure 27.11 are straightforward, and the only additional point to note is that

Iteration	Relative Residual Norm
1	5.835e-01
2	1.535e-01
3	3.640e-02
4	1.004e-02
5	1.117e-03
6	1.996e-05
7	9.935e-09
8	3.844e-15

Table 27.2: Quadratic convergence of the Newton method used to solve the hyperelasticity problem. It is interesting to note that this convergence is obtained even though the 60 degree twist condition was imposed in a single step.

CBC.Twist provides some convenience utilities to simplify the specification of the problem. For example, one can load initial conditions directly from files, and it allows for the specification of boundaries purely as conditional strings.

When `Release` is instantiated and its `solve` method is called, we see the relaxation of the pre-twisted cube. After initial unwinding of the twist, the body proceeds to twist in the opposite direction due to inertia. This process repeats itself, and snapshots of the displacement over the first 0.5 s are shown in Figure 27.12. Figure 27.13 highlights the energy conservation of the CG_1 numerical scheme used to time-step this problem by totaling the kinetic energy and potential energy of the body over the course of the calculation. CBC.Twist provides this information through the methods `kinetic_energy(v)` and `potential_energy(u)`, where `v` and `u` are the discrete velocity and displacement fields respectively.

27.3.3 A hyperelastic dolphin tumbling through a "flow"

In this final example, we aim to crudely simulate the motion of a dolphin under a flow field. The dolphin is assumed to be made out of a Mooney–Rivlin material ($c_1 = 6.169$ N/m^2, $c_2 = 10.15$ N/m^2), and the flow field is simply modeled by a uniform traction force $T = (0.05, 0)$ N acting everywhere on the surface of the dolphin, pushing it to the right.

This example is constructed to exhibit some additional features of CBC.Twist. For one, CBC.Twist is capable of performing dynamic calculations under entirely Neumann boundary conditions. In addition, this calculation points out that the framework can seamlessly handle problems in two dimensions as well.

The problem is specified in the user-created class `FishyFlow` derived from `Hyperelasticity`. There is nothing new to note in the code listing for this problem (Figure 27.14), other than the fact that we now specify Neumann boundary conditions. The specification listing is not very long because CBC.Twist assumes meaningful default values for unspecified information.

To demonstrate one final piece of functionality of CBC.Twist, we don't solve the problem in the same manner as we did the first two examples; that is, we do not instantiate an object of class `FishyFlow` and call its `solve` method. Instead, we set up our own time loop and manually step through time using the `step` method. This is shown in Figure 27.15.

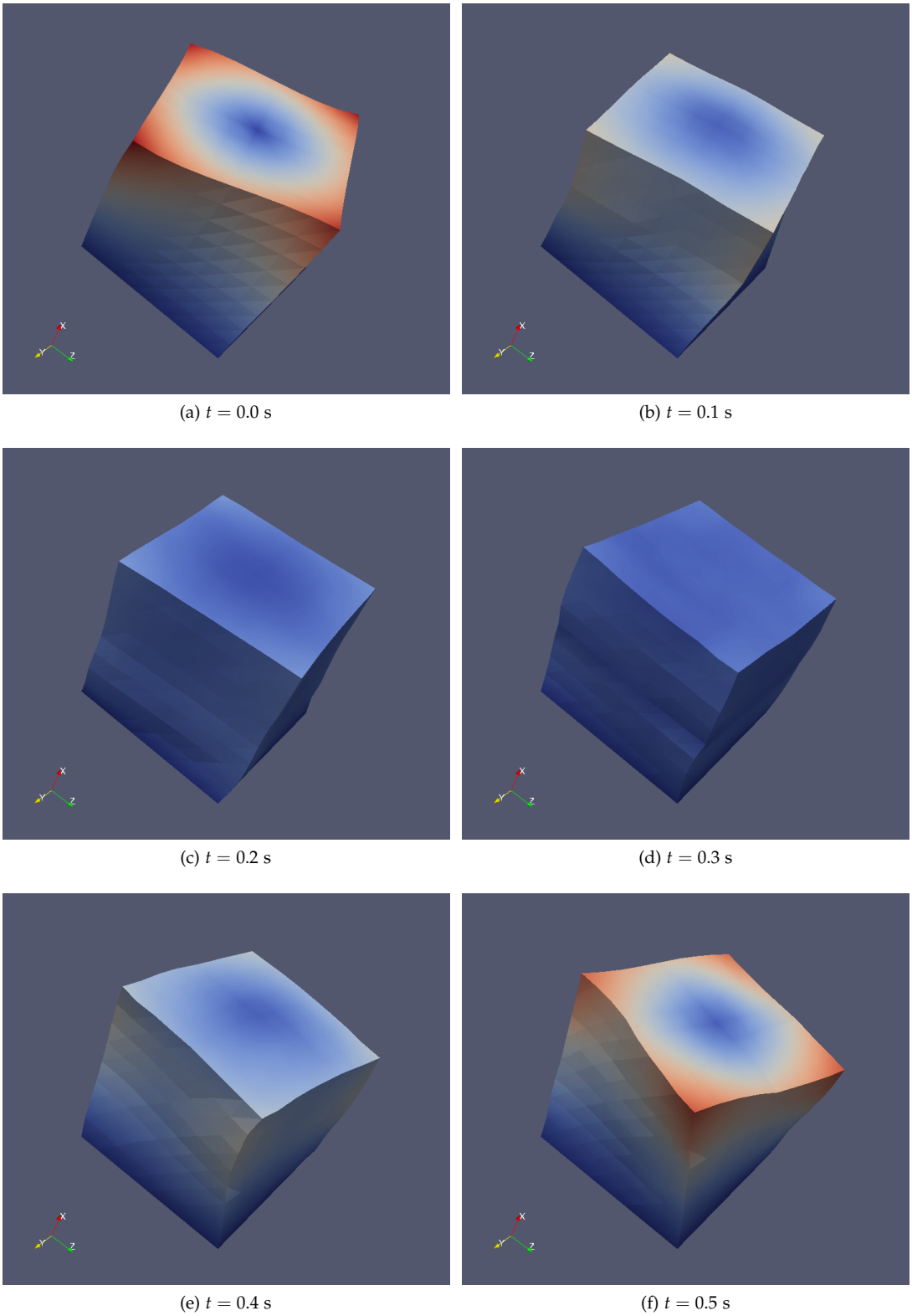
The advantage of solving the problem in this manner is that, now, one has more control over calculations in CBC.Twist. For example, rather than just fixing a traction force on the surface of the dolphin to mimic the effect of flow field, one can instead solve at each time step an actual flow field and use it to correctly drive the solid mechanics. This functionality of CBC.Twist is used in a following

Python code

```
class Release(Hyperelasticity):

    def mesh(self):
        n = 8
        return UnitCube(n, n, n)

    def end_time(self):
        return 2.0

    def time_step(self):
        return 2.e-3

    def time_stepping(self):
        return "CG(1)"

    def reference_density(self):
        return 1.0

    def initial_conditions(self):
        u0 = "twisty.txt"
        v0 = Expression(("0.0", "0.0", "0.0"))
        return u0, v0

    def dirichlet_values(self):
        return [(0, 0, 0)]

    def dirichlet_boundaries(self):
        return ["x[0] == 0.0"]

    def material_model(self):
        mu    = 3.8461
        lmbda = 5.76
        material = StVenantKirchhoff([mu, lmbda])
        return material

    def __str__(self):
        return "A pretwisted cube being released"
```

Figure 27.11: Problem definition: The dynamic release of a twisted cube.

Figure 27.12: Relaxation and re-twisting of a released cube over the first 0.5 s of the calculation.



(a) $t = 0.0$ s (b) $t = 0.1$ s

(c) $t = 0.2$ s (d) $t = 0.3$ s

(e) $t = 0.4$ s (f) $t = 0.5$ s

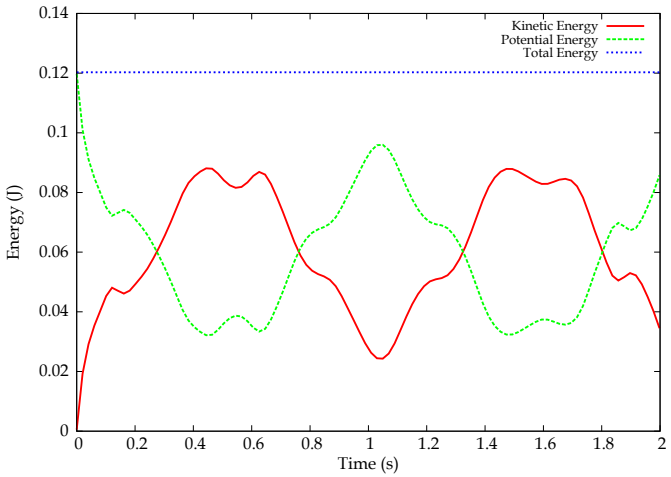


Figure 27.13: Over the course of the computation, the energy in the body is converted between potential and kinetic energy, but the total remains constant.

Python code

```
class FishyFlow(Hyperelasticity):

    def mesh(self):
        mesh = Mesh("dolphin.xml.gz")
        return mesh

    def end_time(self):
        return 10.0

    def time_step(self):
        return 0.1

    def neumann_conditions(self):
        flow_push = Expression(("force", "0.0"), force=0.05)
        return [flow_push]

    def neumann_boundaries(self):
        everywhere = "on_boundary"
        return [everywhere]

    def material_model(self):

        material = MooneyRivlin([6.169, 10.15])
        return material
```

Figure 27.14: Problem definition: A hyperelastic dolphin being pushed to the right.

Figure 27.15: Stepping through time in an external time loop. `step` steps the problem forward by one time step, and `update` updates the values of all time-dependent variables to the current time.

Python code

```
problem = FishyFlow()

dt = problem.time_step()
T = problem.end_time()

t = dt
while t <= T:
    problem.step(dt)
    problem.update()
    t = t + dt
```

chapter on adaptive methods for fluid–structure interaction (29). In that work, the fluid–structure problem is solved using a staggered approach with the solid mechanics equation being solved by CBC.Twist. An external time loop similar to the one in Figure 27.15 is set up to individually step through the fluid problem, the solid problem and a mesh equation; a process which is iterated until convergence is reached at each time step. This process involves the systematic transfer of relevant information (such as fluid loading) from other problems to CBC.Twist.

But returning to our current example, Figure 27.16 shows time snapshots of the motion of the dolphin over the course of the computation. Notice that the fish deforms elastically as it tumbles toward the right.

27.4 Conclusions

This chapter presented an overview of CBC.Twist, an automated computational framework for nonlinear elasticity. Beginning with elements of classical nonlinear elasticity theory to motivate its design, the discourse took a closer look at the algorithms underlying CBC.Twist's implementation. The chapter concluded with some examples, offering a tutorial-like description of how the framework can be used in practice to solve problems.

The discussion aimed to highlight a central feature of CBC.Twist: the ease with which different material models can be defined and used. This feature makes CBC.Twist immediately applicable to a number of real-world problems in engineering, especially those pertaining to polymer and biological tissue mechanics.

CBC.Twist is a collaboratively developed open source project (released under the GNU GPL) that is freely available from its source repository at `https://launchpad.net/cbc.solve/`. Its only dependency is a working FEniCS installation. CBC.Twist is released with the goal that it will allow users to easily solve problems in nonlinear elasticity as part of answering specific questions through computational modeling. Everyone is encouraged to fetch and try it. Users are also encouraged to modify the code to better suit their own purposes, and contribute changes that they think are useful to the community. Along these lines, some possible ideas for extending the framework include:

- Implementing other specific material models
- Allowing for bodies composed of multiple materials
- Support for (nearly) incompressible materials
- Support for anisotropic materials
- Support for viscoelastic materials
- Goal-oriented adaptivity

Contributions toward these (or other useful) extensions are welcome.

Figure 27.16: The motion of a hyperelastic dolphin being forced to the right. Careful observation of the tail fin shows deformation of the dolphin in addition to its overall motion toward the right.

(a) $t = 0.0$ s (b) $t = 0.125$ s (c) $t = 0.250$ s

(d) $t = 0.375$ s (e) $t = 0.500$ s (f) $t = 0.625$ s

(g) $t = 0.750$ s (h) $t = 0.875$ s (i) $t = 1.000$ s

28 *Turbulent flow and fluid–structure interaction*

By Johan Hoffman, Johan Jansson, Niclas Jansson, Claes Johnson and Rodrigo Vilela De Abreu

The FEniCS Project aims towards the goals of generality, efficiency, and simplicity, concerning mathematical methodology, implementation and application, and the Unicorn project is an implementation aimed at FSI and high Re turbulent flow guided by these principles. Unicorn is based on the DOLFIN/FFC/FIAT suite and the linear algebra package PETSc. We here present some key elements of Unicorn, and a set of computational results from applications. The details of the Unicorn implementation are described in Chapter 18.

28.1 Background

For many problems involving a fluid and a structure, decoupling the computation of the two is not feasible for accurate modeling of the phenomenon at hand. Instead, the full fluid–structure interaction (FSI) problem has to be solved together as a coupled problem. This includes a multitude of important problems in biology, medicine and industry, such as the simulation of insect or bird flight, the human cardiovascular and respiratory systems, the human speech organ, the paper making process, acoustic noise generation in exhaust systems, airplane wing flutter, wind induced vibrations in bridges and wave loads on offshore structures. Common for many of these problems is that for various reasons they are very hard or impossible to investigate experimentally, and thus reliable computational simulation would open up for detailed study and new insights, as well as for new important design tools for construction.

Computational methods for FSI is a very active research field today. In particular, major open challenges of computational FSI include: (i) robustness of the fluid–structure coupling, (ii) for high Reynolds numbers (Re) the computation of turbulent fluid flow, and (iii) efficiency and reliability of the computations in the form of adaptive methods and quantitative error estimation.

28.2 Simulation of high Re turbulent flow

The focus of Unicorn is high Re turbulent fluid flow, also including fluid–structure interaction. Direct numerical simulation (DNS) of turbulent flow is limited to moderate Re and simple geometry, due to the high computational cost of resolving all turbulent scales in the flow. The standard approach in the automotive industry is simulation based on Reynolds averaged Navier–Stokes equations (RANS), where time averages (or statistical averages) are computed to an affordable cost, with the drawback of introducing turbulence models based on parameters that have to be tuned for particular applications.

An alternative to DNS and RANS is Large Eddy Simulation (LES) (Sagaut, 2005), where a filter is applied to the Navier–Stokes equations to derive a new set of equations with a smallest scale given by the filter width, and where the effect of the filter is the introduction of so called subgrid stresses

which need to be modeled in a subgrid model. A subgrid model can be motivated by physics theory or experiments, and the main effect of the subgrid model is to dissipate kinetic energy, for example in the form of turbulent viscosity.

Typically, the numerical method used to approximate the LES equations also introduces dissipation, and there are thus two sources of kinetic energy dissipation: the subgrid model and the numerical method. One class of methods, Implicit LES (ILES), relies completely on the numerical method to act as a subgrid model, without any additional explicit subgrid model (Sagaut, 2005). Turbulence simulation in Unicorn is based on ILES in the form of a stabilized finite element method, referred to as a General Galerkin (G2) simulation (Hoffman and Johnson, 2007), where a least squares stabilization based on the residual of the Navier–Stokes equations acts as an ILES subgrid model.

In the current G2/ILES implementation of Unicorn, continuous piecewise linear approximation is used in space and time, together with a least squares stabilization based on the residual; see Hoffman and Johnson (2007) for details.

28.3 Turbulent boundary layers

The choice of boundary conditions at a solid wall is critical for accurate LES modeling of fluid flow, in particular to capture flow separation phenomena. In Unicorn, laminar boundary layers are fully resolved by the computational method by applying no slip (zero velocity) boundary conditions at the wall. On the other hand, computational resolution of turbulent boundary layers is only possible at limited Reynolds numbers and for simple geometries.

The standard way to model the effect of turbulent boundary layers is to divide the computational domain into: (i) an interior part and (ii) a boundary layer region. In the boundary layer, a simplified model of the flow is used to provide boundary conditions to the LES equations to be solved in the interior part. Boundary conditions are typically in the form of a wall shear stress, and the coupling between (i) and (ii) may be one-way from (ii) to (i), or more closely coupled. Wall shear stress models are developed based on experimental data, theory or computation by a simplified model such as, e.g., a RANS model. For an overview of boundary layer modeling, see Sagaut et al. (2006); Piomelli and Balaras (2002).

The wall shear stress model in Unicorn takes a similar form as the simple Schumann model (Schumann, 1975), with the tangential velocity proportional to the local wall shear stress through a skin friction parameter (or function) β. The following boundary conditions are used for the velocity u and stress σ:

$$u \cdot n = 0, \tag{28.1}$$

$$\beta u \cdot \tau_k + (\sigma n) \cdot \tau_k = 0, \quad k = 1, 2, \tag{28.2}$$

for $(x, t) \in \Gamma_{solid} \times [0, T]$, with $n = n(x)$ an outward unit normal vector, and $\tau_k = \tau_k(x)$ orthogonal unit tangent vectors of Γ_{solid}. The non-penetration boundary condition is applied strongly, whereas a weak implementation is used for the wall shear stress boundary condition.

28.4 Adaptivity and a posteriori error estimation

A posteriori error estimation involves only computable quantities, which opens for adaptive methods with quantitative error control. The basic idea of adaptive algorithms is to optimize the computational method with respect to the goal (output of interest) of the computation. Typical parameters of an adaptive finite element method include the local mesh size (h-adaptivity), local degree of the finite element approximation (p-adaptivity), local shape of the cells (r-adaptivity), or combinations thereof.

Other possible parameters may be the time step size and the stabilization parameters.

In Unicorn, an approach to a posteriori error estimation is used where the error in a chosen output quantity can be estimated in terms of the solution of an associated linearized dual problem. The basic framework is described in the survey articles (Eriksson et al., 1995; Becker and Rannacher, 2001; Giles and Süli, 2002).

Using standard techniques of a posteriori error analysis, an a posteriori error estimate for G2 can be derived in the following form:

$$|\mathcal{M}(u) - \mathcal{M}(U)| \leqslant \sum_T \eta_T, \tag{28.3}$$

where $\mathcal{M}(u)$ is the exact value of the target output quantity and $\mathcal{M}(U)$ is the computed approximation. Furthermore, U is a G2 solution and η_T is a local error indicator for cell T. The error indicator η_T is constructed from the residual, measuring local errors, weighted by the solution to a dual (adjoint) problem measuring the effect of local errors on the output $\mathcal{M}(\cdot)$. The implementation of the dual problem in Unicorn is based on the cG(1)cG(1) method described in Hoffman and Johnson (2007). The computational mesh is then modified according to η_T, by mesh refinement, coarsening or smoothing.

Adaptive G2 methods (also referred to as adaptive DNS/LES) have been used in a number of turbulent flow computations to a very low computational cost where convergence is obtained for output quantities such as drag, lift and pressure coefficients and Strouhal numbers, using orders of magnitude fewer numbers of mesh points than with LES methods based on *ad hoc* refined computational meshes found in the literature (Hoffman, 2005; Hoffman and Johnson, 2006; Hoffman, 2006, 2009; Hoffman and Jansson, 2009; de Abreu et al., 2010).

28.5 Robust fluid–structure coupling

In a computational method based on so called weakly coupled FSI, separate solvers can be used for the fluid and the structure, with the benefit of being able to reuse existing dedicated fluid and structure solvers. To couple the fluid and the structure, boundary data such as displacements and stresses are exchanged over the fluid–structure interface. To make the coupling more robust, the equations for the fluid and the structure can be placed in the same algebraic system together with all coupling conditions, corresponding to a so called strong FSI coupling.

Another approach is to formulate the fluid and the structure as one single continuum model, which is then said to be a monolithic FSI method. The monolithic method used in Unicorn is referred to as Unified Continuum FSI (UC-FSI) (Hoffman et al., 2011), where the fluid and the structure are discretized by the same finite element method over the combined fluid–structure continuum. A velocity formulation is used for the structure to make it consistent with the corresponding fluid equations, and the FSI problem thus takes the form of a multiphase flow problem, with the two phases, the fluid and the structure, described by different constitutive laws. In particular, coupling conditions for displacements and stresses are directly satisfied when using a continuous finite element discretization, which makes UC-FSI very robust.

The computational mesh is made to follow the deformation of the structure, and in the fluid part mesh smoothing is used to optimize the mesh quality, corresponding to an ALE method for the combined fluid–structure continuum. In Unicorn, UC-FSI is implemented based on a continuous piecewise linear approximation in space and time, and a simple streamline diffusion stabilization is used, similar to the method in Hansbo (2000). See Hoffman et al. (2011) for details on the method.

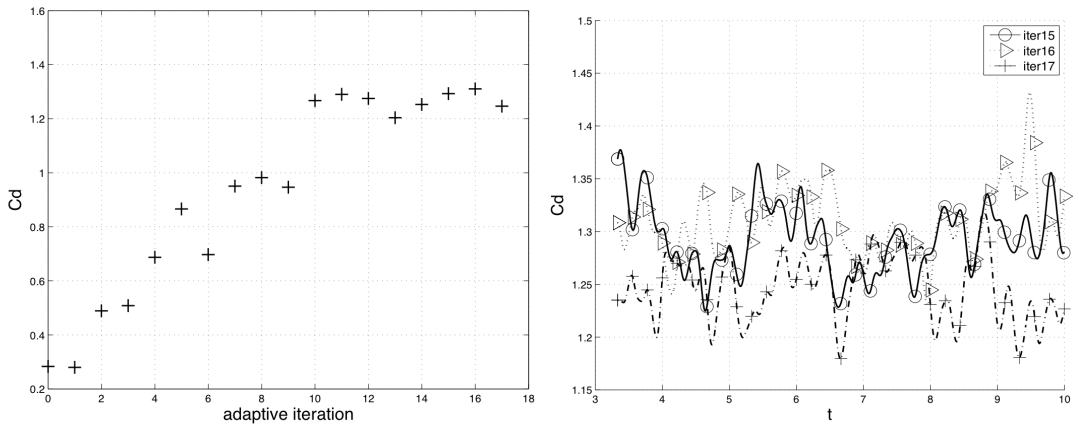


Figure 28.1: Flow around a cube: convergence of the drag coefficient under mesh refinement.

28.6 Applications

In this section, the capability of the Unicorn solver is illustrated by a set of simulation results, connecting to the main focus described above of high Re turbulent flow and robust fluid–structure interaction. Quantitative results are presented for benchmark problems of turbulent flow and FSI, and qualitative results for a turbulent flow FSI problem is presented where no reference data is available. A sensitivity study is also presented, where the skin friction parameter of the boundary layer model is varied to observe the effect on flow separation.

28.6.1 High Re turbulent flow

We first consider the benchmark problem to compute a time average of the drag force on a cube. Starting from a very coarse tetrahedral mesh with 17,952 vertices, the mesh is adaptively refined 17 times, in each iteration marking 10% of the tetrahedrons in the mesh for refinement by bisection. In Figure 28.1, the corresponding drag coefficient is shown as the mesh is refined, converging to a value of $c_D \approx 1.25 \pm 5\%$ over the time interval chosen in the computation. Computing the drag coefficient over a longer time interval would give better confidence in the mean value, at a higher computational cost. In McCormick (1995), an interval of $1.0 - 1.2$ is given for the drag coefficient c_D, although the detailed setup of the underlying experiments is not clear. We note that for this case, when flow separation is given by the sharp corners of the geometry, c_D is considered independent of the specific (high) Reynolds number (McCormick, 1995). We conclude that the Unicorn results are consistent with experimental findings.

In Figure 28.2, snapshots of the solution are shown for the adaptively refined mesh. In Figure 28.3, a snapshot of the gradient of the dual solution is shown which provides sensitivity information with respect to the computation of drag force. Where this gradient is large, the local computational error (local residual weighted by local mesh size) must be made small by refining the mesh, since the error in drag is given by the product of the gradient of the dual solution and local errors. I Figure 28.3, a snapshot of such a local error is shown, where we note that this local error is reduced where the gradient of the dual solution is large but left large in other areas.

Two main features of the adaptive method are: (i) a converged approximation of the drag coefficient c_D (within 5%) is obtained using very few degrees of freedom, and (ii) the mesh is automatically constructed from a coarse mesh, thus bypassing the cost and challenge of ad hoc mesh design. A full discussion of these computations is available in Hoffman et al. (2011a).

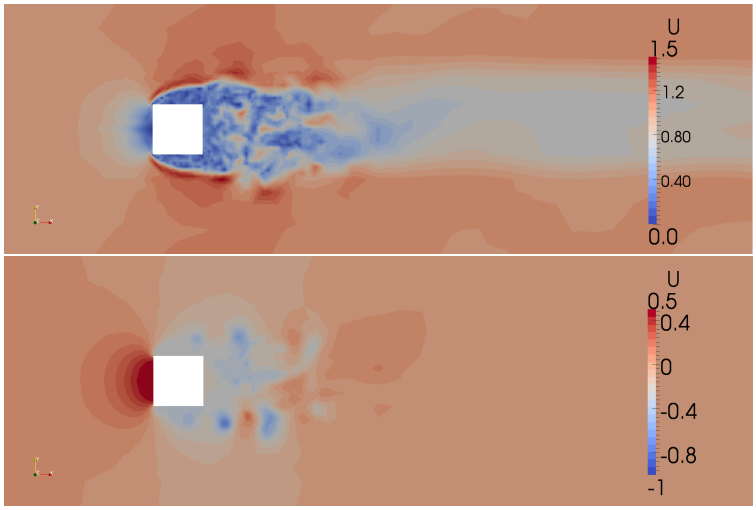


Figure 28.2: Flow around a cube: snapshots of velocity (upper) and pressure (lower) for the finest mesh.

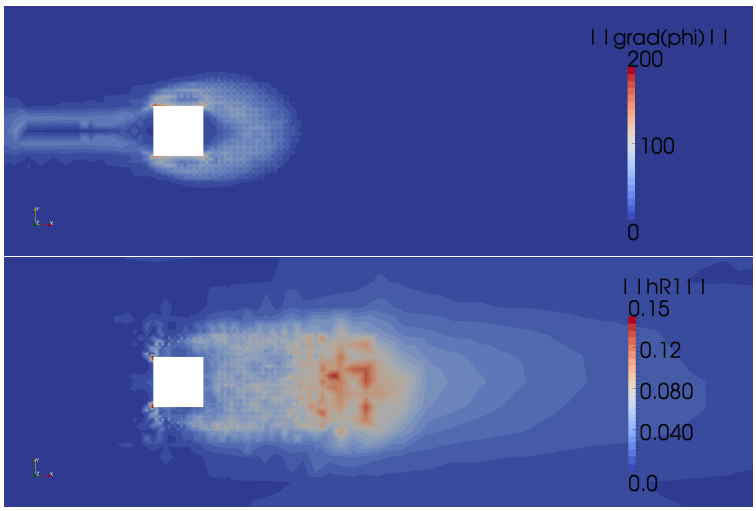


Figure 28.3: Flow around a cube: gradient of dual velocity (upper) and local error (lower), on a mesh after 16 adaptive mesh refinements.

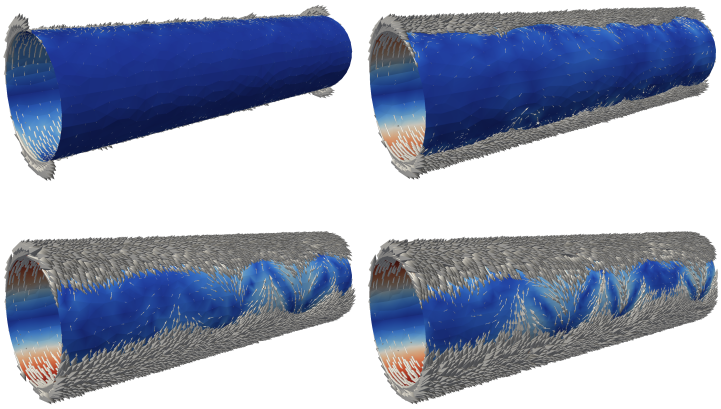


Figure 28.4: Turbulent flow separation (Hoffman and Jansson, 2009): velocity vectors at surface of cylinder; for $\beta = 10^{-1}$, $\beta = 10^{-2}$, $\beta = 10^{-3}$ and $\beta = 0$ (from upper left to bottom right).

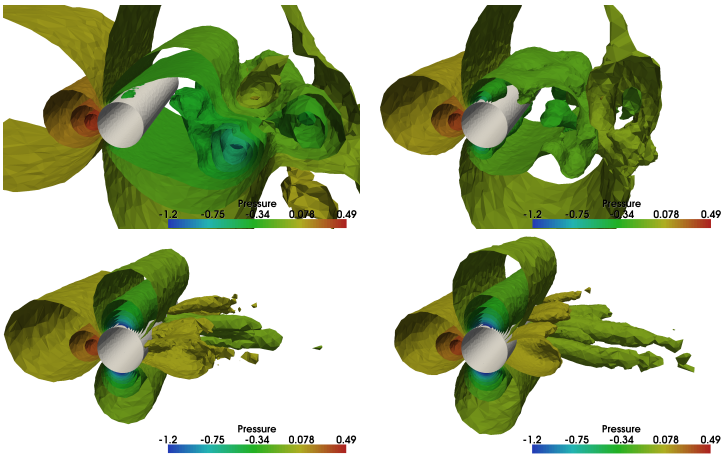


Figure 28.5: Turbulent flow separation (Hoffman and Jansson, 2009): pressure isosurfaces; for $\beta = 10^{-1}$, $\beta = 10^{-2}$, $\beta = 10^{-3}$ and $\beta = 0$ (from upper left to bottom right).

28.6.2 Turbulent flow separation

In Unicorn, the effect of turbulent boundary layers is modeled by a skin friction wall shear stress model, described above. This model has one parameter β, which is related to the skin friction stress. Higher Reynolds number is modeled by a smaller β, based on experimental observation that the skin friction (coefficient) decreases with increasing Re.

To estimate the dependence of the computational result on β, a computational study is carried out using Unicorn, where the drag force of a circular cylinder is computed adaptively based on a posteriori error estimation; see Hoffman and Jansson (2009) for details. In particular, the phenomenon of drag crisis is targeted, characterized by a sudden drop in the non-dimensional drag coefficient for a cylinder for Re increasing beyond a critical size of about 10^5. By decreasing the skin friction parameter β, modeling an increasing Re, the drag crisis scenario is reproduced using Unicorn, in agreement with the high Re experimental data available in the literature (Zdravkovich, 2003). In particular, the drag coefficient drops to a level found in experiments after drag crisis, and 3D so called cell structures develop in the form of streamwise vorticity, also reported in the literature (Zdravkovich, 2003). For vanishing skin friction, the flow approaches a state independent of the skin friction parameter, corresponding to a free slip boundary condition, see Figures 28.4-28.6. Although controversial, this suggests a dominant inviscid separation mechanism independent of the boundary layer, investigated further in Hoffman and Johnson (2008); Hoffman and Jansson (2009).

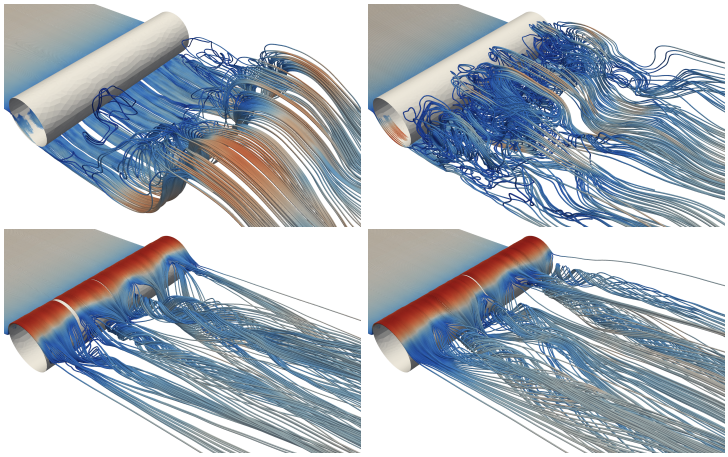


Figure 28.6: Turbulent flow separation (Hoffman and Jansson, 2009): velocity streamlines; for $\beta = 10^{-1}$, $\beta = 10^{-2}$, $\beta = 10^{-3}$ and $\beta = 0$ (from upper left to bottom right).

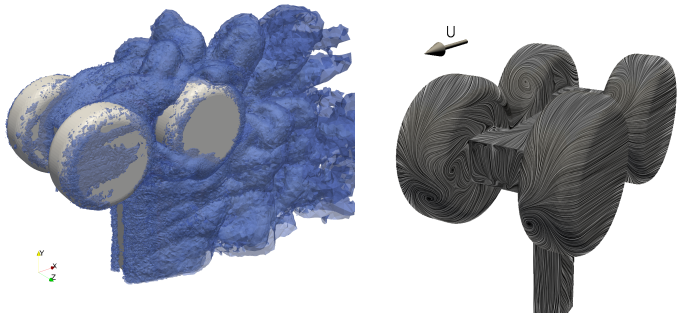


Figure 28.7: Rudimentary landing gear: plot of simulated vorticity (upper) and simulated oil film patterns (lower).

28.6.3 Turbulent flow past complex geometry

With the parallel implementation of Unicorn, realistic problems of complex geometry and high Re turbulent flow can be addressed. As an example, we present simulation results from the NASA/AIAA workshop "Benchmark problems for Airframe Noise Computations" (BANC-1), held in conjunction with the 16^{th} AIAA/CEAS Aeroacoustics Conference in Stockholm in 2010, where Unicorn was used for adaptive simulation of flow past a rudimentary landing gear configuration (de Abreu et al., 2010). The landing gear configuration was designed by Boeing, and detailed experimental results were available, as well as comparison with other participating groups (Spalart and Mejia, 2011).

Starting from a coarse mesh of ca. 70,000 mesh points, the mesh was adaptively refined 7 times with respect to the error in the drag force on the landing gear. The resulting final mesh had 1,000,000 mesh points, and the computation ran on the *Akka* computer at HPC2N using 264 cores. The skin friction boundary layer model was used with $\beta = 0$, thus corresponding to a slip boundary condition.

The Unicorn contribution to the workshop compared well with other participating groups, with overall little spread in the computation of aerodynamic forces, and mean sound pressure levels. Furthermore, mean flow patterns on the surface of the landing gear (see Figure 28.7) showed good agreement with experimental results. Other quantities, such as frequencies, showed a wide spread among the participating groups.

Thanks to the adaptive mesh refinement and the cheap boundary layer model, the Unicorn results were obtained with very few mesh points, less than all other participating groups in the workshop, see Table 28.1.

Team	C_D	#Cells
NTS	1.70	10M
CDA	1.70	29M
KTH	**1.78**	**6M**
KHI	1.81	41M
EXA	1.77	36M
TUB	1.74	11M

Table 28.1: Drag coefficient and number of cells for each of the teams participating in the BANC-1 workshop (Spalart and Mejia, 2011). The Unicorn results are the ones by the KTH team.

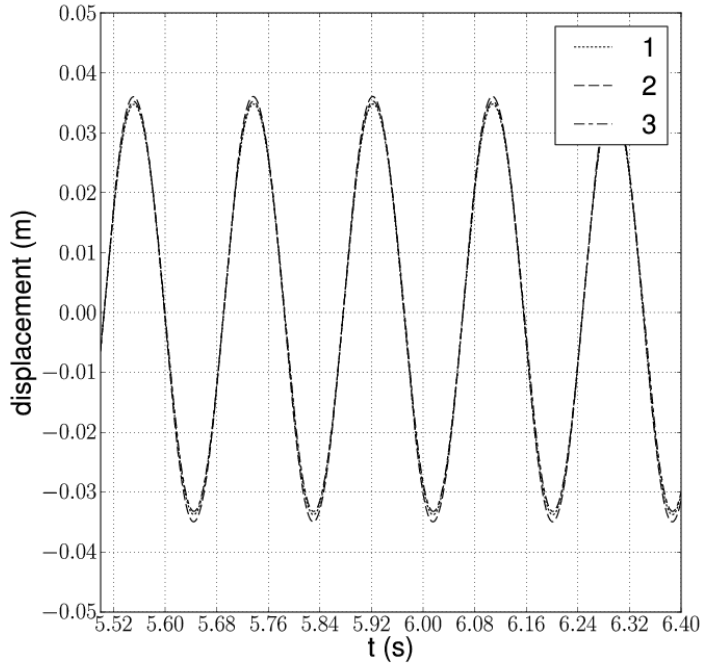


Figure 28.8: The figure shows the displacement along the y-axis of the reference point in the 2D FSI-benchmark problem FLUSTRUK-A, phase aligned to avoid start-up effects, for a sequence of three uniformly refined meshes using Unicorn.

28.6.4 Robust fluid–structure interaction

As a quantitative test of the Unicorn FSI solver, we consider the benchmark problem FLUSTRUK-A, variant 3, which is defined in Hron and Turek (2005). This is a 2D flow in a channel past a fixed cylinder with a thin flexible bar attached to the downstream side of the cylinder. The Unicorn results are compared to the reference results of two other groups: Hron and Turek (2005) and Dunne and Rannacher (2006).

The y-displacement in the oscillation regime varies between 0.0353m and -0.0332m (see Figure 28.8), which is within 1-2% of the Hron/Turek results, and within 11% of the Dunne/Rannacher amplitude. The oscillation frequency is estimated to 5.3Hz from the graph. For comparison, the Hron/Turek frequency is estimated to 5.4Hz and the Dunne/Rannacher frequency is given as 5.48 Hz. See Hoffman et al. (2011) for a full discussion of the results, where also additional basic benchmark results are presented.

Unicorn targets large structure deformations interacting with turbulent fluid flow. In Figure 28.9,

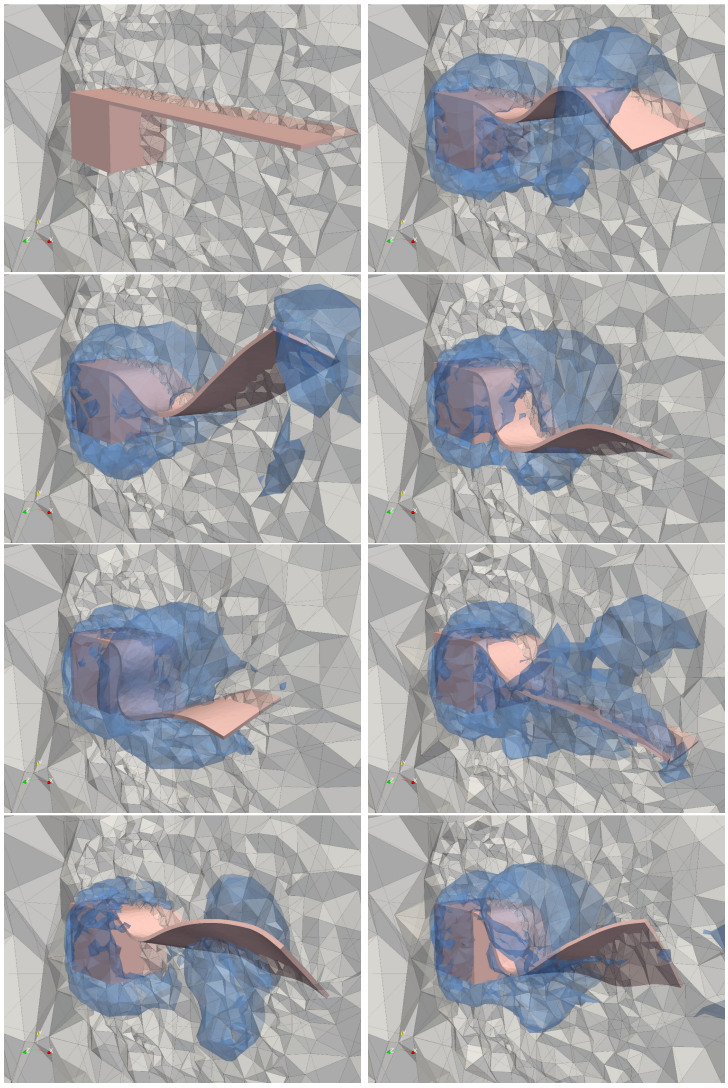


Figure 28.9: Simulation of turbulent flow past a square cylinder with an elastic flag attached downstream (Hoffman et al., 2011): plot of cut of the mesh, isosurface of pressure and fluid–structure phase interface. Going from initial state top left to illustrating violent bending and torsion motion along the long axis of the flag.

we present qualitative results for a model problem of a flexible structure interacting with turbulent flow in 3D, in the form of a fixed cube in turbulent flow with a thin flexible flag mounted in the downstream wake.

We choose an inflow speed of 100 m/s, a cube side length of 1 cm and a flag mounted at the top of the back face of the cube with a length of 0.3 m and a thickness of 5 cm. The viscosity of the fluid is 100 μ Pa s (density $\rho = 1$) which gives a representative Reynold's number of size $\mathrm{Re} = 10^5$. We choose no-slip boundary conditions on the cube and flag with slip boundary conditions on the surrounding channel walls, and a zero pressure outflow condition.

Violent bending and torsion motion in the direction of the long axis of the flag are observed, and we note that the method is robust for these large structure deformations and highly fluctuating flow.

28.7 Acknowledgments

We acknowledge contributions to Unicorn, both software development as well as ideas and scientific support from: Mattias Aechtner, Peter Brune, Zilan Ciftci, Gäetan Compere, Claes Johnson, Ashraful

Kadir, Kaspar Müller, Jeannette Spühler, Michael Stöckli and Rodrigo Vilela de Abreu.

The authors would like to acknowledge the financial support from the European Research Council, Swedish Foundation for Strategic Research, the Swedish Research Council, and the Swedish Energy Agency. The simulations were performed on resources provided by the Swedish National Infrastructure for Computing (SNIC) at High Performance Computing Center North (HPC2N), National Supercomputer Centre in Sweden (NSC) and PDC - Center for High-Performance Computing.

29 An adaptive finite element solver for fluid–structure interaction problems

By Kristoffer Selim

Fluid–structure interaction (FSI) occurs when a fluid interacts with a solid structure, exerting a traction force that causes deformation of the structure and, thus, alters the flow of the fluid itself. The FSI problem is a fully coupled multiphysics problem, whether the problem is solved in a partitioned manner or by a monolithic approach. In many cases we are only interested in one physical output quantity of the fully coupled system, e.g., the displacement of the structure. In order to compute this particular physical output of interest with a high level of accuracy, a goal oriented adaptive finite element method can be used.

This chapter gives a short introduction to goal oriented adaptive finite element approximation for FSI problems and demonstrates how to solve them using FEniCS. We start by formulating an FSI problem and show how it is implemented in FEniCS. We then define and explain the corresponding adaptive algorithm for the FSI problem. For a more comprehensive discussion on goal oriented adaptive finite element methods for FSI problems, we refer to Selim et al. (2011); Dunne (2007, 2006); Grätsch and Bathe (2006); Bengzon and Larson (2010); van der Zee et al. (2008); van der Zee (2009).

29.1 Fluid–structure interaction

Fluids and solids obey the fundamental conservation laws that hold for any adiabatic continuum body: the balance of linear momentum and the conservation of mass. These fundamental conservation laws can be expressed in local form as

$$\mathrm{d}_t(\rho u) - \operatorname{div} \sigma = b, \tag{29.1}$$

$$\mathrm{d}_t(\rho) = 0, \tag{29.2}$$

where (29.1) is the balance of linear momentum and (29.2) is the conservation of mass. Here, $\mathrm{d}_t(\cdot)$ denotes a material time derivative, ρ the density, u the velocity, σ the stress, and b represents a given body force per unit volume. In an FSI problem, the different physical quantities of the fluid and the structure, denoted with subscripts F and S respectively, transfer traction forces and exchange data at a given common fluid–structure boundary. Traction forces are given by normal stresses and at the common fluid–structure boundary, the following equilibrium equation holds:

$$\sigma_F \cdot n_F = -\sigma_S \cdot n_S, \tag{29.3}$$

where n_F and n_S denote the outward normals on the fluid–structure boundary, viewed from the fluid and structure domains, respectively. Hence, $n_F = -n_S$. In addition to (29.3), we also require kinematic

continuity at the common fluid–structure boundary; that is, the velocity of the fluid and the solid are equal at the common fluid–structure boundary. How a continuum responds to stress and in particular to shear stress, distinguishes a fluid continuum from a solid continuum. A fluid cannot withstand shear forces; it will continue to deform as long as the stress is applied. Solids, on the other hand, respond with an angular strain and the strain continues until the displacement is sufficient to generate internal forces that balance the imposed stress. To capture this, the constitutive laws modeling fluids and solids relate the stress tensor to different physical measures. Moreover, these measures are from a practical point of view naturally posed in different frameworks, the so-called Lagrangian framework and the Eulerian framework.

29.1.1 Lagrangian framework and structural mechanics

An essential kinematic measure in structural mechanics is the displacement field which is naturally posed in the Lagrangian framework. In the Lagrangian framework, the motion of a body is related to a fixed material point x_0 and the position of such a point at time t is given by the sufficiently smooth bijective map ϕ that maps the point x_0 at time t to the point $x(t) = \phi(x_0, t)$. The *structure displacement* is defined as $u_S(x_0, t) = \phi(x_0, t) - x_0$ with the corresponding non-singular Jacobi matrix $f = \text{grad}\ \phi$ and Jacobi determinant $j = \det f$. Thus, the material time derivative of a function y in the Lagrangian framework is given by $\mathrm{d}_t(y) = \dot{y}$.

Constitutive laws for hyperelastic materials express the stress tensor σ_S (referred to as the first Piola–Kirchhoff stress) as the Frechét derivative of a given energy functional ψ. The energy functional can depend on different kinds of kinematic measures, and if ψ is dependent on the so-called Green–Lagrange tensor $e = \frac{1}{2}(f^\top f - I)$, the corresponding first Piola–Kirchhoff tensor is given by $\sigma_S = f \cdot \frac{\partial \psi(e)}{\partial e}$. In this chapter, we focus on the compressible St. Venant–Kirchhoff model where the stress is described by the energy functional $\psi(e) = \mu_S \text{tr}(e^2) + \frac{\lambda_S}{2}(\text{tr}(e))^2$, where (μ_S, λ_S) are given positive Lamé constants. Hence, the conservation laws for a St. Venant–Kirchhoff material in the Lagrangian framework is given by

$$\rho_S \ddot{u}_S - \text{div}\ \sigma_S(u_S) = b_S, \tag{29.4}$$

$$\dot{\rho}_S = 0, \tag{29.5}$$

with the corresponding stress tensor $\sigma_S(u_S) = f \cdot (2\mu_S e + \lambda_S \text{tr}(e) I)$. Note that we usually omit the mass conservation equation (29.5) since it is automatically satisfied for compressible materials in the Lagrangian framework. For a more in depth analysis of hyperelastic materials and structural mechanics in general, see Gurtin (1981); Holzapfel (2000).

29.1.2 Eulerian framework and fluid mechanics

In fluid mechanics, the primary variables for describing the fluid motion are the *fluid velocity* u_F and the *fluid pressure* p_F. These variables are naturally posed in the Eulerian framework where the motion of a body is related to a fixed spatial point x and the motion of the body is defined as $u_F(x, t) = u_F(\phi(x_0, t), t)$. Thus, the material time derivative of a function y in the Eulerian framework is given by $\mathrm{d}_t(y) = \dot{y} + \text{grad}\ y \cdot u_F$.

The most common constitutive law for fluids is the Newtonian fluid. For Newtonian fluids, the stress tensor σ_F (referred to as the Cauchy stress) is given by $\sigma_F(u_F, p_F) = 2\mu_F \varepsilon(u_F) - p_F I$, where μ_F denotes the dynamic viscosity and $\varepsilon(\cdot)$ the symmetric gradient. In this chapter, we assume that the fluid is an incompressible Newtonian fluid. The fluid is then described by the incompressible

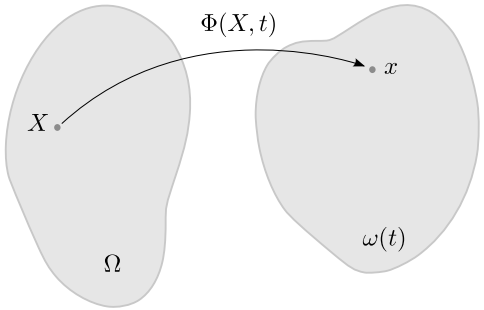


Figure 29.1: The mapping $\Phi(X,t)$ maps a reference point $X \in \Omega$ to the current point $x \in \omega(t)$. The deformation gradient of the reference domain Ω is given by $\text{Grad}\,\Phi = F$, and the volume change of Ω is thus $J = \det(F)$.

Navier–Stokes equations:

$$\rho_F(\dot{u}_F + \text{grad}\, u_F \cdot u_F) - \text{div}\, \sigma_F(u_F, p_F) = b_F, \tag{29.6}$$

$$\text{div}\, u_F = 0. \tag{29.7}$$

For a more in depth analysis of constitutive laws for fluids and for fluid mechanics in general, see Batchelor (1967); Panton (1984).

29.2 FSI and the ALE computational framework

To combine the Lagrangian and the Eulerian frameworks in a computational setting, the fluid traction force from problem (29.6)–(29.7) is transferred to the structure problem (29.4)–(29.5) via the Piola map: $(j\,\sigma_F \cdot f^{-\top}) \cdot n_F = -\sigma_S \cdot n_S$ at the common fluid–structure boundary. The deformation of the structure, given by the structure solution in the material domain, needs to be tracked in the spatial fluid domain and consequently, the mesh in the spatial fluid domain has to be updated. A dynamically deforming mesh without any additional smoothing algorithm will result in a mesh of poor quality. To treat this shortcoming, an additional mesh equation is posed in the fluid domain to enhance the mesh quality. Combining the Lagrangian and the Eulerian frameworks with an additional mesh smoothing algorithm is commonly referred to as the Arbitrary Lagrangian–Eulerian (ALE) method (Donea et al., 1982, 2004). In this method, both the Lagrangian approach, in which the mesh moves with the structure, and the Eulerian approach, in which the mesh represents a fixed reference frame for the fluid, are used. In order to incorporate the mesh equation in the FSI problem, an arbitrary reference frame for the fluid domain is introduced which is independent of the Lagrangian description and the Eulerian description. This arbitrary reference domain is typically the initial undeformed computational domain.

Let Ω be a fixed open domain in $\mathbb{R}^d$ which represents the *reference* (undeformed) computational domain, for $d = 2, 3$. Moreover, let Ω be partitioned into two disjoint open subsets Ω_F and Ω_S such that $\bar{\Omega}_F \cup \bar{\Omega}_S = \Omega$ and $\Omega_F \cap \Omega_S = \emptyset$. Further, let $\omega(t) \in \mathbb{R}^d$ denote the *current* (deformed) computational domain which is similarly partitioned into two disjoint subsets $\omega_F(t)$ and $\omega_S(t)$ such that $\bar{\omega}_F \cup \bar{\omega}_S = \omega$ and $\omega_F(t) \cap \omega_S(t) = \emptyset$, for all time $t \in [0, T]$. The common boundary between the structure and fluid domains is denoted by Γ_{FS} and $\gamma_{FS}(t)$ respectively. In general, to distinguish between variables and operators associated with the reference and current domains, we use upper case and lower case letters respectively. Thus, $\text{Div}\, \Sigma_S(U_S(X,t))$ is the divergence of the structure stress defined on the reference structure domain Ω_S, and $\text{grad}\, u_F(x,t)$ is the current gradient of the fluid velocity defined in the current fluid domain $\omega_F(t)$.

In order to map between the reference domain and the current domain, we introduce the sufficiently smooth bijective map $\Phi(\cdot, t) : \Omega \mapsto \omega(t)$. For any fixed time $t \in [0, T]$, Φ maps a reference point $X \in \Omega$ to the corresponding current point $x \in \omega(t)$; that is, $X \mapsto x = \Phi(X, t)$, see Figure 29.1.

Since we allow the fluid and structure portions of the domain to deform independently (only enforcing that these deformations are identical on the common boundary), the map is split up as follows:

$$\Phi(X,t) = \begin{cases} \Phi_S(X,t), & \forall\, X \in \Omega_S,\ t \in [0,T], \\ \Phi_M(X,t), & \forall\, X \in \Omega_F,\ t \in [0,T]. \end{cases} \tag{29.8}$$

Here, the structure map and the (fluid) mesh map (Φ_S, Φ_M) are defined as

$$\Phi_S(X,t) = X + U_S(X,t), \tag{29.9}$$
$$\Phi_M(X,t) = X + U_M(X,t), \tag{29.10}$$

where (U_S, U_M) are the solutions to the structure problem and the arbitrarily chosen mesh problem. There are several possible ways to formulate and solve the mesh problem to obtain U_M (Hermansson and Hansbo, 2003; López et al., 2008). In the following, we have adopted a time dependent mesh problem related to a linearly elastic description of the fluid domain in which the stress tensor is given by $\Sigma_M(U_M) \equiv \mu_M(\text{Grad}\, U_M + \text{Grad}\, U_M^\top) + \lambda_M \text{tr}(\text{Grad}\, U_M) I$ for some given positive constants (μ_M, λ_M).

To summarize, we identify the three subproblems that together define the fully coupled FSI problem:

- the fluid problem (f) solved in the current fluid domain $\omega_F(t)$;
- the structure problem (S) solved in the reference structure domain Ω_S;
- the mesh problem (M) solved in the reference fluid domain Ω_F.

The corresponding set of equations for the triplet $(f), (S), (M)$ is given by:

$$(f): \quad \rho_F(\dot{u}_F + \text{grad}\, u_F \cdot u_F) - \text{div}\, \sigma_F(u_F, p_F) = b_F \quad \text{in } \omega_F(t), \tag{29.11}$$
$$\text{div}\, u_F = 0 \quad \text{in } \omega_F(t), \tag{29.12}$$
$$(S): \quad \rho_S \ddot{U}_S - \text{Div}\, \Sigma_S(U_S) = B_S \quad \text{in } \Omega_S \times (0,T], \tag{29.13}$$
$$(M): \quad \dot{U}_M - \text{Div}\, \Sigma_M(U_M) = 0 \quad \text{in } \Omega_F \times (0,T], \tag{29.14}$$

together with initial and boundary conditions. We note that, with the proposed notation, the stress from the fluid is transferred to the structure and the movement of the structure is tracked in the fluid domain at the common fluid–structure boundary such that:

$$(J_M\ (\sigma_F \circ \Phi_M) \cdot F_M^{-\top}) \cdot N_F = -\Sigma_S \cdot N_S \quad \text{on } \Gamma_{FS}, \tag{29.15}$$
$$u_F \circ \Phi_M = \dot{U}_S \quad \text{on } \Gamma_{FS}. \tag{29.16}$$

Thus, (29.15)–(29.16) transfer data between all the equations in the FSI system (29.11)–(29.14) at the common FSI interface.

In the numerical solution of the fluid problem (f), we compensate for the additional (unphysical) mesh movement $\dot{u}_M$ in the fluid domain $\omega_F(t)$ introduced by the mesh equation (M). The resulting discrete finite element form of the convective term of the fluid problem takes the form $\rho_F(\dot{u}_F^{hk} + \text{grad}\, u_F^{hk} \cdot (u_F^{hk} - \dot{u}_M^{hk}))$. This additional mesh movement is a pure numerical artifact and is not a part of the continuum representation of the FSI problem.

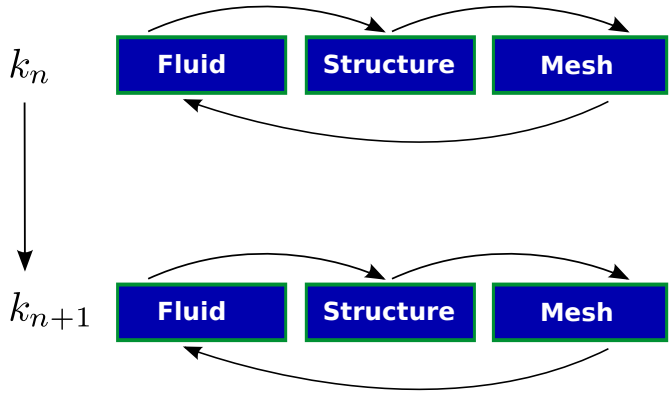


Figure 29.2: A partitioned approach to solving the FSI problem. In each time step k_n, the three subproblems are solved iteratively using a simple fixed point method. The fluid problem is first solved on the given current fluid domain $\omega_F(t)$ and the stress σ_F is evaluated and mapped back to the structure problem in the reference domain. In the structure reference domain Ω_S, the fluid stress is set as a Neumann boundary condition and the structure problem is solved for the given fluid stress. The structure displacement field is then set as a Dirichlet boundary condition at the common fluid–structure boundary for the mesh equation in the fluid reference domain Ω_F. Having obtained the mesh solution, the solution is pushed forward to the current fluid domain and thus defines the new deformed current domain.

29.3 The FSI solver

The proposed system of equations that defines the fully coupled FSI problem (29.11)–(29.14) is a partitioned system, where the subproblems $(f), (S), (M)$ are connected at the fluid–structure interface through the boundary conditions in (29.15)–(29.16). To solve such a system, we utilize a fixed point iteration. The algorithm reads as follows and is illustrated in Figure 29.2.

1. Solve the fluid problem (f).
2. Transfer the fluid stress using (29.15) and solve the structure problem (S).
3. Solve the mesh problem (M) and update the fluid domain.
4. Repeat steps (1)–(3) until convergence.
5. Move on to the next time step.

We note that one may, alternatively, start each time step with an extrapolation of the motion of the structure domain, followed by a solution of the mesh problem, then the fluid problem etc. This might lead to fewer iterations. However, for this work, we have adopted the simple strategy described above.

The two subproblems (f), (S) define a classic set of equations from fluid and structure mechanics. To solve the coupled system, a solver framework for handling both types of physics is needed. For this purpose, we have used the multiphysics framework `CBC.Solve` developed at the Center for Biomedical Computing at Simula Research Laboratory. Currently, `CBC.Solve` consists of two core components; `CBC.Flow` and `CBC.Twist`. These are frameworks explicitly developed for solving fluid mechanics problems and structure mechanics problems, respectively. In the subsequent sections, we will briefly explain these frameworks and the code that solves the FSI problem (29.11)–(29.14).

Python code

```
class NavierStokesSolver(CBCSolver):
    "Navier-Stokes solver"

    def __init__(self, problem):
        "Initialize Navier-Stokes solver"

        ...

        # Tentative velocity step (sigma formulation)
        U = 0.5*(u0 + u)
        F1 = rho*(1/k)*inner(u - u0, v)*dx \
           + rho*inner(grad(u0)*(u0 - w), v)*dx \
           + inner(sigma(U, p0), epsilon(v))*dx \
           + inner(p0*n, v)*ds \
           - mu*inner(grad(U).T*n, v)*ds \
           - inner(f, v)*dx
        a1 = lhs(F1)
        L1 = rhs(F1)

        # Pressure correction
        a2 = inner(k*grad(p), grad(q))*dx
        L2 = inner(k*grad(p0), grad(q))*dx \
           - div(u1)*q*dx

        # Velocity correction
        a3 = inner(u, v)*dx
        L3 = inner(u1, v)*dx \
           + inner(k*grad(p0 - p1), v)*dx
```

Figure 29.3: A code segment of the fluid solver in `CBC.flow`. The momentum equation is multiplied with a test function `v` and the continuity equation is multiplied with a test function `q`. In the first step, the tentative velocity is computed from the momentum equation using a fully implicit formulation of the convective term and the previously computed pressure. Here, `w` denotes the mesh velocity $\dot{u}_M$. In the next step, the pressure is corrected with the continuity equation based on the computed velocity `u1` from the first step. Finally, the velocity is corrected using the corrected pressure.

29.3.1 Fluid subproblem

The fluid subproblem in (29.11)–(29.12) is solved using the `CBC.Solve` module `CBC.Flow`. The fluid problem can be solved in an Eulerian coordinate system or in an ALE coordinate system, and the solver is based on the stress formulation of the so-called Incremental Pressure Correction Scheme (IPCS) (Goda, 1979). The fluid velocity u_F and the fluid pressure p_F are discretized in space using Taylor–Hood elements. The resulting nonlinear variational problem is solved in three steps. In the first step, the tentative fluid velocity is computed from the momentum equation using the previously known pressure. After this step, the pressure at the current time step is computed and corrected using the continuity equation. Finally, in the third step, the velocity is corrected using the corrected pressure. The implementation is illustrated with a code segment from the class `NavierStokesSolver` in Figure 29.3. For a more comprehensive discussion on how to solve and implement different solvers for the incompressible Navier–Stokes equations in FEniCS, see Chapter 21.

29.3.2 Structure subproblem

`CBC.Twist` is a solver collection for structure mechanics problems. This module solves the given structure problem in a Lagrangian coordinate system. The solver allows the user to easily pose problems and provides many standard material models, including St. Venant–Kirchhoff, Mooney–Rivlin, neo-Hookean, Isihara, Biderman and Gent–Thomas. New models may be added easily since the interface allows the user to provide an energy functional as a function of a suitable kinematic measure, such as the Green–Lagrange strain. Both a static and an energy-momentum preserving time-dependent solver are provided. The space discretization relies upon first order Lagrange elements and for the time discretization several different schemes are available, such as the cG(1) method (Eriksson

Python code

```
class CG1MomentumBalanceSolver(CBCSolver):
    def __init__(self, problem):
        ...

        # The variational form corresponding to
            hyperelasticity
        L = rho0*inner(p - p0, v)*dx + k*inner(sigma,
            grad(v))*dx \
          - k*inner(b, v)*dx + inner(u - u0, q)*dx \
          - k*inner(p_mid, q)*dx

        # Add contributions form the Neumann boundary
        neumann_conditions = problem.neumann_conditions()
        neumann_boundaries = problem.neumann_boundaries()

        boundary = MeshFunction("uint", mesh,
            mesh.topology().dim() - 1)
        boundary.set_all(len(neumann_boundaries) + 1)

        for (i, neumann_boundary) in
            enumerate(neumann_boundaries):
            compiled_boundary =
                compile_subdomains(neumann_boundary)
            compiled_boundary.mark(boundary, i)
            L = L - k*inner(v, neumann_conditions[i])*ds(i)

        a = derivative(L, U, dU)
```

Figure 29.4: A code segment from the cG(1) version of the structure solver `CBC.Twist`. In the cG(1) method, the structure problem is re-written as a first order system in time by introducing the additional equation $P_S - \dot{U}_S = 0$ to the structure equation (29.13). The two resulting equations are multiplied with test functions `v` and `q` respectively, and adding the two equations yields the nonlinear variational form `L`. The nonlinear variational form `L` contains the structure velocity `p` and the first Piola–Kirchhoff stress tensor `sigma` (which is a function of the structure displacement U_S). The nonlinear form `L` is linearized using the FEniCS function `derivative` where `U` represents the mixed finite element function containing the structure solution (U_S, P_S). We note that Neumann conditions, such a fluid stress, are imposed in the variational form `L` while the Dirichlet conditions are set directly in the Newton solver. We also note that the proposed variational form holds for a large amount of different structure models, in which the first Piola–Kirchhoff stress tensor `sigma` is given by an appropriate material model.

et al., 1996) or the "HHT" method (Hilber et al., 1977). In the cG(1) method, used in this chapter, the structure problem is re-written as a first order system in time by introducing the additional equation $P_S - \dot{U}_S = 0$ to the structure equation (29.13). The structure stress tensor, regardless of material model, is given as the first Piola–Kirchhoff tensor and the nonlinear variational problem is solved using Newton's method. The implementation of the cG(1) version is illustrated in the code segment from the class `CG1MomentumBalanceSolver` in Figure 29.4. For a more comprehensive discussion of how to solve structure problems using `CBC.Twist`, and especially how to implement different material models, see Chapter 27.

29.3.3 Mesh subproblem

The linear mesh subproblem is solved using first order Lagrange elements in space along with a standard cG(1) formulation in time. We note that although piecewise quadratic functions are used to approximate the velocity in the fluid problem, an affine mapping is used to map the elements in the finite element discretization. It is therefore suitable to approximate the mesh problem using piecewise linears (not quadratics); see Formaggia and Nobile (1999).

The implementation of the variational forms describing the mesh subproblem is illustrated in Figure 29.5. The Dirichlet boundary conditions on the mesh subproblem are imposed weakly through the introduction of the Lagrange multiplier, P_M. This choice introduces coupling between the mesh

Python code

```
# Define cG(1) scheme for time-stepping
a = inner(u, v)*dx + 0.5*k*inner(sigma(u), sym(grad(v)))*dx
L = inner(u0, v)*dx-0.5*k*inner(sigma(u0), sym(grad(v)))*dx
```

Figure 29.5: A code segment of the mesh solver `MeshSolver`.

and structure subproblems in the linearized (adjoint) dual problem which is described below.

29.4 Duality–based error control

As mentioned in the beginning of this chapter, in many cases we are only interested in computing one output quantity of the fully coupled FSI system. This output quantity is commonly referred to as the goal functional. To ensure a high level of accuracy of the functional of interest, the error in the goal functional needs to be controlled. In finite element discretizations, *a posteriori* error analysis provides a general framework for controlling the approximation error of the solution. The extension of the classical *a posteriori* error analysis to estimate the error in a goal functional has been under development over the past two decades, and the technique originates form Eriksson et al. (1995); Becker and Rannacher (2001). This technique is based on the solution of an auxiliary linearized dual (adjoint) problem in order to estimate the error in a given goal functional. By solving the dual problem, one may construct an adaptive algorithm that efficiently targets the computation of a specific goal functional $\mathcal{M}$, such that

$$|\mathcal{M}(U) - \mathcal{M}(U^{hk})| \leqslant \text{TOL}. \tag{29.17}$$

Here, $U - U^{hk} \equiv e$ is the error of the finite element solution in space (h) and time (k), and $\text{TOL} > 0$ is a user-defined tolerance. To define the dual problem for the FSI problem (29.11)–(29.14), we pull the fluid subproblem (f) back from the current fluid domain $\omega_F(t)$ to the fluid reference domain Ω_F using the map Φ_M:

$$(F) \xleftarrow{\Phi_M^{-1}} (f). \tag{29.18}$$

With the fluid problem (F) defined in the reference domain, all the three subproblems (F, S, M) are posed in the reference domain Ω, and we may thus formulate a monolithic counterpart to the FSI problem (29.11)–(29.14). The abstract nonlinear variational form reads: find $U \equiv \{U_F, P_F, U_S, P_S, U_M, P_M\} \in V$ such that

$$a(U; v) = L(v), \tag{29.19}$$

for all $v \equiv \{v_F, q_F, v_S, q_S, v_M, q_M\} \in \hat{V}$, where the trial and test spaces $(V, \hat{V})$ are associated with the geometrically conforming parts of Ω_F and Ω_S, respectively. By introducing the linearized variational form $a'(U; \delta U, v) \equiv \frac{\partial a(U;v)}{\partial U}\delta U$, we note that by the chain rule

$$\begin{aligned} \overline{a'}(e, v) &\equiv \int_0^1 a'(sU + (1-s)U^{hk}; e, v)\,\mathrm{d}s \\ &= \int_0^1 \frac{\mathrm{d}}{\mathrm{d}s} a(sU + (1-s)U^{hk}; v)\,\mathrm{d}s \\ &= L(v) - a(U^{hk}; v) \\ &\equiv r(v), \end{aligned} \tag{29.20}$$

where $r(\cdot)$ is the (weak) residual of (29.19). We now define the following dual problem: find the dual solution $Z \equiv \{Z_F, Y_F, Z_S, Y_S, Z_M, Y_M\} \in V^*$ such that

$$\overline{a'}^*(Z, v) = \mathcal{M}(v), \tag{29.21}$$

for all $v \equiv \{v_F, q_F, v_S, q_S, v_M, q_M\} \in \hat{V}^*$, where the dual test and trial spaces are to be defined below. We assume that the goal functional in (29.21) can expressed in the form

$$\mathcal{M}(v) \equiv \langle \psi^T,\ v(\cdot, T)\rangle + \int_0^T \langle \psi^t,\ v\rangle \,\mathrm{d}t, \tag{29.22}$$

where (ψ^T, ψ^t) are suitable Riesz representers for the goal functional. Based on the solution of the dual problem (29.21) and by inserting $v = e$, we obtain the following computable error representation:

$$\begin{aligned} \mathcal{M}(e) &= \overline{a'}^*(Z, e) \\ &= \overline{a'}(e, Z) \\ &= L(Z) - a(U_{hk}; Z) \\ &= r(Z); \end{aligned} \tag{29.23}$$

that is, the error is the (weak) residual of the dual solution. The dual problem (29.21) measures the sensitivity of the problem with respect to the given goal functional. The dual solution Z contains the dual variables where, for instance, (Z_F, Y_F) represents the dual fluid velocity and dual fluid pressure. The additional dual variable Y_M represents the weakly imposed dual mesh Lagrange multiplier at the common fluid–structure interface. This term is added to account for the coupling between the mesh and structure equations of the FSI problem. We notice that in order for the error representation (29.23) to be consistent, the corresponding dual trial and test spaces are defined as $(V^*, \hat{V}^*) = (\hat{V}, V_0)$, where $V_0 = \{v - w :\ v, w \in V\}$. We interpret the Riesz representer ψ^T is an initial condition in the dual problem and that the dual problem (29.21) runs backwards in time. In the computations, the stated dual problem (29.21) is replaced by the approximated linearized form $\overline{a'}^*(Z, v) \equiv a'^*(U; Z, v) \approx a'^*(U^{hk}; Z, v) = a'(U^{hk}; v, Z)$.

We may express the dual problem on block form as

$$\begin{bmatrix} \hat{v}_F & \hat{v}_S & \hat{v}_M \end{bmatrix} \begin{bmatrix} A_{FF} & A_{FS} & A_{FM} \\ A_{SF} & A_{SS} & A_{SM} \\ A_{MF} & A_{MS} & A_{MM} \end{bmatrix}^\top \begin{bmatrix} \hat{Z}_F \\ \hat{Z}_S \\ \hat{Z}_M \end{bmatrix} = \begin{bmatrix} \mathcal{M}_F \\ \mathcal{M}_S \\ \mathcal{M}_M \end{bmatrix}. \tag{29.24}$$

Here, $\hat{v}_F = (v_F, q_F)$ represents the fluid test functions, A_{FF} denotes the fluid problem linearized around the fluid variables (U_F, P_F), $\hat{Z}_F = (Z_F, Y_F)$ are the dual fluid variables and $\mathcal{M}_F$ the fluid goal functional and so on. The interpretation of the individual blocks in (29.24) is that, for instance, $\hat{v}_F A_{SF}^\top \hat{Z}_S$ is interpreted as $\overline{a'}^*_{SF}(\hat{Z}_S, \hat{v}_F)$ which is the (adjoint) structure form linearized around the fluid variables. We notice that the A_{FS} is zero since the fluid problem linearized around the structure variables are identically zero. Thus, the grey colored entries are by definition zero and that $A_{MS} \neq 0$ by the introduction of the dual mesh Lagrange multiplier Y_M.

To be able to bound the errors in space and time, we add and subtract suitable interpolants (π_h, π_{hk}) in space and space/time in the error representation (29.23) to obtain the following *a posteriori*

Python code

```
# Fluid residual contributions
R_F0 = w*inner(EZ_F - Z_F, Dt_U_F - div(Sigma_F))*dx_F
R_F1 = avg(w)*inner(EZ_F("+") - Z_F("+"),
              jump(Sigma_F, N_F))*dS_F

R_F2 = w*inner(EZ_F - Z_F, dot(Sigma_F, N_F))*ds
R_F3 = w*inner(EY_F - Y_F, div(J(U_M)*
         dot(inv(F(U_M)), U_F)))*dx_F

# Structure residual contributions
R_S0 = w*inner(EZ_S - Z_S, Dt_P_S - div(Sigma_S))*dx_S
R_S1 = avg(w)*inner(EZ_S("-") - Z_S("-"),
             jump(Sigma_S, N_S))*dS_S

R_S2 = w("-")*inner(EZ_S("-") - Z_S("-"),
             dot(Sigma_S("-") - Sigma_F("+"),
             -N_F("+")))*d_FSI

R_S3 = w*inner(EY_S - Y_S, Dt_U_S - P_S)*dx_S

# Mesh residual contributions
R_M0 = w*inner(EZ_M - Z_M, Dt_U_M - div(Sigma_M))*dx_F
R_M1 = avg(w)*inner(EZ_M("+") - Z_M("+"),
              jump(Sigma_M, N_F))*dS_F

R_M2 = w("+")*inner(EY_M - Y_M, U_M - U_S)("+")*d_FSI
```

Figure 29.6: A code segment illustrating the element-wise space error indicators. The indicators consist of three parts where each subproblem is represented. These estimates are obtained by element wise integration by parts of the finite element formulation which is weighted by the dual solution. This results in element indicators R_T defined on the cells and jump terms $R_{\partial T}$ across element edges. Here, `w` represents a discontinuous function of order zero and `jump` denotes the jump across an element edge `dS`. The difference $Z - \pi_h Z$ is approximated with `EZ - Z` where `EZ` is the extrapolated finite element approximation on a richer space and `Z` is the finite element approximation.

error estimate: $|\mathcal{M}(U) - \mathcal{M}(U^{hk})| \leqslant E_h + E_k + E_c$, where

$$E_h \equiv \sum_{n=1}^{N} \int_{I_n} \sum_{T \in \mathcal{T}_h} |\langle R_T,\, Z - \pi_h Z\rangle_T| + |\langle \tfrac{1}{2}[\![R_{\partial T}]\!],\, Z - \pi_h Z\rangle_{\partial T}|\,\mathrm{d}t, \tag{29.25}$$

$$E_k \equiv\ \mathcal{S}(T) \max_{[0,T]} \{k_n |r_k^n|\}, \tag{29.26}$$

$$E_c \equiv |r(\pi_{hk} Z)|. \tag{29.27}$$

Here, E_h estimates the space discretization error which on each space-time slab $S_n = \mathcal{T}_h \times I_n$ is expressed as the sum of error indicators R_T and $R_{\partial T}$ from the cells of the mesh, weighted by the interpolation error of the dual solution. The implementation of these indicators is illustrated in Figure 29.6.

The time discretization error estimate E_k consists of the local time step size k_n multiplied with a local algebraic residual r_k^n and the global stability factor $\mathcal{S}(T) \approx \int_0^T \|\dot{Z}\|_{l^2}\,\mathrm{d}t$. Finally, the computational error estimate E_c accounts for the error introduced when the proposed Galerkin method is solved using a non-Galerkin method, e.g. the IPCS for the fluid subproblem. Also, in addition to the proposed mesh equation, an additional local mesh smoothing is added to the fluid mesh. For a more comprehensive discussion and derivation of this *a posteriori* estimate see Selim et al. (2011).

29.4.1 The adaptive algorithm

With the *a posteriori* error estimate presented in (29.25)–(29.27), we construct an algorithm based on a feedback process that provides an adaptive space-time discretization such that (29.17) holds. In order to determine the stopping criteria for the space and time discretizations, the user defined tolerance TOL needs to be weighted such that

$$\mathrm{TOL} = \mathrm{TOL}_h + \mathrm{TOL}_k + \mathrm{TOL}_c, \tag{29.28}$$

where $\mathrm{TOL}_h = w_h\mathrm{TOL}$, $\mathrm{TOL}_k = w_k\mathrm{TOL}$, $\mathrm{TOL}_c = w_c\mathrm{TOL}$. We here take $w_h = w_k = w_c = 1/3$. The weight w_c affects the tolerance used for the fixed point algorithm when solving (29.11)–(29.14). Based on the spatial error estimate E_h, we refine the mesh until $E_h \leqslant \mathrm{TOL}_h$. There are various ways in which refine the mesh and to determine which elements to refine. In the examples to come, we have adopted the Rivara recursive bisection algorithm as the refinement algorithm and the so-called Dörfler (Dörfler, 1996) marking strategy. The Dörfler marking strategy is based on the idea that for a given $\alpha \in (0, 1]$, a minimum number of elements N is determined such that

$$\sum_{i=1}^{N} \eta_{T_i} \geqslant \alpha \sum_{T \in \mathcal{T}_h} \eta_T, \tag{29.29}$$

where $\{\eta_{T_i}\}_{i=1}^{|\mathcal{T}_h|}$ is a list of error indicators sorted in decreasing order. The adaptive time step size is based on the error estimate E_k and connects the global error to the local error over time. As a first approximation, we may choose the local time step size such that

$$k_n = \mathrm{TOL}_k / (|r_k^{n-1}|\, \mathcal{S}(T)). \tag{29.30}$$

However, this particular choice of time step size introduces oscillations in the time step size since a small algebraic residual gives a large time step which results in a large residual and so on. To overcome this behavior, we use a smoothed version (Logg, 2004), where (29.30) is replaced by

$$k_n = \frac{2\bar{k}_n k_{n-1}}{\bar{k}_n + k_{n-1}}, \tag{29.31}$$

and $\bar{k}_n = \mathrm{TOL}_k / (|r_k^{n-1}|\, \mathcal{S}(T))$. Finally, the estimate for the computational error E_c is only considered in the stopping criterion for the total error.

The main outline of the adaptive FSI algorithm is depicted in Figure 29.7. In the `FSISolver`, the FSI problem (29.11)–(29.14) (referred to as the primal problem) is solved in the module `PrimalSolver`, which solves and transfers data from the three subproblems defined in `CBC.Twist`, `CBC.Flow` and `MeshSolver`. Once the entire primal problem is solved, the primal data is passed to the `DualSolver`. In the dual solver, the linearized dual problem (29.21) is solved. The primal and dual solutions are passed to the module `Residuals` where the error estimate (29.25)–(29.27) is evaluated. The code for the `FSISolver` is illustrated in Figure 29.8 and Figure 29.9.

To summarize, the adaptive feedback process involves the following steps:

1. Solve the partitioned (primal) FSI problem (29.11)–(29.14) for $t \in (0, T]$. For each time step, determine the local time step size k_n according to (29.31).

2. Solve the dual problem (29.21) for $t \in [0, T)$ using the same time step size as in the primal problem.

3. Evaluate the error estimate (29.25)–(29.27) and refine the computational domain in space.

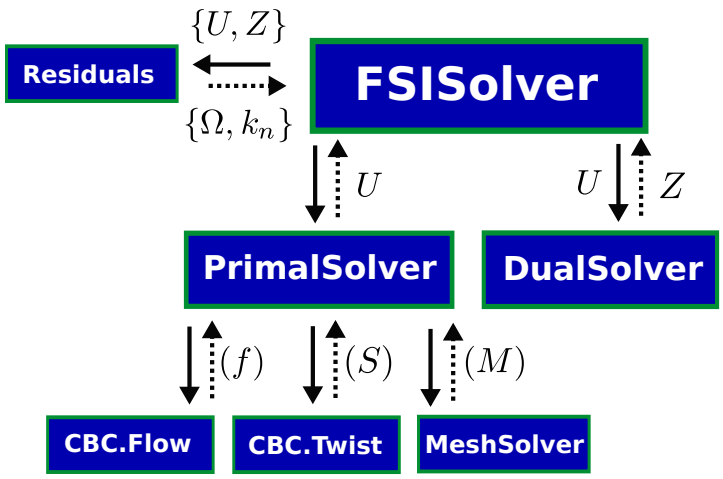


Figure 29.7: A schematic picture of the adaptive FSI algorithm. The primal problem (29.11)–(29.14) is solved iteratively in the `PrimalSolver` and the solution U is passed to the `FSISolver`. The dynamic time step k_n is calculated for each time step in the iterative solver `PrimalSolver` using (29.31) in the module `Residuals`. After the primal problem is solved on the entire time interval, the dual problem is solved with the same time steps as in the primal solution in the module `DualSolver`. Once the dual is solved, the error estimates are evaluated and a new mesh is created.

4. Repeat steps 1 – 3 until $\mathcal{M}(e) \leqslant \text{TOL}$.

29.5 Numerical examples

To demonstrate the above described adaptive algorithm, we solve two simple 2D problems. These problems have different characteristics and they demonstrate how the proposed adaptive algorithm provides both an adequate adaptive mesh refinement and time step selection.

29.5.1 Channel with flap

The first problem is a channel flow with a completely immersed structure called "the flap". The computational domain is given by $\Omega = (0,1) \times (0,4)$, with the structure domain $\Omega_S = (1.4,1.6) \times (0,0.5)$ and the fluid domain $\Omega_F = (\Omega \setminus \Omega_S)^\circ$. For boundary conditions, we consider a pressure driven flow and the flap is attached at the channel wall. As goal functional, we have used the average displacement of the structure in the positive x_1-direction; that is,

$$\mathcal{M}_S(v_S) = \int_0^T \langle \psi_S^t,\, v_S \rangle \, \mathrm{d}t, \tag{29.32}$$

where $\psi_S^t = (1,0)$. The physical parameters related to the problem is set to $(\rho_F, \mu_F) = (1, 0.02)$, $(\rho_S, \mu_S, \lambda_S) = \frac{1}{4}(15, 75, 125)$ and $(\mu_M, \lambda_M) = (3.8461, 5.76)$. The discretization parameters are set to $(\text{TOL}, w_h, w_k, w_c) =$
$(0.05, 0.45, 0.45, 0.1)$ with an initial time step size 0.02 and final time $T = 0.5$. The adaptive primal FSI solution is depicted in Figure 29.10 and the corresponding dual solutions are illustrated in Figure 29.11 and in Figure 29.12.

29.5.2 Driven cavity with an elastic bottom

The second problem is a driven cavity with an elastic bottom. Here, the computational domain is given by $\Omega = (0,2) \times (0,2)$, with structure domain $\Omega_S = (0,2) \times (0,0.5)$ and the fluid domain

Figure 29.8: The adaptive solver class `FSISolver`. Here, the problem specific data is passed through the variable `problem`. In the first adaptive loop, we make an initial guess of the stability factor $\mathcal{S}(T) = 1$ in order to adapt the time step in the first loop. The variable name `error` represents the sum of $E_h + E_k + E_c$ in (29.25)–(29.27) and `indicator` represents η_T in (29.29).

Python code

```python
class FSISolver(CBCSolver):

    def __init__(self, problem):
        "Initialize FSI solver"
            ...
    def solve(self):
        "Solve the FSI problem (main adaptive loop)"

        # Create empty solution (return value when primal is
             not solved)
        U = 5*(None,)

        # Initial guess for stability factor
        ST = 1.0

        # Adaptive loop
        while True:
            # Solve primal problem
            if self.parameters["solve_primal"]:
                primal_solver = PrimalSolver(self.problem,
                     self.parameters)
                U = primal_solver.solve(ST)

            else:
                info("Not solving primal problem")

            # Solve dual problem
            if self.parameters["solve_dual"]:
                dual_solver = DualSolver(self.problem,
                     self.parameters)
                dual_solver.solve()
            else:
                info("Not solving dual problem")
```

Python code

```
            # Estimate error and compute error indicators
            if self.parameters["estimate_error"]:
                error, indicators, E_h =
                     estimate_error(self.problem)
            else:
                info("Not estimating error")
                error = 0

            # Check if error is small enough
            tolerance = self.parameters["tolerance"]
            if error <= tolerance:
                break
            else:

            # Check if mesh error is small enough
            mesh_tolerance = tolerance *
                 self.problem.space_error_weight()
            if E_h <= mesh_tolerance:
                info("Freezing the current mesh")
            else:
                # Refine mesh
                problem = self.problem
                mesh = refine_mesh(problem,
                                   problem.mesh(),
                                   indicators)
                problem.init_meshes(mesh)

        # Return solution
        return U
```

Figure 29.9: The adaptive solver class `FSISolver`, continued.

$\Omega_F = (\Omega \setminus \Omega_S)^\circ$. At the top of the fluid domain, the fluid has the regularized tangential velocity profile in x_1-direction

$$u_F = \begin{cases} 2x, & x \in [0, 0.25], \\ 0.5, & x \in (0.25, 1.75), \\ 2(2-x), & x \in [1.75, 2], \end{cases} \tag{29.33}$$

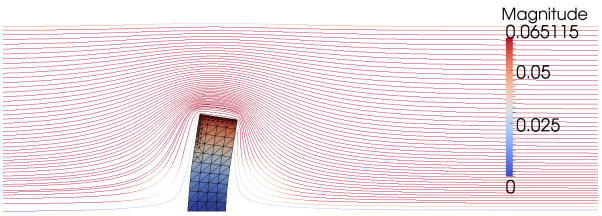
for all $t \in [0, 5]$. The structure is attached at the bottom and the goal functional is set as the average structure displacement in the positive x_2-direction; that is,

$$\mathcal{M}_S(v_S) = \int_0^T \langle \psi_S^t,\, v_S \rangle \, \mathrm{d}t, \tag{29.34}$$

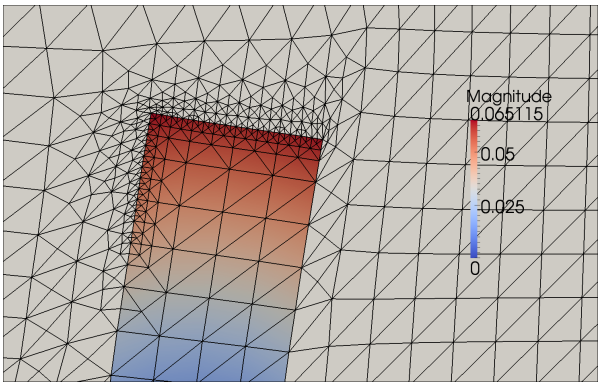
where $\psi_S^t = (0, 1)$. The physical parameters related to the problem is set to $(\rho_F, \mu_F) = (1, 1)$, $(\rho_S, \mu_S, \lambda_S) = (2, 3, 3)$ and $(\mu_M, \lambda_M) = (3.8461, 5.76)$. The discretization parameters are set to $(\mathrm{TOL}, w_h, w_k, w_c) = (0.5, 0.45, 0.45, 0.1)$ with an initial time step size 0.05. In contrast to the previous problem, the solution, and in particular the structure displacement, varies substantially over time. The adaptive primal FSI solution is depicted in Figure 29.13 and the dynamic time step size is illustrated in Figure 29.14.

29.6 Conclusions

In this chapter, an adaptive finite element method for FSI problems has been formulated and its implementation in FEniCS has been demonstrated. By relating the fully coupled partitioned FSI problem (29.11)–(29.14) in a moving domain to a dual problem (29.21) posed on a fixed reference



(a)

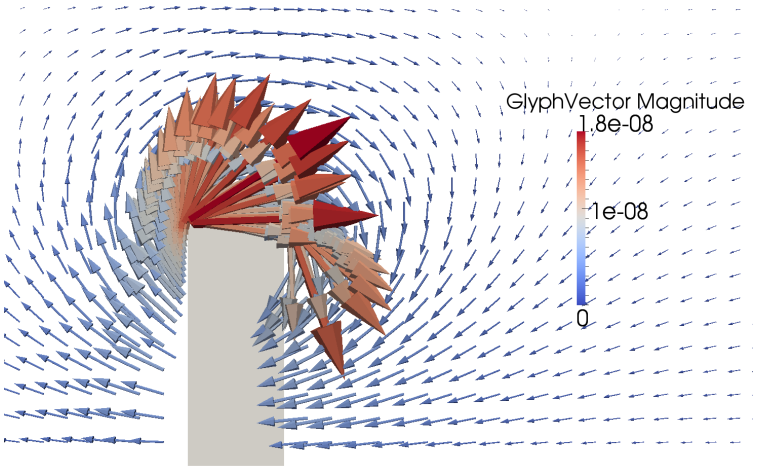


(b)

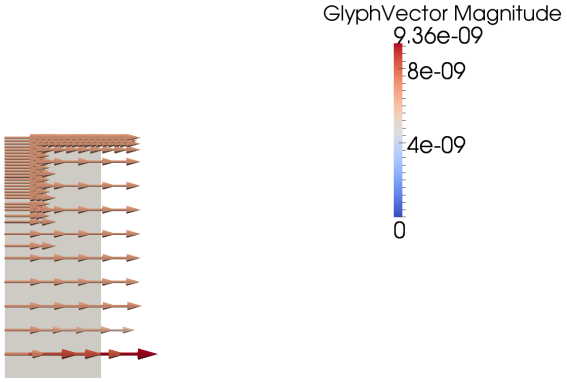
Figure 29.10: The adaptive FSI solution to the channel with flap problem depicted in the current domain $\omega(t)$ at final time $t = 0.5$. In Figure (a), the fluid velocity solution is illustrated using streamlines. A close view of the adaptive mesh is given in Figure (b). The mesh is refined in the immediate area around the structure.

domain, an adapted space and time discretization is obtained.

`CBC.Solve` is a collaboratively developed open source project (released under the GNU GPL) that is freely available from its source repository at `https://launchpad.net/cbc.solve`. Its only dependency is a working FEniCS installation. The fluid–structure interaction solver presented in this chapter is available as CBC.Swing as part of CBC.Solve. The software is released with the goal that it will allow users to easily solve fluid problems, structure and FSI problems. Everyone is encouraged to fetch and try it. Users are also encouraged to modify the code to better suit their own purposes, and contribute changes that they think are useful to the community.



(a) Z_F



(b) Z_S

Figure 29.11: The dual fluid velocity solution Z_F and the dual structure solution Z_S at the "final time" $t = 0$ in the reference domain Ω. Since the only driving force of the fully coupled dual problem is the goal functional (29.32), the dual fluid Z_F is concentrated around the top left corner of the structure where the structure displacement is large. The dual structure displacement Z_S illustrates the choice of goal functional in (29.32).

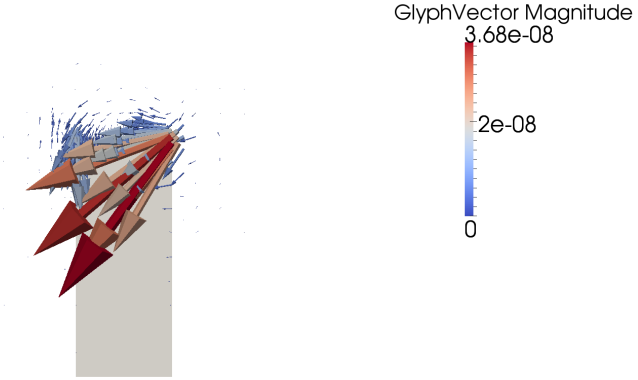
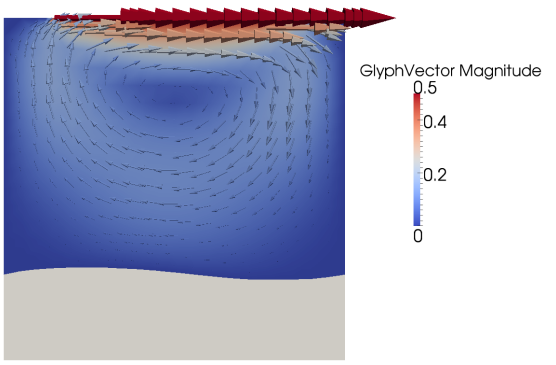
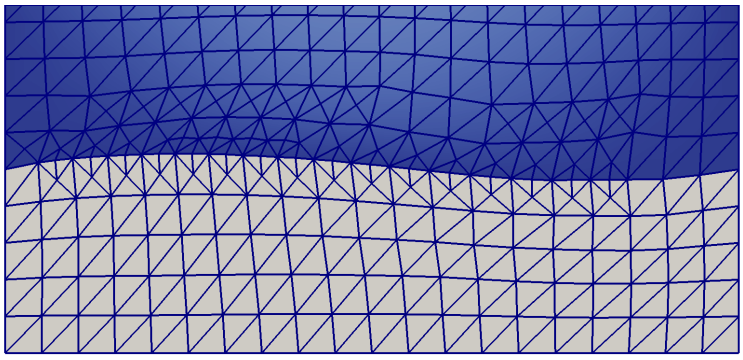


Figure 29.12: The dual mesh displacement Z_M to the channel with flap problem solved in the reference domain Ω. The dual mesh displacement is large close to the top right corner of the structure. This is expected since the mesh in the current domain $\omega(t)$ is significantly compressed in this region.

Figure 29.13: Figure (a) shows the adaptive FSI solution to the driven cavity problem with an elastic bottom at time $t = 2$. The structure does not reach a steady position; instead the structure moves up and down at the common fluid–structure boundary. Figure (b) is a close up view of the refined mesh at the FSI boundary.

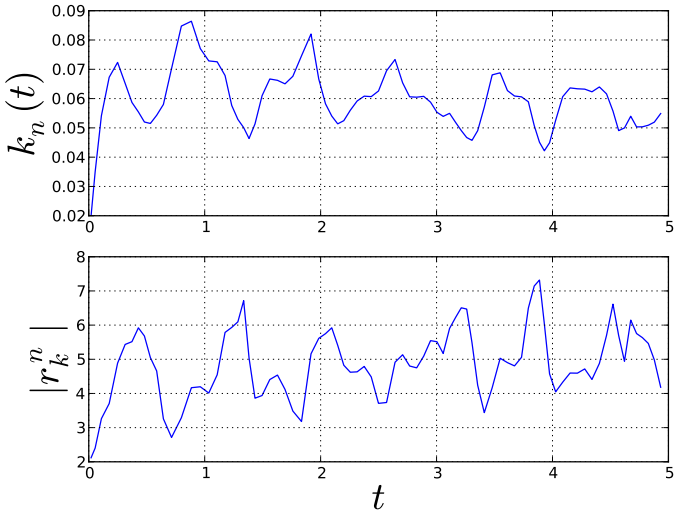


(a)



(b)

Figure 29.14: The time step k_n and the algebraic residual r_k^n as a function of time. As seen in the picture, the solution has a large variation in terms of the magnitude of the residual r_k^n.



30 *Modeling evolving discontinuities*

By Mehdi Nikbakht and Garth N. Wells

We present a framework for solving partial differential equations with discontinuities in the solution across evolving surfaces. The partition-of-unity/extended finite element approach is adopted, and it is demonstrated that such methods can be used in combination with a form compiler to generate equation-specific parts of a program. The automated generation of code makes it straightforward to incorporate discontinuities in formulations involving multiple fields, using both Lagrange and non-Lagrange basis functions. The approach is illustrated through some salient code extracts.

30.1 Background

The numerical solution of differential equations with discontinuities is important in a range of fields. A notable example is the propagation of cracks. Accounting for evolving discontinuities across *a priori* unknown surfaces in simulations using the finite element method poses significant challenges. Early attempts focused on mesh adaption to construct meshes that conformed to the discontinuity surface. More recently, techniques have been developed that make it possible to include discontinuous functions in a finite element basis, with the surface across which the functions are discontinuous being independent of topology of the underlying mesh. These techniques exploit the partition-of-unity property of a standard finite element basis, and are known by a variety of names, including the extended finite element method, the partition of unity method and the generalized finite element method. Discontinuous solutions can evolve during the solution of an equation, without the finite element mesh being adapted to account for the discontinuity explicitly.

We present in this chapter an automated framework for modeling evolving discontinuities which is based on the extended finite element method. The extended finite element method is an approach to modeling discontinuities independently of the underlying mesh (Belytschko and Black, 1999; Moës et al., 1999; Wells and Sluys, 2001). An overview of the extended finite element method and similar methods can be found in Babuška et al. (2003).

The Unified Form Language (UFL) is used to express variational forms for problems with discontinuous solutions, and extensions to the FEniCS Form Compiler (FFC) are developed for generating problem-dependent parts of the computer code from UFL input. To assemble and solve complete problems, various tools are built upon the library DOLFIN. With the developed framework, it is possible to use arbitrary combinations of different finite element bases, and combinations of bases that may or may not be discontinuous across a given surface. Our earlier effort in this direction demonstrated the viability of the approach, but was limited in scope (Nikbakht and Wells, 2009). Only continuous Lagrange basis function functions were available, and only integration on cells was supported. Moreover, the consistent abstractions and algorithms provided by UFL, such as automatic differentiation, were not yet available. Less visible, the entire library has been re-written to permit far greater flexibility.

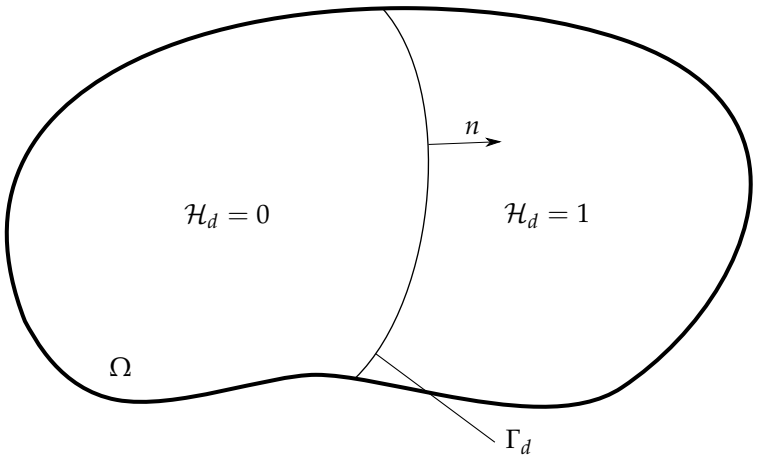


Figure 30.1: Domain Ω intersected by a discontinuity surface Γ_d.

In the remainder of this chapter, we review briefly the extended finite element for modeling discontinuities and formulate it in such a way that the computer input will resemble closely the mathematical description. The software components used in the automated framework are then discussed, as are aspects of the design, including interfaces. The approach is then illustrated using code extracts for a range of examples. The computer code for partition of unity compiler and solver are available at `https://launchpad.net/ffc-pum` and `https://launchpad.net/dolfin-pum`, respectively.

30.2 Partition-of-unity/extended finite element method

Consider a domain $\Omega \subset \mathbb{R}^d$, where $1 \leqslant d \leqslant 3$, that contains the surface Γ_d across which the function $u : \Omega \backslash \Gamma_d \to \mathbb{R}$ is discontinuous (see Figure 30.1). To denote functions that are evaluated at a surface, but by approaching the surface from opposite sides of the surface, the subscripts '$+$' and '$-$' will be used. The outward normal to $\partial\Omega$ and Γ_d will be denoted by n. The vector n is in the direction of the '$+$' side. If we wish to find a function u that satisfies the Poisson equation:

$$-\Delta u = f \quad \text{in } \Omega \backslash \Gamma_d, \tag{30.1}$$

$$u = 0 \quad \text{on } \partial\Omega, \tag{30.2}$$

$$\nabla u_+ \cdot n = q \quad \text{on } \Gamma_d, \tag{30.3}$$

$$[\![\nabla u]\!] \cdot n = 0 \quad \text{on } \Gamma_d, \tag{30.4}$$

where $f : \Omega \to \mathbb{R}$ is a source term, $q : \Gamma_d \to \mathbb{R}$ is the flux across discontinuity surface Γ_d and $[\![a]\!] = a_+ - a_-$. Assuming that the flux on the discontinuity surface is given by $q = q([\![u]\!])$, the corresponding variational problem reads: find $u \in H_0^1(\Omega \backslash \Gamma_d)$ such that

$$\int_{\Omega \backslash \Gamma_d} \nabla u \cdot \nabla v \, dx + \int_{\Gamma_d} q([\![u]\!])\, [\![v]\!] \, ds = \int_\Omega f v \, dx \quad \forall \; v \in H_0^1(\Omega \backslash \Gamma_d). \tag{30.5}$$

To compute approximate solutions to this problem, the task is to formulate a suitable Galerkin finite element method that can accommodate the discontinuous nature of the solution.

Consider a decomposition of the function u according to

$$u = \bar{u} + \mathcal{H}\hat{u}, \tag{30.6}$$

where $\bar{u} : \Omega \to \mathbb{R}$ and $\hat{u} : \Omega \to \mathbb{R}$ are continuous functions, and $\mathcal{H}$ is the Heaviside function centered

at the surface across which u exhibits a jump. We use the terminology "continuous" loosely for now, and for simplicity we consider $\hat{u}$ to be defined everywhere in Ω. If $\mathcal{T}_h$ is a triangulation of the domain Ω, consider the finite element function spaces

$$\bar{V}_h = \left\{ \bar{v}_h \in H_0^1(\Omega) : \bar{v}_h|_T \in P_k(T) \ \forall \ T \in \mathcal{T}_h \right\}, \tag{30.7}$$

$$\hat{V}_h = \left\{ \hat{v}_h \in H_0^1(\hat{\Omega}) : \hat{v}_h|_T \in P_k(T) \ \forall \ T \in \hat{\mathcal{T}}_h \right\}, \tag{30.8}$$

where $\hat{\Omega}$ is the union of the supports of all finite element functions whose support is intersected by the surface Γ_d, $\hat{\mathcal{T}}_h$ in the restriction of $\mathcal{T}_h$ to $\hat{\Omega}$ and T is a finite element cell. A finite dimensional analogue of the decomposition in (30.6) reads

$$u_h = \bar{u}_h + \mathcal{H}\hat{u}_h, \tag{30.9}$$

where $\bar{u}_h \in \bar{V}_h$ and $\hat{u}_h \in \hat{V}_h$ ($\hat{u}_h = 0$ for $x \notin \hat{\Omega}$). Decomposing a test function v_h in the same manner, a Galerkin version of the variational problem in (30.5) reads: find $(\bar{u}_h, \hat{u}_h) \in \bar{V}_h \times \hat{V}_h$ such that

$$\begin{aligned} \int_\Omega \nabla \bar{u}_h \cdot \nabla \bar{v}_h \, dx + \int_{\hat{\Omega}_+} \nabla \hat{u}_h \cdot \nabla \bar{v}_h \, dx + \int_{\hat{\Omega}_+} \nabla \left(\bar{u}_h + \hat{u}_h \right) \cdot \nabla \hat{v}_h \, dx + \int_{\Gamma_d} q(\hat{u}_h) \hat{v}_h \, ds \\ = \int_\Omega f \bar{v}_h \, dx + \int_{\hat{\Omega}_+} f \hat{v}_h \, dx \quad \forall \ (\bar{v}_h, \hat{v}_h) \in \bar{V}_h \times \hat{V}_h, \end{aligned} \tag{30.10}$$

where $\hat{\Omega}_+ \subset \hat{\Omega}$ is the portion of $\hat{\Omega}$ on which $\mathcal{H} = 1$. When presenting finite element function spaces with discontinuities in Section 30.5, the more compact notation

$$V = \left\{ v_h \in H_0^1 \left(\Omega \backslash \Gamma_d \right), v_h|_T \in P_k \left(T \backslash \Gamma_d \right) \forall \, T \right\} \tag{30.11}$$

will be used. The implementation of the discontinuous spaces expressed using the above notation follows the approach described in this section.

In terms of finite element basis functions, equation (30.9) is expressed as follows:

$$u_h = \sum_i^n \bar{\phi}_i \bar{u}_i + \sum_j^m \mathcal{H} \hat{\phi}_j \hat{u}_j, \tag{30.12}$$

where $\bar{\phi}_i$ and $\hat{\phi}_j$ are the finite element basis functions associated with $\bar{V}_h$ and $\hat{V}_h$, respectively, and $\bar{u}_i$ and $\hat{u}_j$ are the regular and "enriched" degrees of freedom, respectively. Note that the usual interpolation property of finite element functions does not hold in the region of a discontinuity surface. In practice $m \ll n$.

There are a number of issues that make the generation of computer code for the extended finite element method more complex than for the conventional finite element method. A key point is that integration schemes must be evaluated at run-time since it is necessary to perform quadrature on both sides of a discontinuity surface for intersected cells. Moreover, for problems in which flux-like quantities are prescribed on discontinuity surfaces, it is necessary to integrate terms on the surface Γ_d. Another issue is that the number of degrees of freedom associated with each element is not constant and it depends on the location of the discontinuity surface, and in the case of an evolving discontinuity, this changes during a simulation. The variable number of cell degrees of freedom can make it difficult to generalize existing finite element solvers to support partition-of-unity methods.

This section has demonstrated the use of a Heaviside enrichment via the extended finite element method. It is possible to use other enrichment functions in combination with the extended finite

element method. Commonly, functions that span the near-tip solution in linear elastic fracture mechanics are used. The scope of our work is limited to the Heaviside function.

30.3 Software components

In automating the generation of extended finite element models, we build upon three key components from the FEniCS project. Firstly, the Unified Form Language (UFL, Chapter 17) is used to express variational statements. Particular use is made of the concept of "enriched" spaces and the UFL concept of a "restriction". The latter is the restriction of functions to a particular entity subdomain. To generate code for a finite element assembler (and additional helper functions), we develop extensions to the FEniCS Form Compiler (FFC, Chapter 11) for generating UFC-compliant code. Finally, re-usable tools for the implementation of extended finite element methods, including an interface layer to transfer enriched degree of freedom data to the generated code, function spaces and surface abstractions, are constructed upon DOLFIN (Chapter 10).

30.3.1 Form language

For the extended finite element method, it is necessary to define function spaces that are restricted to subdomains, to define functions which are restricted on discontinuity surfaces, and to define a measure for surfaces to facilitate integration on surfaces. UFL does not address these three issues explicitly, but it does provide the necessary abstractions that can be used to communicate representations of forms that involve discontinuous function spaces to a form compiler. We use features of UFL, but rely also on a form compiler to interpret the UFL representation appropriately. The UFL definition of a problem is therefore an incomplete definition, with the form compiler being relied upon to interpret various abstractions correctly. Form compilers that support UFL will not necessarily generate the required code.

We define a discontinuous function space by restricting (informally) a continuous function space by a measure `dc`. Motivated by equation (30.6), we wish to locally enrich a continuous function space with a space that contains a discontinuity. We do this by adding continuous and discontinuous function spaces to create an "enriched" space:

Python code

```
Ec = FiniteElement("Lagrange", tetrahedron, 2)
Ed = RestrictedElement(Ec, dc)
E  = Ec + Ed
```

In the above, `Ec` is a regular scalar Lagrange finite element on a tetrahedron of order two. `Ed` is the restriction of the space `Ec` to the subdomain $\hat{\Omega}$, and it will contain a discontinuity. The geometry of the surface across which functions are discontinuous will only be known at run-time, hence details of the restriction can only be determined then. The expression `E = Ec + Ed` creates an enriched finite element. A more classical context in which the enriched space concept in UFL is used is for element-wise bubble functions. We note that construction of discontinuous UFL function spaces in this manner is not unequivocal, but it is simple for the user. The exact details of how the spaces are constructed does have a dependency on the implementation. We feel that this is a limited price to pay in return for ease of use.

Once an enriched finite element is defined, functions can be defined on the enriched space. For example, enriched trial and test functions and an enriched coefficient function are defined by:

Python code

```
u = TrialFunction(E)
```

```
v = TestFunction(E)
f = Coefficient(E)
```

We can also restrict coefficients, test and trial functions defined on discontinuous space to the positive or negative side of a discontinuity. The `jump` and `average` of the function value of v across the surface are defined by:

Python code

```
jump(v) = v("+") - v("-")
avg(v)  = [v("+") + v("-")]/2
```

respectively.

For the rest, variational forms can be expressed just as they are for conventional problems using UFL. In addition to the usual UFL syntax, `dc` can be used to indicate integration of terms on a discontinuity surface. A number of complete examples of UFL input are presented in Section 30.5.

30.3.2 FFC extensions

To generate low-level code for an assembly library, extensions to FFC have been developed for performing tasks that are specific to the extended finite element method. Tasks that are specific to the extended finite element method are:

- Evaluation of element tensors for cells on which enriched functions are active;
- Evaluation of enriched finite element functions that do not satisfy the interpolation property; and
- Generation of DOLFIN wrapper classes to aid in the initialization of enriched function spaces and data corresponding to the enriched degrees of freedom using discontinuity surfaces.

30.3.3 Assembler and solver

The component for the assembly and solution of the finite element equations is developed in C++ and builds on DOLFIN. It is the most complex of the necessary extensions. The main tasks of the solver are:

- Management of data and tools related to the partition-of-unity method;
- Interaction with the code generated by the form compiler;
- Representation of surfaces;
- Extension of surfaces for evolving surface geometry; and
- Visualization of functions with discontinuities.

Some generic details of how these features are implemented are provided in the next section.

30.4 Design and implementation

30.4.1 Form compiler

A small number of Python modules have been developed that extend FFC for problems with discontinuities. Features that a specific to the extensions are:

- Generation of intermediate representations for forms with discontinuous function spaces;
- "Expansion transformer" to separate standard and enriched terms appearing in integrals if any coefficient defined on a discontinuous space exists; and
- Functions to to handle enriched entries in element tensors.

The extended form compiler simply imports FFC modules for the bulk of the functionality.

30.4.2 Interface between the generated code and the solver

The code generated by the extended form compiler conforms to the UFC specification, hence an assembly function that supports UFC can be used without modification. However, to evaluate various objects, such as element tensors, the generated code must be aware of the discontinuity surface. To support this within the framework of a UFC-compliant finite element assembler, the necessary UFC objects are constructed with a `GenericPUM` object, and they store a reference to this object. `GenericPUM` defines an abstract interface through which the generated code can retrieve necessary data from the solver library. `GenericPUM`, together with the UFC specification, therefore define the interface for interactions between generated code and the solver environment.

The member functions of `GenericPUM` provide four basic types of functionality. The first is degree of freedom manipulation, and specifically management of the enriched degrees of freedom associated with the partition-of-unity method. This includes the tabulation of enriched degrees of freedom and the local dimension of a cell tensor, which can change during a simulation (such as as when a discontinuity surface is extended). The second group includes member functions of `GenericPUM` interface that tabulate enrichment functions at points (such as the Heaviside function for problems that involve a discontinuity), which are needed when computing element tensors and when interpolating functions at cell vertices. The `GenericPUM` interface also provides functions for modified quadrature rules. This is essential when using non-polynomial enrichment functions, and in particular discontinuous functions. For discontinuous functions, it is important that a sufficient number of quadrature points are used on either side of a discontinuity surface. `GenericPUM` provides a function that indicates when modified quadrature is required on a given cell or facet, and it provides an interface for returning tailored quadrature schemes. Finally, the `GenericPUM` interface introduces some member functions to update data related to the enriched degrees of freedom when the discontinuity surfaces evolve.

The `GenericPUM` interface is abstract, hence the generated code is independent of various implementation details, such as the method by which a discontinuity surface is represented and the quadrature method on intersected cells.

30.4.3 Assembler and solver

The key classes that are developed upon DOLFIN are a concrete implementation of the `GenericPUM` interface and the representation of surfaces. To decouple the representation of surfaces from other implementation details, an abstract base class `GenericSurface` is defined. The `GenericSurface` interface provides various functions for querying a surface object, such as whether a surface intersects a cell. It also provides an interface for returning quadrature schemes on a surface, as this is intimately related to details of the surface representation. The use of the base class `GenericSurface` permits different surface representations to be used interchangeably with the generated code. Surface representation is an active area of research in the context of the extended finite element method (see Jäger et al. (2008), for example), and the `GenericSurface` interface permits a high degree of flexibility in this respect.

In addition to concrete implementations of `GenericPUM` and `GenericSurface`, a variety other low-level functionality is implemented for performing various geometry operations, such as subtriangulation of finite element cells that are intersected by a surface.

30.5 Examples

Examples are presented in this section to demonstrate usage of framework. These examples aim to illustrate the generality in terms modeling discontinuities in different equations using different basis functions. Low-level code is generated from the compiler input by running from the command-line:

Bash code

```
ffcpum -l dolfin foo.ufl
```

For each presented example the bilinear form a, the linear form L and a function space V are defined. The complete finite element problem then involves: find $u_h \in V$ such that

$$a(u_h, v_h) = L(v_h) \quad \forall\, v_h \in V. \tag{30.13}$$

For a nonlinear equation, the above is interpreted as the linearized problem that is solved within a Newton iteration.

30.5.1 H^1-conforming primal approach to the Poisson equation

As a canonical example, we present the Poisson equation in which the solution u is discontinuous across the surface Γ_d and the flux across the surface $q = k\,(u_+ - u_-)$, where k is a parameter. For a conforming approach, the relevant function space is

$$V = \left\{ v_h \in H_0^1\left(\Omega \backslash \Gamma_d\right),\, v_h|_T \in P_k\left(T \backslash \Gamma_d\right) \forall\, T \right\} \tag{30.14}$$

and the bilinear and linear forms read

$$a\left(u_h, v_h\right) = \int_{\Omega \backslash \Gamma_d} \nabla u_h \cdot \nabla v_h \, dx + \int_{\Gamma_d} k \left[\!\left[u_h\right]\!\right] \left[\!\left[v_h\right]\!\right] \, ds, \tag{30.15}$$

$$L\left(v_h\right) = \int_{\Omega} f v_h \, dx, \tag{30.16}$$

where f is a source term.

The form compiler input for this problem, in two dimensions, using linear Lagrange elements is presented in Figure 30.2. The code generated by the form compiler is used as input for a C++ solver. Discontinuity surfaces are defined in the solver environment. An extract of the C++ solver is shown in Figure 30.3. The C++ code is designed to following the DOLFIN style of mirroring mathematical abstractions and keeping the code compact. The code for the objects `Poisson::FunctionSpace`, `Poisson::BilinearForm` and `Poisson::LinearForm` is problem specific and has been generated by the form compiler, whereas the other elements appearing in Figure 30.3 are standard DOLFIN objects, unless prefaced with the `pum` namespace. Note that the function space in the code extract is initialized with `surfaces`, which is a container of `GenericSurface` objects. This is a convenience wrapper for a UFC function space, with `GenericPUM` being created internally from the surfaces, and then used to initialize the UFC objects. Note also the use of `pum::Function`, which is a subclass of `dolfin::Function` and implements primarily restrictions of discontinuous coefficient functions for use in forms, and interpolation of functions to cell vertices for use in post-processing.

Python code

```
# Define continuous and discontinuous spaces
elem_cont    = FiniteElement("Lagrange", triangle, 1)
elem_discont = RestrictedElement(elem_cont, dc)

# Create enriched space
element = elem_cont + elem_discont

# Create test and trail functions
v, u = TestFunction(element), TrialFunction(element)

# Interface flux parameter and source term
k = Constant(triangle)
f = Coefficient(elem_cont)

# Create linear and bilinear forms
a = inner(grad(u), grad(v))*dx + k*jump(u)*jump(v)*dc
L = f*v*dx
```

Figure 30.2: UFL input for the Poisson equation using a H^1-conforming method with a discontinuous solution across a surface.

A mesh with superimposed discontinuity surfaces and the computed solution contours for this problem on a unit square domain containing two disjoint discontinuity surfaces are shown in Figure 30.4. For this case, $f = 1$ and $k = 1$. Homogeneous Dirichlet boundary conditions are applied along the bottom edge ($y = 0$). The remaining boundaries are flux-free. The impact of the discontinuities on the computed solution contours can be seen clearly in Figure 30.4b.

30.5.2 $H(\mathrm{div})$-conforming mixed approach to Poisson's equation

FFC supports a range of $H(\mathrm{div})$ and $H(\mathrm{curl})$ elements that can be used in combination with other finite element types (Rognes et al., 2009). These elements can also be used within the context of the extended finite element method. To demonstrate this, UFL input for solving the Poisson equation with a discontinuity using the $H(\mathrm{div})$-conforming BDM element (see Brezzi et al. (1985a) and Chapter 3) for the flux and L^2-conforming Lagrange functions for the scalar field is presented. The relevant function spaces for this problem read

$$V_h = \left\{ \tau_h \in H\left(\mathrm{div}, \Omega \backslash \Gamma_d\right), \tau_h|_T \in \left(P_k\left(T \backslash \Gamma_d\right)\right)^d \forall\, T \right\}, \tag{30.17}$$

$$W_h = \left\{ \omega_h \in L^2\left(\Omega\right), \omega_h|_T \in P_{k-1}\left(T \backslash \Gamma_d\right) \forall\, T \right\}. \tag{30.18}$$

For homogeneous Dirichlet boundary conditions, the bilinear and linear forms read:

$$a(\sigma_h; u_h, \tau_h; \omega_h) = \int_{\Omega \backslash \Gamma_d} \sigma_h \cdot \tau_h - u_h(\nabla \cdot \tau_h) + (\nabla \cdot \sigma_h)\omega_h \, dx, \tag{30.19}$$

$$L(\tau_h; \omega_h) = \int_{\Omega} f \omega_h \, dx. \tag{30.20}$$

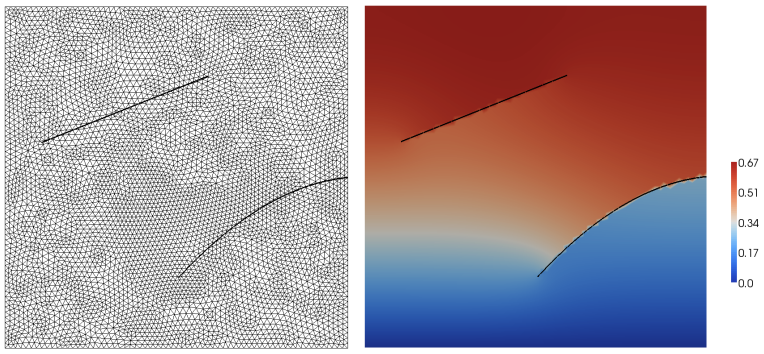
Unlike the H^1-conforming Poisson example, this example implies that $u_h = 0$ (weakly) on Γ_d since the terms $\int_{\Gamma_d} u_{h\pm} \tau_{h\pm} \cdot n_\pm \, ds$, which arise from integration by parts, have been discarded. This example therefore demonstrates how the extended finite element method can be used to apply Dirichlet-type conditions that do not conform to the mesh for a mixed-method.

The form compiler input for this problem is presented in Figure 30.5. Note the distinction between enriched and mixed elements. The usual BDM and piecewise-constant elements are "enriched" to incorporate a potential discontinuity (using the summation sign), and the enriched spaces are then

Figure 30.3: C++ code extract for solver of the Poisson problem with discontinuities in the solution. The notation resembles closely DOLFIN code for conventional problems.

C++ code

```
#include <dolfin.h>
#include <PartitionOfUnity.h>
#include "Poisson.h"

  . . .

int main()
{
  // Create mesh
  dolfin::UnitSquare mesh("mesh.xml.gz");
  . . .

  // Surface 0: A straight line (define by end points)
  std::pair<Point, Point> end_points0(p0_0, p0_1);
  pum::Surface d0(mesh, end_points0);

  // Surface 1: A curved line (define by end points and a
       level set function)
  std::pair<Point, Point> end_points1(p1_0, p1_1);
  const Shape1 shape1;
  pum::Surface d1(mesh, end_points1, shape1);

  // Add surfaces to an STL container
  std::vector<const pum::GenericSurface*>
       surfaces = boost::assign::list_of(d0)(d1);

  // Create function space with discontinuities across
       surfaces
  Poisson::FunctionSpace V(mesh, surfaces);

  // Create bilinear and linear Forms
  Poisson::BilinearForm a(V, V);
  a.k = k;
  Poisson::LinearForm L(V);
  L.f = f;

  // Create a linear variational problem and solve
  pum::Function u(V);
  solve(a == L, u, bcs);

  // Save solution to file for visualisation
  dolfin::File file("poisson.pvd");
  file  << u;

  // Save surfaces to file
  pum::VTKFile file_surface("surface.pvd");
  std::pair<std::vector<const GenericSurface*>,
            const dolfin::Mesh*> out_surfaces(surfaces,
                 &mesh);
  file_surface << out_surfaces;
}
```



(a) (b)

Figure 30.4: Poisson problem in two dimensions: (a) mesh and discontinuity surfaces and (b) solution contours.

combined to create a mixed finite element (using the multiplication sign).

30.5.3 L^2-conforming discontinuous Galerkin approach to linearized elasticity

FFC supports integration on interior facets, which makes it possible to generate code for discontinuous Galerkin methods (Ølgaard et al., 2008). For a discontinuous Galerkin interior penalty formulation of linearized elasticity with a discontinuity in the solution across Γ_d, the relevant function space reads

$$V = \left\{ v_h \in \left(L^2(\Omega) \right)^d, \, v_h|_T \in (P_k(T \backslash \Gamma_d))^d \; \forall \, T \right\}. \tag{30.21}$$

For homogeneous Dirichlet boundary conditions and traction-free discontinuity surfaces ($q = 0$), the bilinear form reads:

$$\begin{aligned} a(u_h, v_h) = \int_{\Omega \backslash \Gamma_d} \sigma(u_h) : \nabla v_h \, dx - \int_{\Gamma_0} [\![u_h]\!] \cdot \langle \sigma(v_h) \rangle n_+ \, ds - \int_{\Gamma_0} \langle \sigma(u_h) \rangle n_+ \cdot [\![v_h]\!] \, ds \\ + \int_{\Gamma_0} \frac{E\alpha}{h} [\![u_h]\!] \cdot [\![v_h]\!] \, ds - \int_{\partial\Omega} u_h \cdot \sigma(v_h) n \, ds \\ - \int_{\partial\Omega} \langle \sigma(u_h) \rangle n \cdot v_h \, ds + \int_{\partial\Omega} \frac{E\alpha}{h} u_h \cdot v_h \, ds, \end{aligned} \tag{30.22}$$

and the linear form reads:

$$L(v_h) = \int_\Omega f \cdot v_h \, dx, \tag{30.23}$$

where $\sigma(u) = \mu(\nabla u + (\nabla u)^T) + \lambda \mathrm{tr}(\nabla u) I$ is the stress tensor and μ and λ are Lamé parameters, n_+ is the unit outward normal to a cell facet from the '+' side, Γ_0 is the union of all interior cell facets, $\langle a \rangle = (a_+ + a_-)/2$ is the average operator on cell facets, $[\![b]\!] = b_+ - b_-$ is the jump operator on cell facets, E is Young's modulus, h is a measure of the cell size and $\alpha > 0$ is a dimensionless penalty parameter that is required for stability. The form compiler input for this problem is presented in Figure 30.6. The necessary operators at cell facets are implemented as part of UFL. Integration over interior and exterior facets is indicated by `*dS` and `*ds`, respectively.

30.5.4 Nonlinear Poisson-like equation

We now consider a nonlinear problem that is solved using Newton's method. This example demonstrates the application of the automatic differentiation feature of UFL for a problem involving

Figure 30.5: Form compiler input for the mixed Poisson problem with discontinuous u and σ.

Python code

```
# Define continuous (cell-wise) spaces
BDM_c = FiniteElement("Brezzi-Douglas-Marini", triangle, 1)
DG_c  = FiniteElement("Discontinuous Lagrange", triangle, 0)

# Define discontinuous spaces
BDM_d, DG_d = RestrictedElement(BDM_c, dc),
     RestrictedElement(DG_c, dc)

# Create enriched spaces
BDM, DG = BDM_c + BDM_d, DG_c + DG_d

# Create mixed element
mixed_element = BDM * DG

# Trial and test functions
sigma, u = TrialFunctions(mixed_element)
tau, w   = TestFunctions(mixed_element)

# Source term
f = Coefficient(DG_c)

# Bilinear form and linear forms
a = dot(sigma, tau)*dx - u*div(tau)*dx + div(sigma)*w*dx
L = f*w*dx
```

Figure 30.6: UFL representation of the discontinuous Galerkin formulation of linear elasticity equation with discontinuous u across Γ_d.

Python code

```
# Define continuous and discontinuous spaces
elem_cont    = VectorElement("DG", triangle, 2)
elem_discont = RestrictedElement(elem_cont, dc)
element      = elem_cont + elem_discont

# Create test and trial functions
v, u = TestFunction(element), TrialFunction(element)

# Compute material properties
E, nu     = 200000.0, 0.3
mu, lmbda = E / (2*(1 + nu)), E*nu / ((1 + nu) * (1 - 2*nu))

# Facet normal component, cell size and source term
n, h = element.cell().n, element.cell().circumradius
f = Coefficient(elem_cont)

# Penalty parameters
alpha = 4.0

# Stress
def sigma(v):
   return 2.0*mu*sym(grad(v)) \
  + lmbda*tr(sym(grad(v)))*Identity(v.cell().d)

# Bilinear form and linear forms
a = inner(sigma(u), grad(v))*dx \
  - inner(jump(u), avg(sigma(v))*n("+"))*dS \
  - inner(avg(sigma(u))*n("+"), jump(v))*dS \
  + (E*alpha/avg(h))*inner(jump(u), jump(v))*dS \
  - inner(u, sigma(v)*n)*ds \
  -inner(sigma(u)*n, v)*ds \
  + (E*alpha/h)*inner(u, v)*ds
L = inner(f, v)*dx
```

discontinuities. The Poisson-like equation

$$-\nabla \cdot \left(1 + u^2\right) \nabla u = f \tag{30.24}$$

on the domain Ω, with $u = 0$ on $\partial\Omega$ and a flux-free discontinuity surfaces ($q = \left(1 + u^2\right) \nabla u \cdot n = 0$), can be phrased in a variational format as: find $u_h \in V$ such that

$$F\left(u_h; v_h\right) \equiv \int_\Omega \left(1 + u_h^2\right) \nabla u_h \cdot \nabla v_h - f v_h \, dx = 0 \quad \forall \; v_h \in V, \tag{30.25}$$

where V is the space defined in equation (30.14). The functional F is linear in v_h but nonlinear in u_h. A nonlinear problem posed in this format can be solved using Newton's method, in which F is driven to zero by solving a series of linear systems until a prescribed tolerance is reached. The functional F, evaluated at the most recent approximation of u_h, serves as the 'linear form' (linear in v_h) and the Jacobian of F, evaluated at the most recent approximation of u_h, serves as the bilinear form (it is linear in v_h and du_h, where du_h is the correction to the solution which is computed by the Newton solver). Formally, the Jacobian is given by

$$a\left(u_h; du_h, v_h\right) = \left. \frac{dF\left(u_h + \epsilon du_h; v_h\right)}{d\epsilon} \right|_{\epsilon=0}. \tag{30.26}$$

Input to the form compiler will mirror this notation.

This example is relatively simple and the computation of the Jacobian by hand is not onerous. However, for complicated nonlinear equations, the derivation of an analytical expression for the Jacobian can be lengthy and error prone (which is compounded by the extra complexity of the extended finite element method). For this task, exact automatic differentiation is particularly attractive. Firstly, it eliminates a source of errors. Secondly, it means that if details of the equation of interest are changed, then there is not need to re-evaluate the Jacobian by hand. UFL provides functionality for automatic differentiation, and the directional derivative feature can be used to compute the Jacobian from a linear form, with the Jacobian filling the role of the bilinear form in the linearized system.

The UFL input for the problem in equation (30.25) is shown in Figure 30.7, where we have chosen to use quadratic Lagrange functions. The trial function in this case is the iterative correction du_h, and the most recent approximation of u_h is provided as a coefficient function to the forms. Following from equation (30.26), the UFL input for the bilinear form (Jacobian) reads:

Python code

```
a = derivative(L, u, du)
```

As with the other problems that we have presented, the generated code for the linear and bilinear forms can be used in a DOLFIN-based solver. An extract of the DOLFIN/C++ code for solving this nonlinear problem is presented in Figure 30.8.

30.6 Summary

It has been demonstrated that code for the extended finite element method can be generated through a form compiler. UFL provides the necessary functionality to describe abstractly variational problems, and when paired with a suitable form compiler computer code can be generated with the same ease as for conventional finite element problems. Automating aspects of the extended finite element method poses a number of challenges that do not feature in the conventional finite element method, such as adaptive quadrature on cells intersected by a discontinuity surface and a variable numbers

Figure 30.7: Form compiler input for the nonlinear Poisson-like equation. The bilinear form (Jacobian) follows from differentiation of the linear form L.

Python code

```
# Define continuous and discontinuous spaces
elem_cont    = FiniteElement("Lagrange", triangle, 2)
elem_discont = RestrictedElement(elem_cont, dc)
element      = elem_cont + elem_discont

# Create test and trial functions
v, du = TestFunction(element), TrialFunction(element)

# Latest solution and source term
u, f = Coefficient(element), Coefficient(elem_cont)

# Bilinear form and Linear form
L = (1.0 + u**2)*inner(grad(u), grad(v))*dx - f*v*dx
a = derivative(L, u, du)
```

Figure 30.8: C++ code extract for the nonlinear Poisson-like equation.

C++ code

```
// Create function space
NonlinearPoisson::FunctionSpace V(mesh, surfaces);

// Solution function
pum::Function u(V);

// Create linear form
NonlinearPoisson::LinearForm F(V);
F.u = u; F.f = f;

// Create jacobian dF = F' (for use in nonlinear solver).
NonlinearPoisson::BilinearForm dF(V, V);
dF.u = u;

// Solve nonlinear variational problem
solve(F == 0, u, bc, dF);

// Save solution in VTK format
dolfin::File file("nonlinear_poisson.pvd");
file << u;
```

of degrees of freedom at cells that are under the influence of a discontinuity. By automating the generation of large parts of the computer code, models that employ a variety of different finite element basis functions, some with and others without discontinuous enrichment, can be easily and rapidly developed. Moreover, the generated code and details of the adopted approach, such as the surface representation and modified quadrature on intersected cells, can be decoupled via suitably abstract class interfaces. This permits different implementation details to be employed without impacting on other aspects of the solver.

31 *Dynamic simulations of convection in the Earth's mantle*

By Lyudmyla Vynnytska, Stuart R. Clark and Marie E. Rognes

In this chapter, we model dynamic convection processes in the Earth's mantle; linking the geodynamical equations, numerical implementation and Python code tightly together. The convection of material is generated by heating from below with a compositionally distinct and denser layer at the bottom. The time-dependent nonlinear partial differential equations to be solved are the quasi-static Stokes equations with depth- and temperature-dependent viscosity, and advection–diffusion equations for the composition and temperature. We present a numerical algorithm for the simulation of these equations as well as an implementation of this algorithm using the DOLFIN Python interface. The results show that the compositional heterogeneities persist, but interact strongly with the convecting system, generating upwellings and movement as material from the surface displaces them. This chapter will be of interest to those seeking to model compositional discontinuities using field methods, as well as those interested in mantle convection simulations.

31.1 Convection in the Earth's mantle

In contrast to the hydrosphere and atmosphere, the Earth's crust and mantle are primarily solid in nature, allowing the rapid progression of seismic waves. However, one of the most important discoveries of geodynamics is that materials can behave elastically on certain timescales and viscously on other timescales. Glaciers work on this principle: solid ice slowly deforms and flows under the effect of gravity. While glaciers move on the order of meters per day, the mantle moves at the speed of a few centimeters per year (van der Meer et al., 2010). At this rate, a piece of the Earth's lithosphere, or slab, would take at least one hundred million years to sink from the Earth's surface to the outer core (Jarvis and Lowman, 2007).

Blanketing the outer core, seismologists detect a layer through which seismic waves are anomalously slow: the D″ (*dee* double prime) layer. In some regions, this layer is very thin, overlain by fast zones that may indicate slabs buried deep within the mantle (McNamara and Zhong, 2005). Underneath southern Africa and the Pacific, two prominent seismically anomalous slow regions exist, seemingly pointing to a hotter or compositionally denser material (McNutt, 1998). Such heterogeneities have led geoscientists to speculate on the existence of large chemically isolated reservoirs in the mantle, perhaps a remnant from the early Earth's mantle (Burke et al., 2008). But how can these chemically isolated reservoirs survive in a vigorously convecting mantle? Geodynamicists have tried to answer this question with computer simulations of thermomechanical convection of a compositionally heterogeneous mantle; such simulations are more simply termed thermochemical (McNamara et al., 2010). The challenge for geodynamicists is: can the assumptions made in matching the longevity

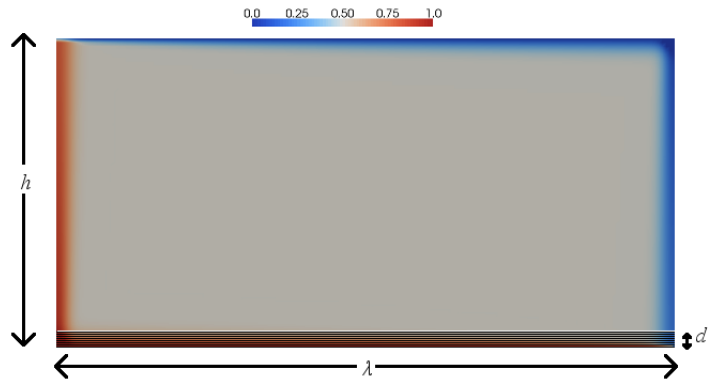


Figure 31.1: Initial configuration of the model in the h by λ rectangular box Ω. Color shows dimensionless temperature (for dimensional values, see Section 31.5). The shaded layer on the bottom with height d is the denser layer corresponding to the D″ layer.

of these reservoirs be consistent with seismic observations and the physics we know of the Earth's interior and plate tectonics?

Primarily, the observations we need to match are simply the transfer of matter between the Earth's interior and surface. At the surface of the Earth, tectonic plates are bent and pushed into the interior. In other places, we see large volcanic terranes created by material sourced from the Earth's mantle. To model the motion of the mantle over long timescales, the Stokes equations are well established in their ability to replicate this behavior, given the right assumptions, coupled with a conservation equation for the thermal energy in the mantle.

31.2 Mathematical statement of the problem

We model the problem in a rectangular box Ω in the Cartesian coordinate system with coordinates (x_1, x_2), neglecting the sphericity of the Earth, representing the mantle from the surface to the core-mantle boundary (see Figure 31.1). The base of the box is covered by a relatively thin layer of denser material, while the initial temperature field is set to represent the colder lithosphere along the top and slab descending on the right-hand side; a hotter layer is imposed along the bottom and left to represent the hotter D″ layer and a plume ascending respectively following the boundary layer arguments of van Keken et al. (1997). The idea is simply to create an initial configuration that drives the convection of the problem.

The viscosity of the mantle, on the order of 10^{22}Pa s, is so high that inertial forces and compressibility are negligible (Ricard, 2009). If the assumption of compressibility is relaxed, the lower mantle can support piles of geochemically isolated material with sharp edges (Tan and Gurnis, 2005). However, for our purposes, we assume an incompressible thermal flow driven by temperature and compositional density variations modeled by the following system of equations:

$$-\operatorname{div}\sigma' - \operatorname{grad} p = (Rb\,\phi - Ra\,T)\,e, \tag{31.1}$$

$$\operatorname{div} u = 0, \tag{31.2}$$

$$\frac{\partial T}{\partial t} + u \cdot \operatorname{grad} T = \Delta T. \tag{31.3}$$

Here, σ' is the deviatoric stress tensor, p is the pressure, Ra and Rb are the thermal and compositional Rayleigh numbers, respectively, T is the temperature, ϕ is a composition field, u is the velocity, and e is a unit vector in the direction of gravity $(-x_2)$. Equations given in this section are nondimensional; scaling and physical constants are presented in Section 31.5.

The Rayleigh numbers measure the relative importance of buoyancy to thermal and viscous dissipation given by thermal diffusivity (k_{th}) and the reference viscosity (η_0), respectively (Kennett

and Bunge, 2008). The change in density due to heat is a product of the thermal contrast (ΔT), thermal expansivity (α) and the reference density (ρ_0), while for the composition it is simply the density contrast ($\Delta\rho$) between the two materials; the gravitational acceleration (g) turns these densities into buoyancies in the following way:

$$Ra = \frac{\alpha g \rho_0 \Delta T h^3}{k_{th}\eta_0}, \quad Rb = \frac{\Delta\rho g h^3}{k_{th}\eta_0}. \tag{31.4}$$

The Rayleigh numbers, Ra and Rb, are defined as equal and as 10^6 within the ranges for the Earth (Montague et al., 1998) and such that fluid convection dominates.

The mantle flow induces transport of the composition ϕ. This transport is governed by the equation

$$\frac{\partial\phi}{\partial t} + u \cdot \operatorname{grad}\phi = 0. \tag{31.5}$$

However, some chemical diffusivity k_{ch} is also present in the physical system (van Keken et al., 1997; Hansen and Yuen, 1988). Therefore, we substitute the pure advection equation (31.5) by an advection–diffusion equation of the form

$$\frac{\partial\phi}{\partial t} + u \cdot \operatorname{grad}\phi = k_c \Delta\phi. \tag{31.6}$$

Here, $k_c = k_{ch}/k_{th}$.

It remains to specify the constitutive relationship relating the deviatoric stress tensor σ' to the other variables. Here, we consider the case of a Newtonian rheology with a depth- and temperature-dependent viscosity η (Blankenbach et al., 1989). The stress-strain relationship is then described by the equations

$$\sigma' = 2\eta\dot{\varepsilon}(u), \tag{31.7}$$

$$\dot{\varepsilon}(u) = \frac{1}{2}\left(\operatorname{grad} u + \operatorname{grad} u^\top\right), \tag{31.8}$$

$$\eta = \eta(T, x_2) = \eta_0 \exp\left(-bT/\Delta T + c(h - x_2)/h\right). \tag{31.9}$$

Here, $\dot{\varepsilon}$ is the strain rate tensor, η_0 is (still) the reference viscosity, and b and c are given additional parameters.

For the current scenario, we will consider the following boundary conditions. For the Stokes equations (31.1), we apply no slip conditions on the bottom boundary ($x_2 = 0$), and free slip and reflective symmetry on the remaining boundary $\Gamma = \partial\Omega \backslash \{x : x_2 = 0\}$:

$$u|_{x_2=0} = 0, \quad u_n|_\Gamma = \sigma'_{n\tau}|_\Gamma = 0, \tag{31.10}$$

where n is the outward normal and τ is the tangent vector on the boundary $\partial\Omega$. For the temperature, we fix its value on the top and bottom boundary and apply symmetry conditions (or no heat exchange) on the left- and right-hand boundaries:

$$T|_{x_2=0} = \Delta T, \quad T|_{x_2=h} = 0, \quad \partial_{x_1} T|_{x_1=0} = \partial_{x_1} T|_{x_1=\lambda} = 0. \tag{31.11}$$

For the composition ϕ, we set

$$\phi|_{x_2=0} = 1, \quad \partial_n \phi|_\Gamma = 0; \tag{31.12}$$

that is, we fix the composition on the bottom of the box Ω and apply zero flux conditions on the

remainder of the boundary. This condition can be viewed as a consequence of the no outflow conditions for the velocity.

The initial temperature field is given by T_0 (see Figure 31.1). In the below equations, we give an analytical expression for T_0 based on boundary layer theory (van Keken et al., 1997), taking the value of Ra with input from the upper T_u, lower T_l, right T_r and left T_s parts of the domain into account:

$$T_0 = T_u + T_l + T_r + T_s - 1.5, \tag{31.13}$$

$$q = \frac{\lambda^{7/3}}{(1+\lambda^4)^{2/3}} \left(\frac{Ra}{2\sqrt{\pi}} \right)^{2/3}, \tag{31.14}$$

$$Q = 2\sqrt{\frac{\lambda}{\pi q}}, \tag{31.15}$$

$$T_u = 0.5\text{erf}\left(\frac{1-x_2}{2} \sqrt{\frac{q}{x_1}} \right), \tag{31.16}$$

$$T_l = 1 - 0.5\text{erf}\left(\frac{x_2}{2} \sqrt{\frac{q}{\lambda - x_1}} \right), \tag{31.17}$$

$$T_r = 0.5 + \frac{Q}{2\sqrt{\pi}} \sqrt{\frac{q}{x_2+1}} \text{expc}\left(-\frac{x_1^2 q}{4x_2+4} \right), \tag{31.18}$$

$$T_s = 0.5 - \frac{Q}{2\sqrt{\pi}} \sqrt{\frac{q}{2-x_2}} \text{expc}\left(-\frac{(\lambda - x_1)^2 q}{8-4x_2} \right). \tag{31.19}$$

In order to keep the initial temperature distribution in the range between zero and one, we perform an additional correction: according to the above equations there are two peaks, a positive peak in the top right corner and a negative peak in the bottom left corner. However, we map all values below zero to zero and above one to one. Furthermore, the initial composition ϕ is a step function equal to one in the bottom layer and to zero in the top layer.

31.3 Numerical method

In this section, we present a numerical solution method for the thermochemical convection model established in the previous. Instead of solving the fully coupled system of nonlinear time-dependent partial differential equations, we consider a predictor–corrector based splitting scheme (van den Berg et al., 1993; Hansen and Ebel, 1988). In view of this, we present numerical methods for the solution of each separate equation before formulating the complete algorithm. Special attention must be paid to the discretization of the energy (31.3) and transport equations (31.6) due to the temperature gradient and the interface between the compositionally distinct layers. For the Stokes equations (31.1) – (31.2), we use a mixed finite element formulation, thus obtaining solutions for the velocity and pressure simultaneously.

31.3.1 Mixed finite element formulation of the Stokes equations

Let $\mathcal{T}_h = \{T\}$ be a mesh[1] of the domain Ω. Let V_h and Q_h be finite dimensional spaces, defined relative to the mesh $\mathcal{T}_h$, for the velocity and pressure fields, respectively. The standard discrete mixed finite element formulation with independent approximation of the velocity field u and the pressure field p for the incompressible Stokes equations (31.1) – (31.2) with the boundary conditions given

[1]Note that T is used to denote an element (cell) in a mesh in this section, in addition to being used to denote the continuous temperature field T.

by (31.10) reads as follows: for a given temperature T_h and composition ϕ_h, find $u_h \in V_h$ and $p_h \in Q_h$ such that

$$a_{IS}((u_h, p_h), (v_h, q_h)) = L_{IS}((v_h, q_h)) \tag{31.20}$$

for all $v_h \in V_h$ and all $q_h \in Q_h$, where

$$a_{IS}((u_h, p_h), (v_h, q_h)) = \int_\Omega 2\eta \dot{\varepsilon}(u_h) \cdot \dot{\varepsilon}(v_h) + p_h \operatorname{div} v_h + q_h \operatorname{div} u_h \, \mathrm{d}x, \tag{31.21}$$

$$L_{IS}((v_h, q_h)) = \int_\Omega (RaT_h - Rb\phi_h)e \cdot v_h \, \mathrm{d}x. \tag{31.22}$$

In the subsequent simulations, we use the lowest order Taylor–Hood elements for the velocity and the pressure; that is, the combination of continuous piecewise quadratic vector fields for V_h and continuous piecewise linears for Q_h (Taylor and Hood, 1973).

31.3.2 Discontinuous Galerkin formulation of advection–diffusion equations

The energy and transport equations (31.3) and (31.6) have the same structure from the mathematical point of view. The equations both model the time evolution of advection–diffusion processes, allowing the numerical analysis to be performed by the same numerical scheme. However, the numerical treatment of advection dominated advection–diffusion equations is nontrivial. There exists a large body of research on the development of efficient computational schemes for such kinds of problems (Lin, 2006; Zienkiewicz and Taylor, 2000). Within the finite element setting, there are two main approaches: Petrov-Galerkin approximation and discontinuous Galerkin methods. Here, we prefer a discontinuous Galerkin method because it deals effectively with discontinuous property fields. In the following, we describe an upwinded discontinuous Galerkin formulation for the equation (31.6) for the compositional field ϕ. This formulation also applies for the energy equation (31.3) with $k_c = 1$. For the sake of clarity, we consider the discretization of (31.5) separately first.

Using full upwind numerical flux and taking into consideration that the normal component of the velocity is equal to zero on the boundary, the spatial, discontinuous finite element discretization of (31.5) with a given u_h reads as (di Pietro et al., 2006): find $\phi_h \in P_h$ such that

$$\sum_{T \in \mathcal{T}_h} \int_T \frac{\partial \phi_h}{\partial t} \psi_h \, \mathrm{d}x + a_A(u_h; \phi_h, \psi_h) = 0, \tag{31.23}$$

for all $\psi_h \in P_h$, where

$$a_A(u_h; \phi_h, \psi_h) = -\sum_{T \in \mathcal{T}_h} \int_T \phi_h u_h \cdot \operatorname{grad} \psi_h \, \mathrm{d}x + \sum_{e \in \Gamma_i} \int_e \left(u_h \cdot [\![\psi_h]\!] \langle \phi_h \rangle + \frac{1}{2} |u_h \cdot n| [\![\psi_h]\!] [\![\phi_h]\!] \right) \mathrm{d}s, \tag{31.24}$$

wherein Γ_i denotes the interior edges of $\mathcal{T}_h$. The jump $[\![\cdot]\!]$ and average $\langle \cdot \rangle$ operators on an internal edge shared by elements T^+ and T^- with outward normals n^+ and n^- respectively, are defined for generic scalar fields α and vector fields β as

$$[\![\alpha]\!] = \alpha^+ n^+ + \alpha^- n^-, \quad [\![\beta]\!] = \beta^+ \cdot n^+ + \beta^- \cdot n^-, \tag{31.25}$$

$$\langle \alpha \rangle = \frac{1}{2} \left(\alpha^+ + \alpha^- \right), \quad \langle \beta \rangle = \frac{1}{2} \left(\beta^+ + \beta^- \right), \tag{31.26}$$

$$\alpha^\pm = \alpha|_{T^\pm}, \quad \beta^\pm = \beta|_{T^\pm}. \tag{31.27}$$

We now turn to consider the diffusive term of (31.6) separately. Its standard variational form for a symmetric discontinuous Galerkin discretization with a stabilization penalty term is given by (Arnold,

1982):

$$a_D(\phi_h, \psi_h) = \sum_{T \in \mathcal{T}_h} \int_T k_c \operatorname{grad} \phi_h \cdot \operatorname{grad} \psi_h \, \mathrm{d}x + \sum_{e \in \Gamma_i} \int_e -\langle k_c \operatorname{grad} \phi_h \rangle \cdot [\![\psi_h]\!] \, \mathrm{d}s + \sum_{e \in \Gamma_i} \int_e \left(-\langle k_c \operatorname{grad} \psi_h \rangle \cdot [\![\phi_h]\!] + \frac{\alpha_h k_c}{h_T} [\![\phi_h]\!] \cdot [\![\psi_h]\!] \right) \mathrm{d}s, \quad (31.28)$$

where α_h is a sufficiently large constant to ensure stability, and h_T is the size of element T. Combining (31.24) and (31.28), we obtain the following spatially discrete variational formulation of the transport equation (31.6): find $\phi_h \in P_h$ such that

$$\sum_{T \in \mathcal{T}_h} \int_T \frac{\partial \phi_h}{\partial t} \psi_h \, \mathrm{d}x + a_A(u_h; \phi_h, \psi_h) + a_D(\phi_h, \psi_h) = 0 \quad (31.29)$$

for all $\psi_h \in P_h$, where P_h is a finite element space of discontinuous piecewise polynomial fields, and correspondingly for the temperature T_h. In the subsequent simulations, we will let P_h be the space of (discontinuous) piecewise linears defined relative to the mesh $\mathcal{T}_h$.

31.3.3 A decoupling predictor–corrector scheme

Instead of solving the fully coupled nonlinear system of equations defined by (31.1) – (31.2), (31.3), and (31.6), we use a splitting scheme. In particular, we consider a predictor–corrector scheme (van den Berg et al., 1993; Hansen and Ebel, 1988) for the temperature T in combination with a filtering algorithm for the composition ϕ. The filtering algorithm is aimed at correcting the compositional field from numerical diffusion and dispersion errors, and is motivated and described in detail by Lenardic and Kaula (1993).

Before outlining the algorithm, we make some comments on the temporal discretization of (31.29) and the corresponding equation for the temperature. Rewriting (31.29) as

$$\frac{\partial r}{\partial t} = W, \quad (31.30)$$

the common θ-scheme for the evolution of r from r^{k-1} to r^k with time step Δt_k reads:

$$\frac{r^k - r^{k-1}}{\Delta t_k} = \theta W^k + (1 - \theta) W^{k-1}. \quad (31.31)$$

The choice $\theta = 1$ corresponds to the backward Euler scheme while $\theta = 0.5$ corresponds to the Crank–Nicolson scheme. The predictor–corrector scheme draws on the Crank–Nicolson scheme in using a two-step procedure. Taking the energy equation for the temperature as an example, assuming that the temperature at the previous time T_h^{k-1} and the previous velocity u_h^{k-1} are known, the predictor step computes a predicted temperature $T_h^{pr} \in P_h$ solving the backward Euler equations for all $\psi_h \in P_h$:

$$\sum_{T \in \mathcal{T}_h} \int_T \frac{T_h^{pr} - T_h^{k-1}}{\Delta t_k} \psi_h \, \mathrm{d}x + a_A(u_h^{k-1}; T_h^{pr}, \psi_h) + a_D(T_h^{pr}, \psi_h) = 0, \quad (31.32)$$

where a_A and a_D are defined by (31.24) and (31.28), respectively. Later, the corrector step computes

the corrected temperature T_h^k by a Crank–Nicolson scheme

$$\sum_{T\in\mathcal{T}_h}\int_T \frac{T_h^k - T_h^{k-1}}{\Delta t_k}\psi_h \,\mathrm{d}x + 0.5\left(a_A(u_h^{pr}; T_h^k, \psi_h) + a_D(T_h^k, \psi_h)\right) + 0.5\left(a_A(u_h^{k-1}; T_h^{k-1}, \psi_h) + a_D(T_h^{k-1}, \psi_h)\right) = 0, \quad (31.33)$$

but using a predicted velocity u_h^{pr}, which will be further specified in Algorithm 9 below.

Algorithm 9 A predictor–corrector algorithm

Initialize temperature T^0 and composition ϕ^0.
Compute initial velocity u^0 by solving (31.20) with T^0 and ϕ^0.
Compute time step Δt_1 from u^0 according to (31.34).
for $k = 1, \ldots, n$.
(1) Solve (31.32) to obtain T_h^{pr}.
(2) Solve (the composition equivalent of) (31.32) to obtain ϕ_h^{pr}.
(3) Filter the predicted composition ϕ_h^{pr} to obtain ϕ_h^k.
(4) Solve (31.20) with T_h^{pr} and ϕ_h^k as input to obtain u_h^{pr}.
(5) Solve (31.33) to obtain T_h^k.
(4) Solve (31.20) with T_h^k and ϕ_h^k as input to obtain u_h^k.
(6) Compute the new time step Δt_{k+1} according to (31.34).
end for

In order to satisfy a CFL-type stability condition, a variable time step will be used. In particular, we define each time step Δt_k by the formula

$$\Delta t_k = C_{\mathrm{CFL}} h_{\min} / \max |u_h^{k-1}|, \quad (31.34)$$

where $h_{\min}$ is the minimum cell size of the mesh and C_{CFL} is a chosen positive number.

The basic idea of the filtering algorithm is to ensure that ϕ remains within the bounds $0 \leqslant \phi \leqslant 1$, and to minimize dispersion error. We refer the reader to Lenardic and Kaula (1993) for the detailed explanation and here give the outline of the algorithm for a discrete property field $\phi = \{\phi_i\}$.

Algorithm 10 A property filtering algorithm

(1) Compute the initial sum S_0 of all values of ϕ.
(2) Find the minimal value $\phi_{\min}$ below 0.
(3) Find the maximal value $\phi_{\max}$ above 1.
(4) Set $\phi_i = 0$ if $\phi_i \leqslant |\phi_{\min}|$.
(5) Set $\phi_i = 1$ if $\phi_i \geqslant 2 - \phi_{\max}$.
(6) Compute the sum S_1 of all values of ϕ.
(7) Compute the number num of $0 < \phi_i < 1$.
(8) Add $dist = (S_1 - S_0)/num$ to all $0 < \phi_i < 1$.

31.4 Implementation

31.4.1 Main algorithm

The main body of the implementation consists of the temporal loop defined in Algorithm 9. An abridged version of this code is listed in Figure 31.2, and explained in detail in the corresponding caption. As the filtering step is straightforward to implement based on the algorithm described in Algorithm 10, we will not comment any further on this aspect. Below, we make some general observations and comments.

- In each iteration, five variational problems are solved. First, a predicted temperature is computed based on the velocity and the temperature from the previous time step. Next, a tentative composition is computed based on the velocity and the composition from the previous time step; then filtered as described by Algorithm 10. With the filtered composition and the predicted temperature, the velocity and pressure are predicted. The corrected temperature is then calculated using the predicted velocity and the velocity and the temperature from the previous time step. The Stokes system is then solved again, this time with the filtered composition and the corrected temperature as input to yield the corrected velocity and pressure at this time step.

- The advection–diffusion problems for calculating the predicted composition and the predicted and corrected temperature depend on the velocity at the previous and current time steps. Analogously, the Stokes equations for the predicted and corrected velocity depend on the composition and temperature through the viscosity and the source terms. Thus, the linear systems of equations have to be assembled (and solved) at each time. For simplicity, we generate the variational forms describing each of these problems at each time. The compute time used for this is insignificant in comparison with the time required for the assembly and solution of the linear systems.

- The linear systems resulting from the equations for the composition and the temperature are positive definite but not symmetric. Hence, these are solved iteratively using a standard generalized minimal residual solver (GMRES). For the simulations considered in the subsequent section, these solvers converge in $4 - 10$ iterations. On the other hand, the linear systems resulting from the Stokes equations are symmetric but indefinite. Non-preconditioned iterative solvers typically fail to converge for such systems, while direct solvers are prohibitively (memory) expensive. Therefore, these systems require preconditioning. Following Chapter 35, we here take advantage of a standard Stokes preconditioner, neglecting possible advantages in using a preconditioner that varies synchronously with the viscosity. As such, we can assemble the preconditioner matrix outside the loop and reuse it and the Krylov solver in each iteration.

- The time step `dt` is computed adaptively in each iteration of the temporal loop using the formula (31.34). The minimal mesh size is easily extracted using `mesh.hmin()` and the maximal value for the velocity is extracted as the maximal degree of freedom from the `numpy` array of degrees of freedom. Since the time step varies in each iteration and with the mesh size, it is convenient to use a `Constant` for its representation in order to avoid recompilation of the variational forms at each iteration.

- The solutions for the composition, the temperature and the velocity are stored at each time using the `TimeSeries` class; allowing for easy storage and retrieval of meshes and solution vectors. Moreover, it naturally encourages a decoupling of the simulation and the post-processing of the simulation data. This can be highly advantageous, especially for computations with significant run-times.

In the subsections below, we discuss the definition of each of the variational forms and problems and an implementational structure for these.

31.4.2 Variational formulation of the Stokes equations

The mixed variational formulation for the Stokes equations is classical and listed in Figure 31.3. The definition of the bilinear and linear forms rely only on the element space, a source vector field $f = (RaT_h - Rb\phi_h)e$ and the viscosity η, see (31.20). The viscosity itself is temperature- and depth-dependent, but crucially not velocity dependent. Thus the bilinear and linear forms can be represented by standard linear formulations. Since the preconditioner for this system relies on the same function spaces and basis functions, we define the form for the preconditioner together with the variational forms describing the partial differential equation.

31.4.3 Variational formulation of advection–diffusion equations

In the implementation of the variational forms for the advection–diffusion problems, we emphasize the following: first, the variational forms defining the predictor step for the temperature and the composition are the same (modulo the diffusivity constant and possibly the penalty constant). Second, the predictor and the corrector steps for the temperature involve the same mathematical building blocks. Third, discontinuous Galerkin methods often involve quite a number of terms and the combined forms may easily become intractable. In view of these aspects, a minimal (as in highly reused) code close to mathematical syntax seemed preferable.

To this end, the implementation mimics the structure defined by (31.24), (31.28), and (31.31). The pure advection form a_A and the pure diffusion form a_D are defined through separate Python functions. The code for these are listed in Figures 31.4 and 31.5 and explained in the captions of those figures. The implementation of the weak forms of the predictor equation for the composition, and the predictor and corrector equations for the temperature are then built using these basic functions. The code for the corrector equation is included and discussed in Figure 31.6. The code for the predictor equations is similar but simpler and not discussed here.

31.5 Results

In this section we present the results of the thermochemical convection simulation, calculated on a mesh constructed by dividing the domain into 160×80 squares. Each square is split into two triangles by the right diagonal. The C_{CFL} time step parameter in (31.34) is 0.5, while $b = \ln(2.5)$ and $c = \ln(2.0)$ in (31.9).

The equations presented in Section 31.2 are dimensionalised using the physical parameters in Table 31.1. Scaling parameters for time t_s and velocity u_s are obtained as follows:

$$t_s = \frac{h^2}{k_{th}}, \quad u_s = \frac{h}{t_s}. \tag{31.35}$$

Convection in the model domain is determined by the kinematic energy of the fluid, given by:

$$E_K = \frac{1}{2} \int_\Omega \rho \|u\|^2 dx. \tag{31.36}$$

Since the variation in density is small, ρ can be taken out of the integral, and by defining the

Python code

```
...

# Functions at previous timestep (and initial conditions)
(phi_, T_, u_, P) = compute_initial_conditions(W, Q)

# Containers for storage
velocity_series = TimeSeries("bin/velocity")
temperature_series = TimeSeries("bin/temperature")
density_series = TimeSeries("bin/density")

# Solver for the Stokes systems
solver = KrylovSolver("tfqmr", "amg")
...
while (t <= finish):

    # Solve for predicted temperature
    (a, L) = energy(Q, Constant(dt), u_, T_)
    solve(a == L, T_pr, T_bcs,
          solver_parameters={"linear_solver": "gmres"})

    # Solve for predicted phi
    (a, L) = transport(Q, Constant(dt), u_, phi_)
    solve(a == L, phi_pr, bc,
          solver_parameters={"linear_solver": "gmres"})

    # Filter predicted phi (in place)
    filter_properties(phi_pr)
    phi.assign(phi_pr)

    # Solve for predicted velocity
    eta = viscosity(T_pr)
    (a, L, precond) = momentum(W, eta,
                               (Ra*T_pr - Rb*phi_pr)*g)
    (A, b) = assemble_system(a, L, bcs)
    solver.set_operators(A, P)
    solver.solve(velocity_pressure.vector(), b)
    u_pr.assign(velocity_pressure.split()[0])

    # Solve for corrected temperature T
    (a, L) = energy_correction(Q, Constant(dt), u_pr, u_, T_)
    solve(a == L, T, T_bcs,
          solver_parameters={"linear_solver": "gmres"})

    # Solve for corrected velocity
    eta = viscosity(T)
    (a, L, precond) = momentum(W, eta,
                               (Ra*T - Rb*phi)*g)
    (A, b) = assemble_system(a, L, bcs)
    solver.set_operators(A, P)
    solver.solve(velocity_pressure.vector(), b)
    u.assign(velocity_pressure.split()[0])

    # Store stuff
    store(phi, T, u, t)

    # Compute time step
    dt = compute_timestep(u)

    # Move to new timestep and update functions
    phi_.assign(phi)
    T_.assign(T)
    u_.assign(u)
    t += dt
```

Figure 31.2: Abridged code for the main predictor–corrector algorithm, see Algorithm 9. The initialization of the mesh `mesh`, the viscous and chemical parameters and the boundary conditions are omitted. The solution fields are consistently named `T`, `u`, and `phi` for solutions at the current time; `T_`, `u_`, and `phi_` for fields at the previous time; and `T_pr`, `u_pr`, and `phi_pr` for predictor fields. These `Functions` are all initialized (but the initialization is not included here). First, the initial conditions are constructed. This involves the solution of a Stokes system for the initial velocity `u_`, and in particular the construction of a preconditioner matrix `P`. Since the matrix is to be reused, the initial condition computation also return this matrix. Next, `TimeSeries` objects are initialized for easy storage of the solutions at each time. The `KrylovSolver` can also be reused in each iteration and is therefore created outside the loop. The contents of the loop follow steps (1) – (6) of Algorithm 9. First, the predicted temperature `T_pr` must be computed. This is here accomplished in 2 substeps: the forms for the variational problem are created and then the variational problem is solved using the free function `solve`. Next, the same steps are performed for the predicted composition `phi_pr`. The predicted composition is then filtered (in place) and also assigned to `phi`. Using the predicted temperature and filtered composition, the source term and viscosity for the Stokes equations are defined, and then the variational Stokes form is created. In order to retain the symmetry of the matrix under the application of Dirichlet boundary conditions, the linear system is assembled and solved explicitly. This consists of calling the method `assemble_system`, updating the `KrylovSolver` with the current operators `A` and `P`, and applying the solver to the right-hand side vector. The procedure is repeated for the corrected temperature and again for the corrected Stokes system. Finally, the solution fields at the current time are stored, the new time step `dt` is computed based on the current velocity and the current solutions are assigned to the previous time as we step forward in time.

Figure 31.3: Definition of the variational forms for the Stokes equations and the corresponding preconditioner. The equation is phrased in the style $F(\cdot, \cdot) = 0$, and the form for the preconditioner is defined using the same basis functions. The left- and right-hand side of the form F are extracted using the UFL functions `lhs` and `rhs`.

Python code

```python
def momentum(W, eta, f):

    # Define basis functions
    (u, p) = TrialFunctions(W)
    (v, q) = TestFunctions(W)

    # Define equation F((u, p), (v, q)) = 0
    F = (2.0*eta*inner(strain(u), strain(v))*dx
         + div(v)*p*dx
         + div(u)*q*dx
         + inner(f, v)*dx)

    # Define form for preconditioner
    precond = inner(grad(v), grad(u))*dx + q*p*dx

    # Return left and right hand side and preconditioner
    return (lhs(F), rhs(F), precond)
```

Figure 31.4: Definition of an upwinded discontinuous Galerkin formulation of the advection term. The input consists of: `phi` and `psi` (typically functions or basis functions); a given velocity `u`; a facet normal `n`; and an optional scalar multiplier `theta`. The absolute value of the normal component of `u` is defined and the contributions from the cell integrals and facet integrals are defined in accordance with (31.24). The sum of the contributions (multiplied by theta) is returned.

Python code

```python
def advection(phi, psi, u, n, theta=1.0):

    # Define |u * n|
    un = abs(dot(u('+'), n('+')))

    # Contributions from cells
    a_cell = - dot(u*phi, grad(psi))*dx

    # Contributions from interior facets
    a_int = (dot(u('+'), jump(psi, n))*avg(phi)
             + 0.5*un*dot(jump(phi, n), jump(psi, n)))*dS

    return theta*(a_cell + a_int)
```

Figure 31.5: Definition of a discontinuous Galerkin formulation of the diffusion term. The input consists of: `phi` and `psi` (typically functions or basis functions); the diffusivity constant `k_c`; a stabilization parameter `alpha`; a facet normal `n`; the cell size `h`; and an optional scalar multiplier `theta`. The contributions from the cell integrals and facet integrals are defined in accordance with (31.28). The sum of the contributions (multiplied by theta) is returned.

Python code

```python
def diffusion(phi, psi, k_c, alpha, n, h, theta=1.0):

    # Contribution from the cells
    a_cell = k_c*dot(grad(phi), grad(psi))*dx

    # Contribution from the interior facets
    tmp = (alpha('+')/h('+')*dot(jump(psi, n), jump(phi, n))
           - dot(avg(grad(psi)), jump(phi, n))
           - dot(jump(psi, n), avg(grad(phi))))*dS
    a_int = k_c('+')*tmp

    return theta*(a_cell + a_int)
```

Python code

```
def energy_correction(Q, dt, u, u_, T_):

    # Define test and trial functions
    T = TrialFunction(Q)
    psi = TestFunction(Q)

    # Diffusivity constant
    k_c = Constant(1.0)

    # Constants associated with DG scheme
    alpha = Constant(50.0)
    mesh = Q.mesh()
    h = CellSize(mesh)
    n = FacetNormal(mesh)

    # Define discrete time derivative operator
    def Dt(T):
        return backward_euler(T, T_, dt)

    # Add syntactical sugar for a_A and a_D
    def a_A(u, T, psi):
        return advection(T, psi, u, n, theta=0.5)
    def a_D(T, psi):
        return diffusion(T, psi, k_c, alpha, n, h, theta=0.5)

    # Define form
    F = (Dt(T)*psi*dx
         + a_A(u, T, psi) + a_A(u_, T_, psi)
         + a_D(T, psi) + a_D(T_, psi))

    return (lhs(F), rhs(F))
```

Figure 31.6: Definition of variational forms for one correction step for the temperature, see (31.33). The input is the element space `Q`, the time step `dt`, two given velocities `u` and `u_`, and (typically) a previous temperature `T_`. We define the unknown temperature `T` and the test function `psi`. The diffusivity constant `k_c` is in this case 1.0. We define the penalty parameter `alpha` and also the cell normal `n` and mesh size `h` to be used in the advection and diffusion forms. We use Python functions to reduce the number of input arguments to the advection and diffusion functions. This is in part merely syntactical, but does also increase readability and facilitates debugging. The input to the functions `a_A` and `a_D` thus directly corresponds to the arguments of a_A and a_D. The equation is again phrased in the style $F(\cdot,\cdot) = 0$. The reader is encouraged to compare the code with the mathematical formulation of the equation, see (31.33). Finally, the left- and right-hand side forms are extracted using the UFL functions `lhs` and `rhs`.

Parameter		Dimensional value	Dimensionless value
Box height	h	3000km	1.0
Box length	λ	6000km	2.0
Boundary layer thickness	d	150km	0.05
Acceleration due to gravity	g	$10\mathrm{m/s^2}$	1.0
Thermal contrast	ΔT	3000K	1.0
Thermal expansivity	α	$2 \times 10^{-5}\mathrm{K}^{-1}$	
Thermal diffusivity	k_{th}	$10^{-6}\mathrm{m^2/s}$	
Chemical diffusivity	k_{ch}	$10^{-10}\mathrm{m^2/s}$	
Reference density	ρ_0	$3100\mathrm{kg/m^3}$	
Compositional density contrast	$\Delta\rho$	$185\mathrm{kg/m^3}$	1.0
Reference viscosity	η_0	$5 \times 10^{22}\mathrm{Pa\,s}$	1.0
Thermal Rayleigh number	Ra	1×10^6	1×10^6
Chemical Rayleigh number	Rb	1×10^6	1×10^6
Velocity	u_s	$3 \times 10^{-13}\mathrm{m/s}$	1.0
Time	t_s	$1 \times 10^{19}\mathrm{s}$	1.0

Table 31.1: Specification of parameters and parameter values.

root-mean square velocity, u_{rms} by:

$$u_{\mathrm{rms}} = \left(\frac{1}{\lambda h} \int_\Omega \|u\|^2 dx \right)^{1/2}, \tag{31.37}$$

we have the relationship:

$$E_K \approx \frac{1}{2} \rho \lambda h u_{\mathrm{rms}}^2. \tag{31.38}$$

Thus, u_{rms} is a scaled measure of the kinematic energy of convection. For the discussion of the results, we will use Figure 31.7 to describe the local turning points, A to M, of the root-mean square velocity and refer to the driving forces in the model to explain these turning points. See the captions to Figures 31.8 and 31.9 for details.

31.6 Discussion and concluding remarks

The presented results show the mantle convecting with two distinct chemical layers: plumes arise from atop piles on the denser bottom layer, but are not compositionally distinct. The location of these piles is initially set by the thermally dense slabs. Slabs collapsing into the mantle drive the largest changes in the system energy, while plumes drive smaller increases because their composition counteracts their thermal buoyancy. The upwellings and downwellings react: slabs rapidly sinking cause upwellings to form; the lateral motion of upwellings at the top pushes and thickens the top layer, causing it to become unstable and sink. As the system evolves, colder slab material builds up at the bottom, increasing the viscosity of the lower mantle, while the reverse happens in the upper mantle.

The simulation code for the convecting mantle has been included almost in its entirety. We can conclude that the amount of code required to implement such a problem within the FEniCS framework is quite small. Moreover, the code for the variational problems closely matches the mathematical

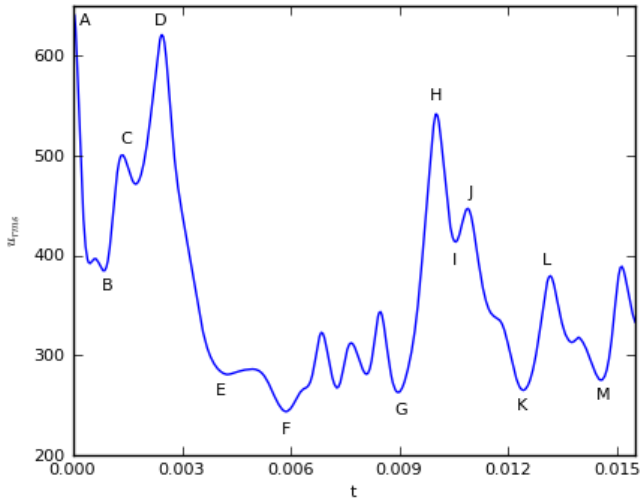


Figure 31.7: Dimensionless root mean square velocity over dimensionless time. A to M are labels used to refer to stages in the model. A nondimensional time period of 0.001 corresponds to approximately 300 Ma, and the dimensional value for $u_{\text{rms}} = 100$ is approximately 1 mm/year.

formulation of the problems, and thus the complexity of the code scales with the complexity of the numerical algorithm. The numerical simulations presented here are spatially two-dimensional and serve as a simplified model. Realistic three-dimensional simulations would require taking advantage of the parallel, and possibly more sophisticated adaptive, features of the FEniCS Project. However, such would not require significant additional problem-specific implementational effort, though preconditioning would have to be carefully considered.

Tracking composition in the problem requires the solution of the compositional advection–diffusion equation (31.6), creating difficulties for standard field-based methods to solve because of sharp discontinuities, often numerically smoothed by such methods. Tracers and marker chain approaches are often used to overcome this problem (Ismail-Zadeh and Tackley, 2010). However, the approach presented here allows us to represent compositional heterogeneities because it employs discontinuous Galerkin methods, while a filtering scheme minimizes the numerical smoothing error.

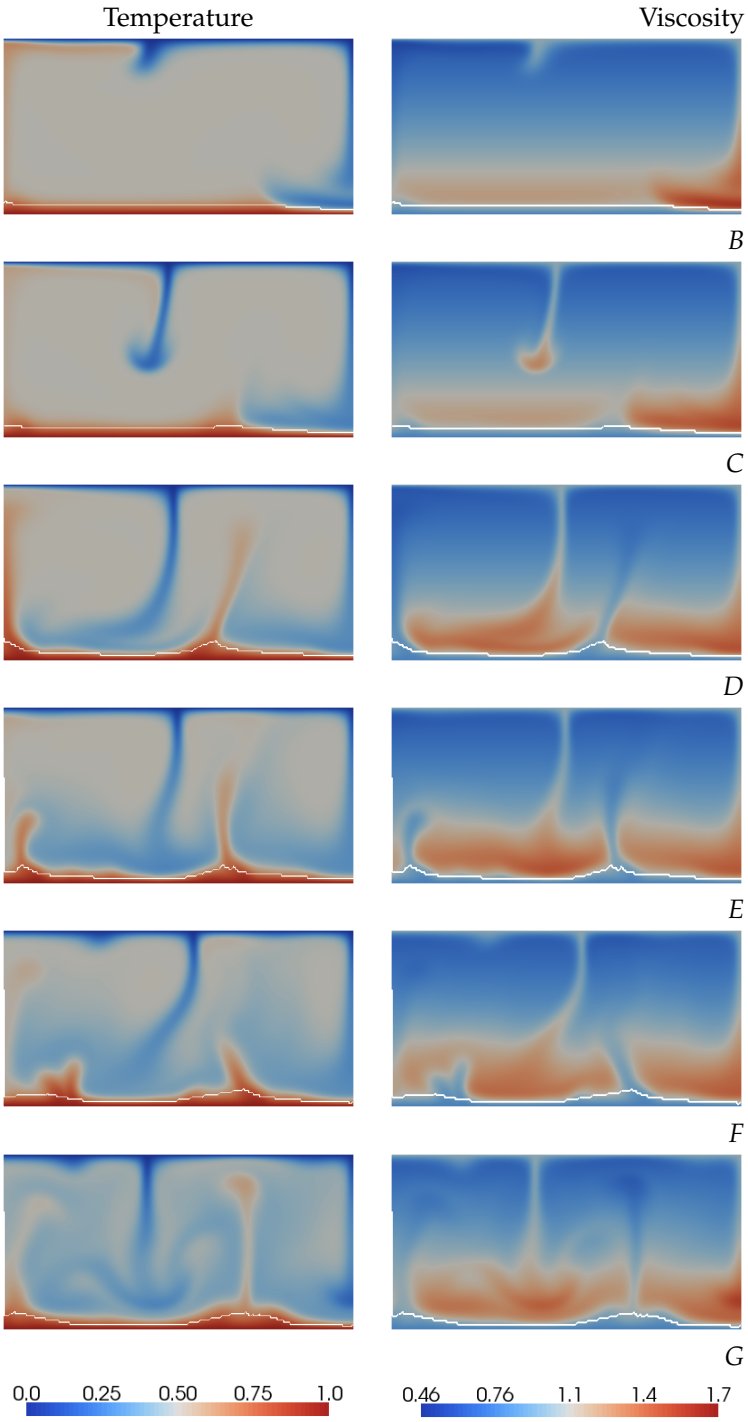


Figure 31.8: Dimensionless temperature and viscosity for turning points A to G, with composition barrier shown in white. To convert to physically relevant temperature contrast and viscosities, these values should be multiplied by dimensional η_0 and ΔT from Table 31.1. The initial condition drives the system's high velocity from point A, but as the cold surface material (slab) reaches the deeper mantle, there is a retardation of the flow, towards B. The rising hot material (plume) during the stage A to B drives lateral flow of the surface causing cold material to build up until B. From B to C this cold material begins to rapidly sink, increasing u_{rms} until it is slowed by the increasing viscosity at C. However, the slab's downward motion has created a thermal instability at the base of the model, which rises between C and D increasing u_{rms} again. The pace of the material slows as the plume necks between D and E and the compositional density of the remaining material prevents it from rising further. A small plume rises from the left side of the base, increasing the u_{rms} briefly. Between E and G short-lived plumes and slabs are generated from the bottom and top boundary layers, causing small instabilities in the root-mean square velocity.

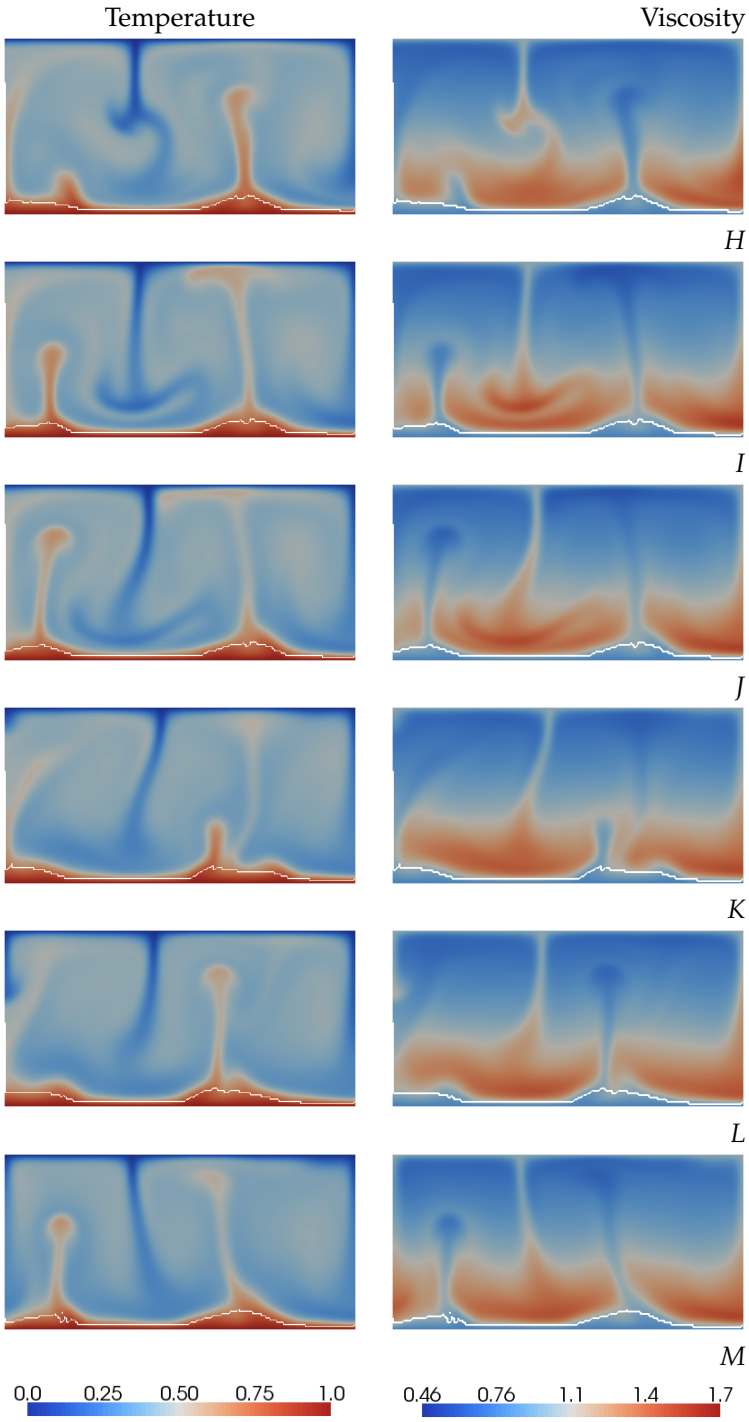


Figure 31.9: Continued from Figure 31.8, temperature and viscosity for points H to M. During the stage G-H, a second slab forms and the two downwellings merge increasing the root-mean square velocity rapidly. From H, the downwelling is impeded by the higher viscosity at depth, reducing u_{rms} until I. Then a plume arising from the bottom left rises more rapidly through the lower viscosities of the upper mantle until J. Between J and K, no new up- or downwellings occur, retarding the root-mean square velocity. From K to L a new plume increases the kinetic energy in the model, then pushes material laterally from underneath the top boundary layer until a slab begins to sink at the edge from M, increasing the velocity again.

32 *Automatic calibration of depositional models*

By Hans Joachim Schroll

A novel concept for calibrating depositional models is presented. In this approach, transport coefficients are determined from well output measurements. Finite element implementation of the multi-lithology models and their duals is automated by the DOLFIN Python interface.

32.1 Issues in sedimentary deposition

Evidence indicates that millions of years of pressure cooking transforms the remains of living organisms into crude oil and natural gas. This process takes place in sealed reservoirs, so-called structural and/or stratigraphic traps in the reservoir rock. To identify potential new reservoirs, it is of great interest to understand the geological evolution of depositional basins in the Earth's crust. Different types of forward computer models are used by sedimentologists and geomorphologists to simulate the process of sedimentary deposition over geological time periods. The models can be used to predict the presence of reservoir rocks and stratigraphic traps at a variety of scales. State-of-the-art advanced numerical software, like for example DIONISOS by Beicip-Franlab, provides accurate approximations to the mathematical model, which commonly is expressed in terms of a nonlinear diffusion dominated PDE system. The potential of today's simulation software in industrial applications is limited however, due to major uncertainties in crucial material parameters that combine a number of physical phenomena and therefore are difficult to quantify. Examples of such parameters are diffusive transport coefficients.

The idea in this contribution is to calibrate uncertain transport coefficients to direct observable data, like well measurements from a specific basin. In this approach the forward evolution process, mapping data to observations, is reversed to determine the data; that is, transport coefficients. Mathematical tools and numerical algorithms are applied to automatically calibrate geological models to actual observations — a critical but so far missing link in forward depositional modeling.

Automatic calibration, in combination with stochastic modeling, will boost the applicability and impact of modern numerical simulations in industrial applications.

32.2 A multidimensional sedimentation model

Submarine sedimentation is an evolution process. By flow into the basin, sediments build up and evolve in time. In dual lithology models, two types of sediments are considered: sand and mud. The evolution follows geophysical laws, expressed as diffusive PDE models. The following system is a multidimensional version of the model by Rivenæs (1992, 1993):

$$\begin{pmatrix} A & s \\ -A & 1-s \end{pmatrix} \begin{pmatrix} s \\ h \end{pmatrix}_t = \nabla \cdot \begin{pmatrix} \alpha s \nabla h \\ \beta(1-s)\nabla h \end{pmatrix} \text{ in } [0,T] \times B. \tag{32.1}$$

Here h denotes the thickness of a layer of deposit and s models the volume fraction for the sand lithology. Consequently, $1-s$ is the fraction for mud. The system is driven by fluxes anti proportional to the flow rates $s\nabla h$ and $(1-s)\nabla h$ resulting in a diffusive, but incomplete parabolic, PDE system. The domain of the basin is denoted by B. Parameters in the model are: The transport layer thickness A and the diffusive transport coefficients α, β.

For a forward in time simulation, the system requires initial and boundary data. At initial time, the volume fraction s and the layer thickness h need to be specified. According to geologists, such data can be reconstructed by some kind of "back stripping". Along the boundary of the basin, the flow rates $s\nabla h$ and $(1-s)\nabla h$ are given.

32.3 An inverse approach

The parameter-to-observation mapping $R : (\alpha, \beta) \mapsto (s, h)$ is commonly referred to as the forward problem. In a basin, direct observations are only available at wells. Moreover, from the age of the sediments, their history can be reconstructed. Finally, well-data is available in certain well areas $W \subset B$ and backward in time.

The objective of the present investigation is to determine transport coefficients from observed well-data and in that way, to calibrate the model to the data. This essentially means to invert the parameter-to-observation mapping. Denoting observed well-data by $(\widetilde{s}, \widetilde{h})$, the goal is to minimize the output functional

$$J(\alpha, \beta) = \frac{1}{|W|} \int_0^T \int_W (\widetilde{s} - s)^2 + (\widetilde{h} - h)^2 \, \mathrm{d}x \, \mathrm{d}t \tag{32.2}$$

with respect to the transport coefficients α and β.

In contrast to the "direct inversion" as described by Imhof and Sharma (2007), which is considered impractical, we do not propose to invert the time evolution of the diffusive depositional process. We actually use the forward-in-time evolution of sediment layers in a number of wells to calibrate transport coefficients. Via the calibrated model we can simulate the basin and reconstruct its historic evolution. By computational mathematical modeling, the local data observed in wells determines the evolution throughout the entire basin.

32.4 The Landweber algorithm

In a slightly more abstract setting, the task is to minimize an objective functional J which implicitly depends on the parameters p via u subject to the constraint that u satisfies some PDE model; a PDE constrained minimization problem: find p such that $J(p) = J(u(p)) = \min$ and $\mathrm{PDE}(u, p) = 0$.

Landweber's steepest descent algorithm (Landweber, 1951) iterates the following sequence until convergence:
While $\|\nabla_p J(p^k)\| > TOL$:

1. Solve $\mathrm{PDE}(u^k, p^k) = 0$ for u^k.
2. Evaluate $d^k = -\nabla_p J(p^k) / \|\nabla_p J(p^k)\|$.
3. Update $p^{k+1} = p^k + \Delta p^k d^k$.

Note that the search direction d^k, the negative gradient, is the direction of steepest descent. To avoid scale dependence, the search direction is normed.

The increment Δp^k is determined by a one dimensional line search algorithm. As $J(p)$ is only implicitly defined via the solution of the PDE, a locally quadratic approximation to J is minimized

along the line $p^k + \gamma d^k$. We use the Ansatz

$$J(p^k + \gamma d^k) = a\gamma^2 + b\gamma + c, \quad \gamma \in \mathbb{R}. \tag{32.3}$$

The extreme value of this parabola is located at

$$\gamma^e = -\frac{b}{2a}. \tag{32.4}$$

To determine γ^e, the parabola is fitted to the local data. For example, b is given by the gradient

$$J_\gamma(p^k) = b = \nabla_p J(p^k) \cdot d^k = -\|\nabla_p J(p^k)\|. \tag{32.5}$$

To find a, another γ-derivative of J along the line $p^k + \gamma d^k$ is needed. To avoid an extra evaluation of the gradient, we project $p^{k-1} - p^k = -\Delta p^{k-1} d^{k-1}$ onto d^k. The scalar projection is

$$\bar{\gamma} = -\Delta p^{k-1} d^{k-1} \cdot d^k. \tag{32.6}$$

Next, we approximate J_γ in the projected point $p^k - \bar{\gamma} d^k$ by the directional derivative evaluated in the previous iterate

$$J_\gamma(p^k - \bar{\gamma} d^k) \approx \nabla_p J(p^{k-1}) \cdot d^k. \tag{32.7}$$

Note that this approximation is exact if two successive directions d^{k-1} and d^k are in line. From the Ansatz (32.3) and the approximation (32.7), it follows that

$$-2a\bar{\gamma} + b = \nabla_p J(p^{k-1}) \cdot d^k. \tag{32.8}$$

Using (32.5) and (32.6), we find

$$-2a = \frac{(\nabla_p J(p^{k-1}) - \nabla_p J(p^k)) \cdot d^k}{-\Delta p^{k-1} d^{k-1} \cdot d^k}. \tag{32.9}$$

Thus, the increment (32.4) evaluates as

$$\Delta p^k = \gamma^e = \frac{\nabla_p J(p^k) \cdot \nabla_p J(p^{k-1})}{\nabla_p J(p^k) \cdot \nabla_p J(p^{k-1}) - \|\nabla_p J(p^k)\|^2} \frac{\|\nabla_p J(p^k)\|}{\|\nabla_p J(p^{k-1})\|} \cdot \Delta p^{k-1}. \tag{32.10}$$

32.5 Evaluation of gradients by duality arguments

Every single step of Landweber's algorithm requires the simulation of a time dependent, nonlinear PDE system and the evaluation of the gradient of the objective functional. The most common approach to numerical derivatives, via finite differences, is impractical for complex problems: Finite difference approximation would require to perform $n + 1$ forward simulations in n parameter dimensions. Using duality arguments however, n nonlinear PDE systems can be replaced by one linear, dual problem. After all, J is evaluated by one forward simulation of the nonlinear PDE model and the complete gradient ∇J is obtained by one (backward) simulation of the linear, dual problem. Apparently, one of the first references to this kind of duality arguments is Chavent and Lemmonier (1974).

The concept is conveniently explained for a scalar diffusion equation

$$u_t = \nabla \cdot (\alpha \nabla u). \tag{32.11}$$

As transport coefficients may vary throughout the basin, we allow for a piecewise constant coefficient

$$\alpha = \begin{cases} \alpha_1 & x \in B_1 \\ \alpha_2 & x \in B_2 \end{cases}, \quad B = B_1 \cup B_2. \tag{32.12}$$

Assuming no flow along the boundary and selecting a suitable function space, the equation in variational form reads

$$a(u,\phi) := \int_0^T \int_B u_t\phi + \alpha\nabla u \cdot \nabla\phi \,\mathrm{d}x\,\mathrm{d}t = 0, \tag{32.13}$$

where ϕ is a test function. Taking a derivative $\partial/\partial\alpha_i$, $i = 1,2$ under the integral sign, we find

$$a(u_{\alpha_i},\phi) = \int_0^T \int_B u_{\alpha_i,t}\phi + \alpha\nabla u_{\alpha_i} \cdot \nabla\phi \,\mathrm{d}x\,\mathrm{d}t = -\int_0^T \int_{B_i} \nabla u \cdot \nabla\phi \,\mathrm{d}x\,\mathrm{d}t. \tag{32.14}$$

The corresponding derivative of the output functional $J = \frac{1}{|W|}\int_0^T \int_W (u-d)^2 \,\mathrm{d}x\,\mathrm{d}t$ reads

$$J_{\alpha_i} = \frac{2}{|W|}\int_0^T \int_W (u-d)u_{\alpha_i} \,\mathrm{d}x\,\mathrm{d}t, \quad i = 1,2. \tag{32.15}$$

The trick is to define a dual problem

$$a(\phi,\omega) = \frac{2}{|W|}\int_0^T \int_W (u-d)\phi \,\mathrm{d}x\,\mathrm{d}t \tag{32.16}$$

such that $a(u_{\alpha_i},\omega) = J_{\alpha_i}$ and by using the dual solution ω in (32.14)

$$a(u_{\alpha_i},\omega) = J_{\alpha_i} = -\int_0^T \int_{B_i} \nabla u \cdot \nabla\omega \,\mathrm{d}x\,\mathrm{d}t, \quad i = 1,2. \tag{32.17}$$

In effect, the desired gradient ∇J is expressed in terms of primal and dual solutions. In this case the dual problem reads

$$\int_0^T \int_B \phi_t\omega + \alpha\nabla\phi \cdot \nabla\omega \,\mathrm{d}x\,\mathrm{d}t = \frac{2}{|W|}\int_0^T \int_W (u-d)\phi \,\mathrm{d}x\,\mathrm{d}t, \tag{32.18}$$

which in strong form appears as a backward in time heat equation with zero terminal condition

$$-\omega_t = \nabla \cdot (\alpha\nabla\omega) + \frac{2}{|W|}(u-d)|_W. \tag{32.19}$$

Note that this dual equation is linear and entirely driven by the data mismatch in the well. With perfectly matching data $d = u|_W$, the dual solution is zero.

Along the same lines of argumentation one derives the multilinear operator to the depositional model (32.1)

$$\begin{aligned} a(u,v)(\phi,\psi) = &\int_0^T \int_B \left(Au_t + uh_t + sv_t\right)\phi + \alpha u\nabla h \cdot \nabla\phi + \alpha s\nabla v \cdot \nabla\phi \,\mathrm{d}x\,\mathrm{d}t \\ &+ \int_0^T \int_B \left(-Au_t - uh_t(1-s)v_t\right)\psi - \beta u\nabla h \cdot \nabla\psi + \beta(1-s)\nabla v \cdot \nabla\psi \,\mathrm{d}x\,\mathrm{d}t. \end{aligned} \tag{32.20}$$

The dual system related to the well output functional (32.2) reads

$$a(\phi,\psi)(\omega,\nu) = \frac{2}{|W|}\int_0^T \int_W (s-\widetilde{s})\phi + (h-\widetilde{h})\psi \,\mathrm{d}x\,\mathrm{d}t. \tag{32.21}$$

By construction it follows $a(s_p,h_p)(\omega,\nu) = J_p(\alpha,\beta)$. Given both primal and dual solutions, the gradient of the well output functional evaluates as

$$J_{\alpha_i}(\alpha,\beta) = -\int_0^T \int_{B_i} s\nabla h\cdot\nabla\omega \,\mathrm{d}x\,\mathrm{d}t, \tag{32.22}$$

$$J_{\beta_i}(\alpha,\beta) = -\int_0^T \int_{B_i} (1-s)\nabla h\cdot\nabla\nu \,\mathrm{d}x\,\mathrm{d}t. \tag{32.23}$$

A detailed derivation including non zero flow conditions is given in Schroll (2008). For completeness, not for computation(!), we state the dual system in strong form

$$-A(\omega-\nu)_t + h_t(\omega-\nu) + \alpha\nabla h\cdot\nabla\omega = \beta\nabla h\cdot\nabla\nu + \frac{2}{|W|}(s-\widetilde{s})\Big|_W, \tag{32.24}$$

$$-(s\omega+(1-s)\nu)_t = \nabla\cdot(\alpha s\nabla\omega + \beta(1-s)\nabla\nu) + \frac{2}{|W|}(h-\widetilde{h})\Big|_W. \tag{32.25}$$

Obviously the system is linear and driven by the data mismatch at the well. It always comes with zero terminal condition and no flow conditions along the boundary of the basin. Thus, perfectly matching data results in a trivial dual solution.

32.6 *Aspects of the implementation*

The FEniCS project DOLFIN (Logg and Wells, 2010) automates the solution of PDEs in variational formulation and is therefore especially attractive for implementing dual problems, which are derived in variational form. In this section the coding of the dual diffusion equation (32.18) is illustrated. Choosing a test function ϕ supported in $[t_n,t_{n+1}]\times B$, the weak form reads

$$-\int_{t_n}^{t_{n+1}}\int_B \omega_t\phi + \alpha\nabla\omega\cdot\nabla\phi \,\mathrm{d}x\,\mathrm{d}t = \frac{2}{|W|}\int_{t_n}^{t_{n+1}}\int_W (u-d)\phi \,\mathrm{d}x\,\mathrm{d}t. \tag{32.26}$$

Using dG(0) time integration with a trapezoidal quadrature rule applied to the right-hand side gives

$$-\int_B(\omega^{n+1}-\omega^n)\phi\,\mathrm{d}x + \frac{\Delta t}{2}\int_B \alpha\nabla(\omega^{n+1}+\omega^n)\cdot\nabla\phi\,\mathrm{d}x = \frac{\Delta t}{|W|}\int_W (u^{n+1}-d^{n+1}+u^n-d^n)\phi\,\mathrm{d}x, \quad n=N,N-1,\ldots,0. \tag{32.27}$$

To evaluate the right-hand side, the area of the well is defined as a subdomain:

Python code

```
class WellDomain(SubDomain):
    def inside(self, x, on_boundary):
        return 0.2 <= x[0] and x[0] <= 0.3 and 0.2 <= x[1] and x[1] <= 0.3
```

Next, it gets marked:

Python code

```
well = WellDomain()
subdomains = MeshFunction("uint", mesh, mesh.topology().dim())
well.mark(subdomains, 1)
```

An integral over the well area is defined:

Python code

```
dxWell = dx(1)
```

The area of the well is computed:

Python code

```
wellarea = assemble(Constant(1.0)*dxWell, cell_domains=subdomains, mesh=mesh)
```

The driving source in (32.27) is written as:

Python code

```
f = dt*(u1-d1+u0-d0)*phi*dxWell/wellarea
```

The first line in (32.27) is stated in variational formulation:

Python code

```
F = (u_trial-u)*phi*dx + 0.5*dt*(d*dot( grad(u_trial + u), grad(phi)))*dx
```

Let DOLFIN sort out left- and right-hand sides:

Python code

```
a = lhs(F)
L = rhs(F)
```

Construct the variational problem:

Python code

```
problem = LinearVariationalProblem(a, L + f, u)
```

And solve it:

Python code

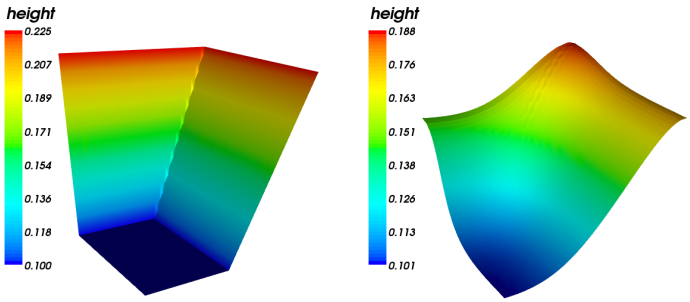
```
solver = LinearVariationalSolver(problem)
solver.solve()
```

32.7 Numerical experiments

With these preparations, we are now ready to inspect the well output functional (32.2) for possible calibration of the dual lithology model (32.1) to "observed", actually generated synthetic, data. We consider the PDE system (32.1) with discontinuous coefficients

$$\alpha = \begin{cases} \alpha_1 & x \geq 1/2 \\ \alpha_2 & x < 1/2 \end{cases}, \quad \beta = \begin{cases} \beta_1 & x \geq 1/2 \\ \beta_2 & x < 1/2 \end{cases} \tag{32.28}$$

Figure 32.1: Evolution of h, initial left, $t = 0.04$ right.



in the unit square $B = [0,1]^2$. Four wells $W = W_1 \cup W_2 \cup W_3 \cup W_4$ are placed one in each quarter

$$W_4 = [0.2, 0.3] \times [0.7, 0.8], \quad W_3 = [0.7, 0.8] \times [0.7, 0.8], \tag{32.29}$$

$$W_1 = [0.2, 0.3] \times [0.2, 0.3], \quad W_2 = [0.7, 0.8] \times [0.2, 0.3]. \tag{32.30}$$

Initially, s is constant $s(0, \cdot) = 0.5$ and h is piecewise linear

$$h(0, x, y) = 0.5 \max(\max(0.2, (x - 0.1)/2), y - 0.55). \tag{32.31}$$

The diffusive character of the process is evident from the evolution of h as shown in Figure 32.1. No flow boundary conditions are implemented in all simulations throughout this section.

To inspect the output functional, we generate synthetic data by computing a reference solution. In the first experiment, the reference parameters are $(\alpha_1, \alpha_2) = (\beta_1, \beta_2) = (0.8, 0.8)$. We fix β to the reference values and scan the well output over the α-range $[0.0, 1.5]^2$. The upper left plot in Figure 32.2 depicts contours of the apparently convex functional, with the reference parameters as the global minimum. Independent Landweber iterations, started in each corner of the domain, identify the optimal parameters in typically five steps. The iteration is stopped if $\|\nabla J(p^k)\| \leq 10^{-7}$, an indication that the minimum is reached. In all figures below, a green dot marks the optimal, calibrated parameters. The lower left plot shows the corresponding scan over β where $\alpha = (0.8, 0.8)$ is fixed. Obviously the search directions follow the steepest descent, confirming that the gradients are correctly evaluated via the dual solution. In the right column of Figure 32.2 we see results for the same experiments, but with 5% random noise added to the synthetic well data. In this case the optimal parameters are of course not the reference parameters, but still close. The global picture appears stable with respect to noise, suggesting that the concept allows to calibrate diffusive, depositional models to data observed in wells.

Ultimately, the goal is to calibrate all four parameters $\alpha = (\alpha_1, \alpha_2)$ and $\beta = (\beta_1, \beta_2)$ to available data. Figure 32.3 depicts Landweber iterations in four dimensional parameter space, started at $\alpha = (1.4, 0.2)$ and $\beta = (0.2, 1.4)$. Actually, projections onto the α and β coordinate plane are shown. Obviously, both iterations converge and, without noise added, the reference parameters, $\alpha = \beta = (0.8, 0.8)$, are detected as optimal parameters. Adding 5% random noise to the recorded data, we simulate data observed in wells. In this situation, see the right column, the algorithm identifies optimal parameters, which are clearly off the reference. Figure 32.5 depicts fifty realizations of this experiments. The distribution of the optimal parameters is shown together with their average in red. The left graph in Figure 32.5 corresponds to the reference parameters $(\alpha_1, \alpha_2) = (\beta_1, \beta_2) = (0.8, 0.8)$ as in Figure 32.3. On average, the calibrated, optimal parameters are $\bar{\alpha} = (0.860606, 0.800396)$ and $\bar{\beta} = (0.729657, 0.827728)$ with standard deviations $(0.184708, 0.127719)$ and $(0.176439, 0.12411)$.

In the next experiments, non-uniform reference parameters are set for $\alpha = (0.6, 1.0)$ and $\beta = (1.0, 0.6)$. Figure 32.4 shows iterations with the noise-free reference solution used as data on the

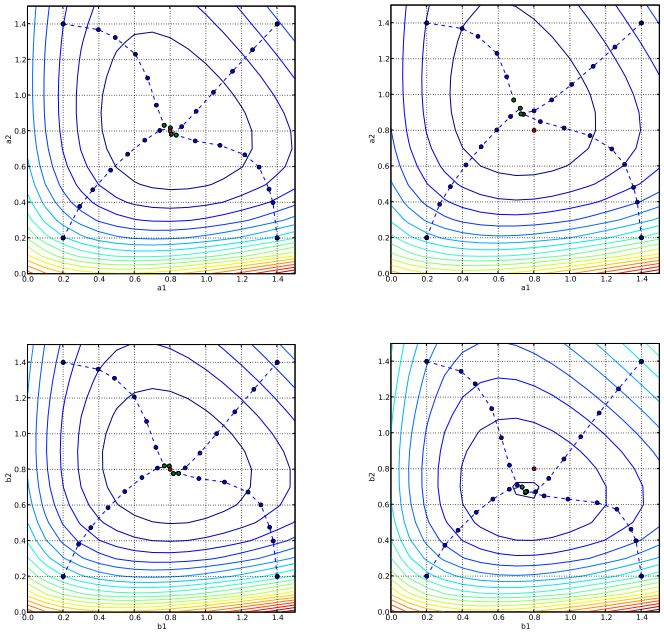


Figure 32.2: Contours of J and Landweber iterations. Optimal parameters (green), reference parameters (red). Clean data left column, noisy data right. α-iterations upper row, β-iterations lower row.

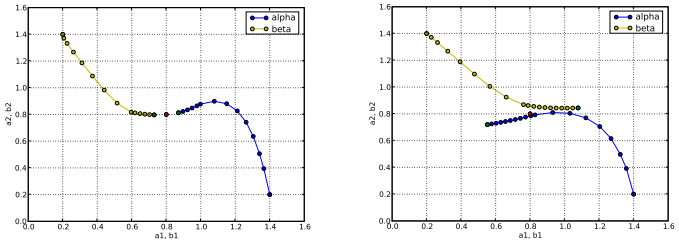


Figure 32.3: Landweber iterations. Optimal parameters (green), reference parameters (red). Clean (left) and noisy data (right).

Figure 32.4: Landweber iterations. Optimal parameters (green), reference parameters (red). Clean (left) and noisy data (right).

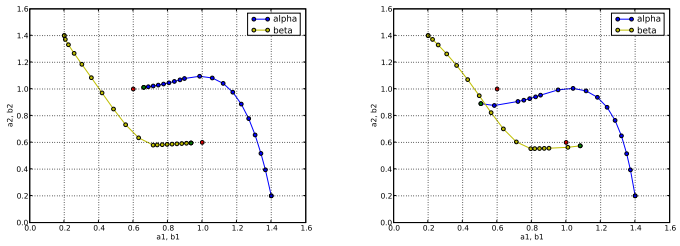
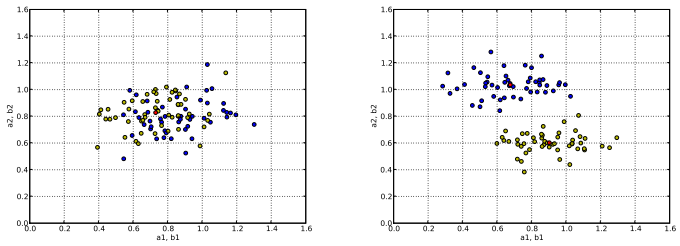


Figure 32.5: Sets of optimal parameters calibrated to noisy data, α blue, β yellow, average red.



left-hand side. Within the precision of the stopping criterion, the reference parameters are detected. Adding 5% noise to the well data leads to different optimal parameters, just as expected. On average however, the optimal parameters obtained in repeated calibrations $\bar{\alpha} = (0.676051, 1.03604)$ and $\bar{\beta} = (0.902532, 0.602344)$ match the reference parameters quite well, see Figure 32.5, right-hand side. The standard deviations in this case are $(0.18374, 0.0886383)$ and $(0.166801, 0.0750035)$, respectively.

In the last experiment, α is discontinuous along $x = 1/2$, while β is discontinuous along $y = 1/2$:

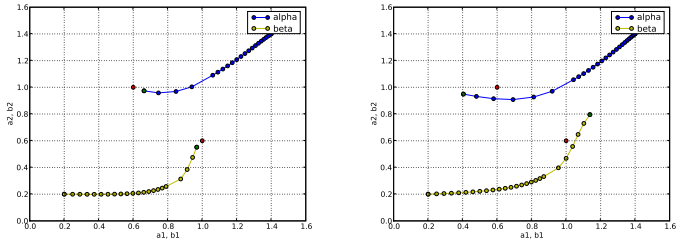
$$\alpha = \begin{cases} 1.0 & x \geq 1/2 \\ 0.6 & x < 1/2 \end{cases}, \quad \beta = \begin{cases} 0.6 & y \geq 1/2 \\ 1.0 & y < 1/2 \end{cases} \tag{32.32}$$

In this way the evolution is governed by different diffusion parameters in each quarter of the basin. Having placed one well in each quarter, one can effectively calibrate the model to synthetic data with and without random noise, as shown in Figure 32.6.

32.8 Results and conclusions

The calibration of piecewise constant diffusion coefficients using local data in a small number of wells is a well behaved inverse problem. The convexity of the output functional, which is the basis for a successful minimization, remains stable with random noise added to the well data. The Landweber algorithm, with duality based gradients, automatically detects optimal parameters. As the dual

Figure 32.6: Landweber iterations. Optimal parameters (green), reference parameters (red). Clean (left) and noisy data (right).



problems are derived in variational form, the FEniCS project DOLFIN is the ideal tool for efficient implementation.

Acknowledgements

The author wants to thank Are Magnus Bruaset for initiating this work. Many thanks to Bjørn Fredrik Nielsen for thoughtful suggestions regarding inverse problems as well as Anders Logg and Ola Skavhaug for their support regarding the DOLFIN software.

The presented work was funded by a research grant from Statoil. The work has been conducted at Simula Research Laboratory as part of CBC, a Center of Excellence awarded by the Norwegian Research Council.

33 A coupled stochastic and deterministic model of Ca^{2+} dynamics in the dyadic cleft

By Johan Hake

From the time we are children we are told that we should drink milk because it is an important source of calcium (Ca^{2+}), and that Ca^{2+} is vital for a strong bone structure. What we do not hear as frequently is that Ca^{2+} is one of the most important cellular messengers in the human body (Alberts et al., 2002). In particular, Ca^{2+} controls cell death, neural signaling, secretion of different chemical substances and the contraction of cells in the heart. The latter is the focus of this chapter.

In this chapter, we will present a computational model that can be used to model Ca^{2+} dynamics in a small subcellular domain called the dyadic cleft. The model includes Ca^{2+} diffusion, which is described by an advection–diffusion partial differential equation, and discrete channel dynamics, which are described by stochastic Markov models. Numerical methods implemented in DOLFIN solving the partial differential equation will also be presented. In the last section, we describe a time stepping scheme that is used to solve both the stochastic and deterministic models. We will also present a solver framework, `diffsim`, that implements this time stepping scheme together with the numerical methods to solve the model described above.

33.1 Biological background

In a healthy heart every heart beat originates in the sinoatrial node where pacemaker cells trigger an electric signal. This signal is a difference in electric potential between the interior and exterior of the heart cells; these two domains are separated by the cell membrane. The difference in the electric potential between these two domains is called the membrane potential. The membrane potential propagates through the whole heart via electrical currents which move through the cell membrane using specific ion channels. The actively propagating membrane potential is called an action potential. When an action potential arrives at a specific heart cell, it triggers the opening of L-type Ca^{2+} channels (LCCs), which bring Ca^{2+} into the cell. Some of the Ca^{2+} diffuses over a small distance, called the dyadic cleft, and causes further Ca^{2+} release from an intracellular Ca^{2+} storage, the sarcoplasmic reticulum (SR), through a channel known as the ryanodine receptor (RyR). The Ca^{2+} ions then diffuse to the main intracellular domain of the cell, the cytosol, in which the contractile proteins are situated. These proteins are responsible for generating contraction in the heart cell and Ca^{2+} serves as the trigger. The strength of the contraction is controlled by the strength of the Ca^{2+} concentration ($[Ca^{2+}]$) in cytosol. The contraction is succeeded by a period of relaxation caused by the extraction of Ca^{2+} from the intracellular space by various proteins.

This chain of events is known as the Excitation Contraction (EC) coupling (Bers, 2001). Several severe heart diseases can be related to impaired EC coupling. By broadening knowledge of the EC

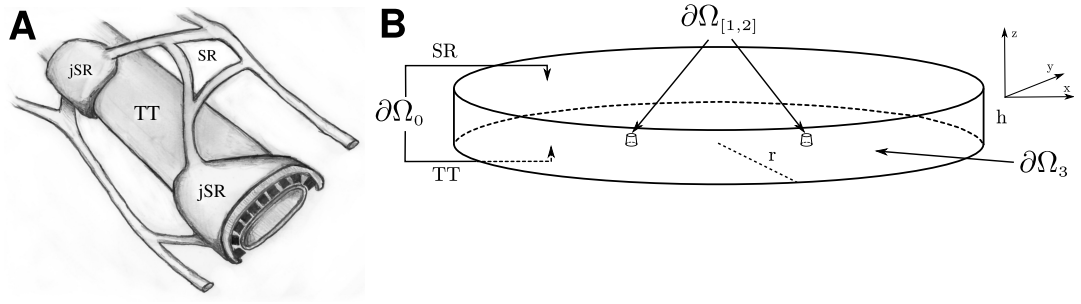


Figure 33.1: (A): A diagram showing the relationship between the TT, the SR, and the jSR in the interior of a heart cell. The volume between the flat jSR and the TT is the dyadic cleft. The black structures in the cleft are Ryanodine receptors, which are large channel proteins. (B): The geometry used for the computational model of the dyadic cleft. The top of the disk is the cell membrane of the SR or jSR. The bottom is the cell membrane of the TT, and the circumference of the disk is the interface to the cytosol. The elevations in the TT membrane model two Ca^{2+} ion channels.

coupling, it may be possible to develop better treatments for such diseases. Although grasping the big picture of EC coupling is straightforward, it actually involves the nonlinear action of hundreds of different protein species. Computational methods have emerged as a natural complement to experimental studies to better understand this process. In this chapter, we focus on the initial phase of EC coupling wherein Ca^{2+} flows into the cell and triggers further Ca^{2+} release.

33.2 Mathematical models

In this section we describe the computational model for the early phase of EC coupling. We first present the morphology of the dyadic cleft and how we model this in our study. We then describe the mathematical formulation for the diffusion of Ca^{2+} inside the cleft as well for the Ca^{2+} fluxes across the boundaries. Finally, we discuss stochastic models that govern discrete channel dynamics of the LCCs and RyRs.

33.2.1 Morphology

The dyadic cleft is the volume in the interior of the heart cell between a structure called the t-tubule (TT) and the SR. The TT is a network of pipe-like invaginations of the cell membrane that perforate the heart cell (Soeller and Cannell, 1999). In Figure 33.1 (A), a sketch of a small part of a single TT together with a piece of SR is presented. Here we see that the junctional SR (jSR) is wrapped around the TT. The small volume between these two structures is the dyadic cleft. The space is not well defined as it is crowded with channel proteins and varies in size. In computational studies it is commonly approximated as a disk or a rectangular slab (Peskoff et al., 1992; Soeller and Cannell, 1997; Koh et al., 2006; Tanskanen et al., 2007). In this study a disk with height h = 12 nm and radius r = 50 nm has been used for the domain Ω; see Figure 33.1 (B).

33.2.2 Ca^{2+} diffusion

Electro-diffusion. We will use Fick's second law to model diffusion of Ca^{2+} in the dyadic cleft. The diffusion constant of Ca^{2+} is set to $\sigma = 10^5$ nm^2 ms^{-1} (Langer and Peskoff, 1996). Close to the cell membrane, ions are affected by an electric potential caused by negative charges on the membrane (McLaughlin et al., 1971; Langner et al., 1990). The potential attenuates rapidly as it is countered by positive ions that are attracted by the negative electrical field. This process is known as screening. We

will describe the electric potential using the Gouy–Chapman method (Grahame, 1947). This theory introduces an advection term to the standard diffusion equation, which makes the resulting equation more difficult to solve. To simplify the presentation, we will use a non-dimensional electric potential ψ, which is the electric potential scaled by a factor of e/kT. Here e is the electron charge, k is Boltzmann's constant and T is the temperature. We will also use a non-dimensional electric field which is given by:

$$E = -\nabla\psi. \tag{33.1}$$

The Ca^{2+} flux in a solution in the presence of an electric field is governed by the Nernst–Planck equation,

$$J = -\sigma\left(\nabla c - 2\,cE\right), \tag{33.2}$$

where $c = c(x,t)$ is the [Ca^{2+}] ($x \in \Omega$ and $t \in$[0,T]), σ the diffusion constant, $E = E(x)$ the non-dimensional electric field and 2 is the valence of Ca^{2+}. Assuming conservation of mass, we arrive at the advection–diffusion equation,

$$\dot{c} = \sigma\left(\Delta c + \nabla\cdot\left(2\,cE\right)\right), \tag{33.3}$$

where $\dot{c}$ is the time derivative of c.

The strength of ψ is defined by the amount of charge at the cell membrane and by the combined screening effect of all the ions in the dyadic cleft. In addition to Ca^{2+}, the intracellular solution also contains K^+, Na^+, Cl^-, and Mg^{2+}. Following the previous approach by Langner et al. (1990) and Soeller and Cannell (1997), these other ions are treated as being in steady state. The cell membrane is assumed to be planar and effectively infinite. This last assumption allows us to use an approximation of the electric potential in the solution,

$$\psi(z) = \psi_0 \mathrm{e}^{-\kappa z}. \tag{33.4}$$

Here ψ_0 is the non-dimensional potential at the membrane, κ the inverse Debye length and z the distance from the cell membrane. We will use $\psi_0 = -2.2$ and $\kappa = 1$ nm (Soeller and Cannell, 1997).

Boundary fluxes. The boundary $\partial\Omega$ is divided into four disjoint boundaries $\partial\Omega_k$ for $k = 0,\ldots,3$; see Figure 33.1 (B). To each boundary we assign a flux, $J_{|\partial\Omega_k} = J_k$. The SR and TT membranes are impermeable to ions, effectively making $\partial\Omega_0$ in Figure 33.1 (B) a no-flux boundary, giving us

$$J_0 = 0. \tag{33.5}$$

We include 2 LCCs in our model. The Ca^{2+} flows into the cleft at the $\partial\Omega_{[1,2]}$ boundaries; see Figure 33.1 (B). Ca^{2+} entering via these channels then diffuses to the RyRs, triggering Ca^{2+} release from the SR. This additional Ca^{2+} flux will not be included in the simulations; however, the stochastic dynamics of the opening of the RyR channel will be considered. The Ca^{2+} that enters the dyadic cleft diffuses into the main compartment of cytosol, introducing a third flux. This flux is included in the model at the $\partial\Omega_3$ boundary.

The LCC is a stochastic channel that is either open or closed. When the channel is open Ca^{2+} flows into the cleft. The current amplitude of an open LCC channel is modeled to be -0.1 pA (Guia et al., 2001). The LCC flux is then,

$$J_{[2,3]} = \begin{cases} 0, & \text{closed channel,} \\ -\frac{i}{2FA}, & \text{open channel,} \end{cases} \tag{33.6}$$

where i is the amplitude, 2 the valence of Ca^{2+}, F Faraday's constant and A the area of the channel.

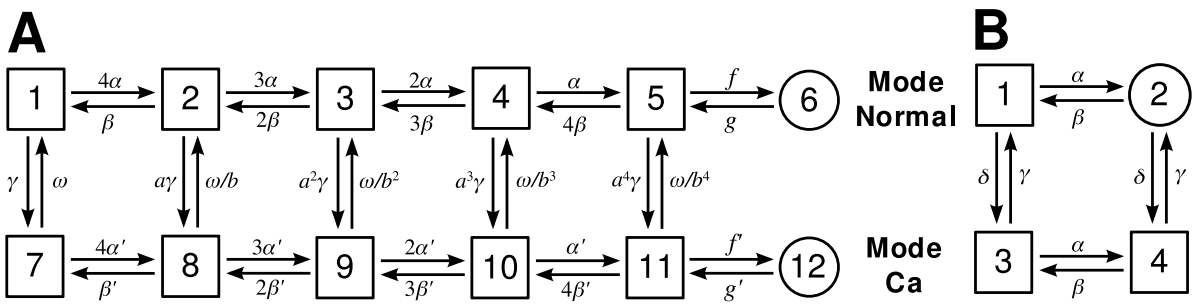


Figure 33.2: (A): State diagram of the discrete LCC Markov model from Jafri et al. (1998). Each channel can be in one of the 12 states. The transitions between the states are controlled by propensities. The α, and β are voltage-dependent, γ is $[Ca^{2+}]$-dependent and f, a, b, and ω are constant (see Jafri et al. (1998) for further details). The channel operates in two modes: *Mode normal*, represented by the states in the upper row, and *Mode Ca*, represented by the states in the lower row. In state 6 and 12, the channel is open, but state 12 is rarely entered as $f' \ll f$, effectively making *Mode Ca* an inactivated mode. (B): State diagram of an RyR from Stern et al. (1999). The α and γ propensities are Ca^{2+}-dependent, representing the activation and inactivation dependency of the cytosolic $[Ca^{2+}]$. The β and δ propensities are constant.

Note that an inward current is, by convention, negative.

The flux to the cytosol is modeled as a concentration-dependent flux,

$$J_3 = -\sigma \frac{c - c_0}{\Delta s}, \tag{33.7}$$

where c is the concentration in the cleft at the boundary, c_0 the concentration in the cytosol, and Δs is an approximation of the distance to the center of the cytosol. In our model, we have used Δs = 50 nm.

33.2.3 Stochastic models of single channels

Discrete and stochastic Markov chain models are used to describe single channel dynamics for the LCC and RyR. Such models are described by a certain number of discrete states. Each channel can be in any one of these states, and a transition between two states is a stochastic event. The frequency of these events is determined by the so called propensity functions associated with each transition. These functions, which may vary with time, characterize the probability per unit time that the corresponding transition event occurs. Each Markov model defines its own propensity functions.

L-type Ca^{2+} channel. The LCC opens when an action potential arrives at the cell, and the channel inactivates when Ca^{2+} ions bind to binding sites on the intracellular side of the channel. An LCC is composed of a complex of four transmembrane subunits. Each of these can be permissive or non-permissive. For the whole channel to be open, all four subunits must be permissive, and the channel must then undergo a last conformational change to the open state (Hille, 2001). In this chapter, we employ a Markov model of the LCC that incorporates voltage-dependent activation together with a Ca^{2+}-dependent inactivation (Jafri et al., 1998; Greenstein and Winslow, 2002). The state diagram of this model is presented in Figure 33.2 (A). It consists of 12 states, where state 6 and 12 are the only conducting states and hence define the open states. The transition propensities are defined by a set of functions and constants, which are all described in Greenstein and Winslow (2002).

Ryanodine receptors. RyRs are Ca^{2+} specific channels that are gathered in clusters at the SR membrane in the dyadic cleft. These clusters can consist of several hundred RyRs (Beuckelmann and Wier, 1988; Franzini-Armstrong et al., 1999). They open by single Ca^{2+} ions attaching to the receptors on the cytosolic side. We will use a modified version of a phenomenological RyR model that mimics the physiological functions of the channel (Stern et al., 1999). The model consists of four states where only one, state 2, is conducting; see Figure 33.2 (B). The α and γ propensities are Ca^{2+}-dependent, representing the activation and inactivation dependency of cytosolic [Ca^{2+}]. The β and δ propensities are constants. For specific values of propensities; see Stern et al. (1999).

33.3 Numerical methods for the continuous system

In this section we will describe the numerical methods used to solve the continuous part of the computational model of Ca^{2+} dynamics in the dyadic cleft. We will provide DOLFIN code for each part of the presentation. The first part of the section describes the discretization of the continuous problem using a finite element method. The second part describes a method to stabilize the discretization, and we also conduct a parameter study to find the optimal stabilization parameters.

33.3.1 Discretization

The continuous problem is defined by (33.3 -33.7) together with an initial condition. Given a bounded domain $\Omega \subset \mathbb{R}^3$ with the boundary $\partial\Omega$ we want to find $c = c(x,t) \in \mathbb{R}_+$, for $x \in \Omega$ and $t \in [0,T]$, such that:

$$\begin{cases} \dot{c} &= \sigma\Delta c - \nabla\cdot(ca) & \text{in } \Omega, \\ \sigma\partial_n c - ca\cdot n &= J_k & \text{on } \partial\Omega_k,\ k = 1,\ldots,4, \end{cases} \tag{33.8}$$

and $c(\cdot,0) = c_0(x)$. Here $a = a(x) = 2\sigma E(x)$ and J_k is the k^{th} flux at the k^{th} boundary $\partial\Omega_k$, where $\bigcup_{k=1}^4 \partial\Omega_k = \partial\Omega$, $\partial_n c = \nabla c\cdot n$, where n is the outward normal on the boundary. The J_k are given by (33.5)-(33.7).

The continuous equations are discretized using a finite element method in space. (33.8) is multiplied with a proper test function, v, and integrated over the spatial domain, thus obtaining:

$$\int_\Omega \dot{c}v\,\mathrm{d}x = \int_\Omega \left(\sigma\Delta c - \nabla(ca)\right) v\,\mathrm{d}x. \tag{33.9}$$

Integration by parts together with the boundary conditions in (33.8) yields:

$$\int_\Omega \dot{c}v\,\mathrm{d}x = -\int_\Omega (\sigma\nabla c - ca)\cdot\nabla v\,\mathrm{d}x + \sum_k \int_{\partial\Omega_k} J_k v\,\mathrm{d}s_k. \tag{33.10}$$

Consider a mesh $\mathcal{T}_h = \{T\}$ of simplicial cells T. Let V_h denote the space of piecewise linear polynomials defined relative to the mesh $\mathcal{T}_h$. Using the backward Euler methods in time, we seek an approximation of c: $c_h \in V_h$ with nodal basis $\{\phi_i\}_{i=1}^N$. (33.10) can now be discretized as follows: Consider the n^{th} time step, then given c_h^n find $c_h^{n+1} \in V_h$ such that

$$\int_\Omega \frac{c_h^{n+1} - c_h^n}{\Delta t} v\,\mathrm{d}x = -\int_\Omega \left(\sigma\nabla c_h^{n+1} - c_h^{n+1}a\right)\cdot\nabla v\,\mathrm{d}x + \sum_k \int_{\partial\Omega} J_k v\,\mathrm{d}s_k, \quad \forall\, v \in V_h, \tag{33.11}$$

where Δt is the size of the time step. The trial function $c_h^n(x)$ is expressed as a weighted sum of basis

functions,

$$c_h^n(x) = \sum_j^N C_j^n \phi_j(x). \tag{33.12}$$

where C_j^n are the coefficients. Due to the choice of V_h, the number of unknowns N will coincide with the number of vertices of the mesh.

Taking test function, $v = \phi_i$, $i \in \{1, \ldots, N\}$ gives the following algebraic system of equations in terms of the coefficients $\left\{c_i^{n+1}\right\}_{i=1}^N$:

$$\frac{1}{\Delta t}\boldsymbol{M}\left(C^{n+1} - C^n\right) = \left(-\boldsymbol{K} + \boldsymbol{E} + \sum_k \alpha^k \boldsymbol{F}^k\right) C_j^{n+1} + \sum_k c_0^k f^k. \tag{33.13}$$

Here $C^n \in \mathbb{R}^N$ is the vector of coefficients from the discrete solution, $c_h^n(x)$, α^k and c_0^k are constant coefficients given by (33.5)–(33.7) and

$$\begin{aligned} \boldsymbol{M}_{ij} &= \int_\Omega \phi_i \phi_j \,\mathrm{d}x, & \boldsymbol{K}_{ij} &= \int_\Omega \nabla\phi_i \cdot \nabla\phi_j \,\mathrm{d}x, \\ \boldsymbol{E}_{ij} &= \int_\Omega a\phi_i \cdot \nabla\phi_j \,\mathrm{d}x, & \boldsymbol{F}_{ij}^k &= \int_{\partial\Omega_k} \phi_i \phi_j \,\mathrm{d}s, \end{aligned} \tag{33.14}$$

are the entries in the $\boldsymbol{M}$, $\boldsymbol{K}$, $\boldsymbol{E}$ and $\boldsymbol{F}^k$ matrices. f^k are boundary source vectors corresponding to the k^{th} boundary, with vector elements given by:

$$f_i^k = \int_{\partial\Omega_k} \phi_i \,\mathrm{d}s. \tag{33.15}$$

The following DOLFIN code assembles the matrices and vectors from (33.14)–(33.15):

Python code

```
import numpy as np
from dolfin import *

mesh = Mesh("cleft_mesh.xml.gz")
mesh.order()

Vs = FunctionSpace(mesh, "CG", 1)
Vv = VectorFunctionSpace(mesh, "CG", 1)

v = TestFunction(Vs)
u = TrialFunction(Vs)

# Defining the electric field-function
a = Expression(["0.0","0.0","phi_0*valence*kappa*sigma*exp(-kappa*x[2])"],
               phi_0=-2.2, valence=2, kappa=1, sigma=1.e5,
               element=Vv.ufl_element())

# Assembly of the K, M and A matrices
K = assemble(inner(grad(u), grad(v))*dx)
M = assemble(u*v*dx)
E = assemble(-u*inner(a, grad(v))*dx)

# Collecting face markers from a file, and skip the 0 one
sub_domains = MeshFunction("uint", mesh, "cleft_mesh_face_markers.xml.gz")
unique_sub_domains = list(set(sub_domains.array()))
unique_sub_domains.remove(0)
```

```
# Assemble matrices and source vectors from exterior facets domains
domain = MeshFunction("uint", mesh, 2)
F = {}; f = {}; tmp = K.copy(); tmp.zero()
for k in unique_sub_domains:
    domain.array()[:] = (sub_domains.array() != k)
    F[k] = assemble(u*v*ds, exterior_facet_domains = domain, \
                    tensor = tmp.copy(), reset_sparsity = False)
    f[k] = assemble(v*ds, exterior_facet_domains = domain)
```

In the above code we define only one form for the different boundary mass matrices and boundary source vectors, `u*v*ds` and `v*ds`, respectively. The `assemble` routine will assemble these forms over the 0^{th} subdomain. By passing subdomain specific `MeshFunctions` to the assemble routine, we can assemble the correct boundary mass matrices and boundary source vectors. We collect the matrices and boundary source vectors; these are then added to form the linear system to be solved at each time step. If an LCC opens, the collected source vector from that boundary will contribute to the right-hand side. If an LCC closes the same source vector are removed from the right-hand side. When an LCC either opens or closes, a large flux is either added to or removed from the system. To be able to resolve sharp time gradients correctly we need to take smaller time steps following such an event. After the time step has been reset to a small number we can start expanding it by multiplying the time step with a constant > 1.

The sparse linear system is solved using the PETSc linear algebra backend (Balay et al., 2001) in DOLFIN together with the Bi-CGSTAB iterative solver (van der Vorst, 1992), and the BoomerAMG preconditioners from hypre (Falgout and Yang, 2002). A script that solves the algebraic system from (33.13) together with a crude time stepping scheme for the opening and closing of the included LCC channel is presented below.

33.3.2 Stabilization

It turns out that the algebraic system in (33.13) can be numerically unstable for physiological relevant values of a. This is due to the transport term introduced by $\boldsymbol{E}_{ij}$ from (33.14). We have chosen to stabilize the system using the Streamline upwind Petrov–Galerkin (SUPG) method (Brooks and Hughes, 1982). This method adds an upwind discontinuous contribution to the test function in the streamline direction (33.9):

$$v' = v + s, \text{ where } s = \tau \frac{h\tau_e}{2\|a\|} a \cdot \nabla v. \tag{33.16}$$

Here τ is a parameter we want to optimize (see later in this Section), $\|\cdot\|$ is the Euclidean norm in $\mathbb{R}^3$, $h = h(x)$ is the element size, and $\tau_e = \tau_e(x)$, is given by,

$$\tau_e = \coth(\mathrm{PE_e}) - \frac{1}{\mathrm{PE_e}}, \tag{33.17}$$

where $\mathrm{PE_e}$ is the element Péclet number:

$$\mathrm{PE_e} = \frac{\|a\|h}{2\sigma}. \tag{33.18}$$

When $\mathrm{PE_e}$ is larger than 1 the system becomes unstable, and oscillations are introduced.

In the 1D case with a uniform mesh, the stabilization term defined by (33.17)–(33.18) can give exact nodal solutions (Christie et al., 1976; Brooks and Hughes, 1982). Our choice of stabilization parameter is inspired by this. We have used h to denote the diameter of the sphere that circumscribes the local tetrahedron. This is what DOLFIN implements in the function `Cell.diameter()`. We recognize that

other choices exist, which might give improved stabilization (John and Knobloch, 2007); for example Tezduyar and Park (1986) use a length based on the size of the element in the direction of a.

The DOLFIN code that assembles the SUPG part of the problem is presented in the following script:

Python code

```
cppcode = """class Stab : public Expression {
public:
  double sigma; boost::shared_ptr<GenericFunction> field;
  boost::shared_ptr<Mesh> mesh;
  Stab(): Expression(3), sigma(1.0e5){}
  void eval(Array<double>& v, const Array<double>& x,
            const ufc::cell& c) const {
    Cell cell(*mesh, c.index);
    double field_norm = 0.0; double tau = 0.0;
    double h = cell.diameter();
    field->eval(v, x, c);
    for (uint i = 0;i < x.size(); ++i)
      field_norm += v[i]*v[i];
    field_norm = sqrt(field_norm);
    double PE = 0.5*field_norm * h/sigma;
    if (PE > DOLFIN_EPS)
      tau = 1/tanh(PE)-1/PE;
    for (uint i = 0;i < x.size(); ++i)
      v[i] *= 0.5*h*tau/field_norm;}};
"""
stab = Expression(cppcode); stab.field = a; stab.mesh = mesh

# Assemble the stabilization matrices
E_stab = assemble(div(a*u)*inner(stab, grad(v))*dx)
M_stab = assemble(u*inner(stab, grad(v))*dx)

# Adding them to the A and M matrices, weighted by the global tau
tau = 0.28; E.axpy(tau, E_stab, True); M.axpy(tau, M_stab, True)
```

In the above script, two matrices `E_stab` and `M_stab` are assembled. Both matrices are added to the corresponding advection and mass matrices `E` and `M`, weighted by the global parameter `tau`.

A mesh with finer resolution close to the TT surface, at $z = 0$ nm, is used to resolve the steep gradient of the solution in this area. It is here the electric field is at its strongest yielding an element Péclet number larger than 1. However the field attenuates quickly: at $z = 3$ nm the field is down to 5% of the maximum amplitude, and at $z = 5$ nm it is down to 0.7%. The mesh can thus be fairly coarse in the interior of the domain. The mesh generator `tetgen` is used to to produce meshes with the required resolution (Si, 2007).

33.3.3 *Solving the discretized system*

The DOLFIN code that solves the discretized and stabilized system from (33.13) is given by:

Python code

```
# Model parameters
dt_min = 1.0e-10; dt = dt_min; t = 0; c0 = 0.1; tstop = 1.0
events = [0.2,tstop/2,tstop,tstop]; dt_expand = 2.0;
sigma = 1e5; ds = 50; area = pi; Faraday = 0.0965; amp = -0.1
t_channels = {1:[0.2,tstop/2], 2:[tstop/2,tstop]}

# Initialize the solution Function and the left and right hand side
u = Function(Vs); x = u.vector()
```

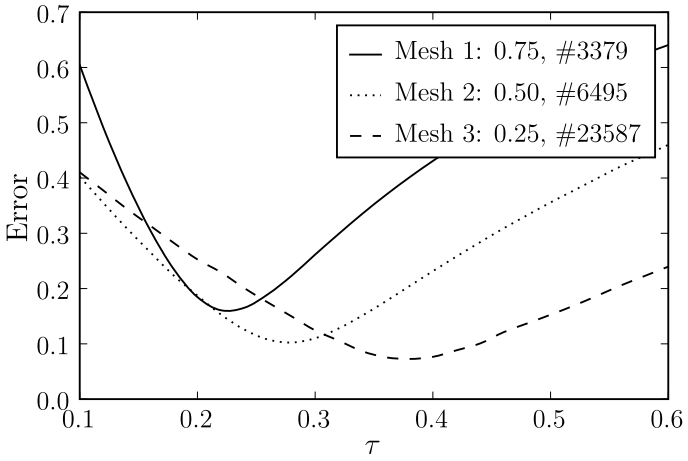


Figure 33.3: The figure shows a plot of the error versus the stabilization parameter τ for 3 different mesh resolutions. The mesh resolutions are given by the median of the z distance of all vertices and the total number of vertices in the mesh; see legend. We see that the minimal values of the error for the three meshes occur at three different τ: 0.22, 0.28, and 0.38.

```
x[:] = c0*np.exp(-a.valence*a.phi_0*np.exp(-a.kappa*mesh.coordinates()[:,-1]))
b = Vector(len(x)); A = K.copy();

solver = KrylovSolver("bicgstab","hypre_amg")
solver.parameters["relative_tolerance"] = 1e-10
solver.parameters["absolute_tolerance"] = 1e-7

plot(u, vmin=0, vmax=4000, interactive=True)
while t < tstop:
    # Initalize the left and right hand side
    A.assign(K); A *= sigma; A += E; b[:] = 0

    # Adding channel fluxes
    for c in [1,2]:
        if t >= t_channels[c][0] and t < t_channels[c][1]:
            b.axpy(-amp*1e9/(2*Faraday*area),f[c])

    # Adding cytosole flux at Omega 3
    A.axpy(sigma/ds,F[3],True); b.axpy(c0*sigma/ds,f[3])

    # Applying the Backward Euler time discretization
    A *= dt; b *= dt; b += M*x; A += M

    solver.solve(A,x,b)
    t += dt; print "Ca Concentration solved for t:",t

    # Handle the next time step
    if t == events[0]:
        dt = dt_min; events.pop(0)
    elif t + dt*dt_expand > events[0]:
        dt = events[0] - t
    else:
        dt *= dt_expand

    plot(u, vmin=0, vmax=4000)

plot(u, vmin=0, vmax=4000, interactive=True)
```

The time stepping scheme presented in the above code is crude, but simple and explicit. The solution algorithm is based on pre-assembled matrices. Adding matrices and vectors together makes the

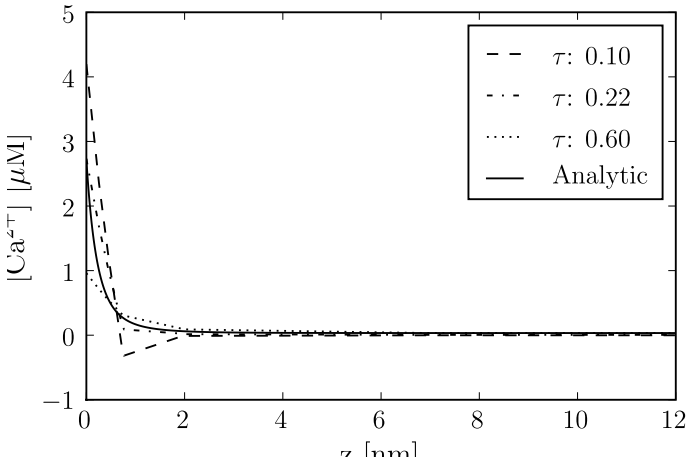


Figure 33.4: The figure shows concentration results from numerical solutions from Mesh 1 (see legend of Figure 33.3), for three different τ, together with the analytical solution. The solutions were picked from a line between the points (0,0,0) and (0,0,12). We see that the solution with $\tau = 0.10$ oscillates. The solution with $\tau = 0.22$ was the solution with smallest global error for this mesh (see Fig 33.3), and the solution with $\tau = 0.60$ undershoots the analytical solution at $z = 0$ nm with $\simeq$1.7 μM.

construction of the linear system more complicated compared to including the time discretization directly into a variational form. However, by pre-assemble the matrices and source vectors we do not have to reassemble the linear system during the time stepping, and time is saved during execution. This becomes important when larger meshes and hundred of channels are included.

33.3.4 Finding an optimal stabilization parameter

The global stabilization parameter, τ, is problem-dependent. To find an optimal τ for a certain electric field and mesh, the system in (33.13) is solved to steady state, defined as $T = 1.0\,\text{ms}$, using only homogeneous Neumann boundary conditions. A homogeneous concentration of $c_0 = 0.1$ μM is used as the initial condition. The numerical solution is then compared with the analytical solution of the problem. This solution is acquired by setting $J = 0$ in (33.2) and solving for the c, with the following result:

$$c(z) = c_b \mathrm{e}^{-2\psi(z)}. \tag{33.19}$$

Here ψ is given by (33.4), and c_b is the bulk concentration; that is, where z is large. c_b was chosen such that the integral of the analytical solution was equal to $c_0 \times V$, where V is the volume of the domain.

The error of the numerical solution for different values of τ and for three different mesh resolutions is plotted in Figure 33.3. The meshes are enumerated from 1-3, and a higher number corresponds to a better resolved boundary layer at z=0 nm. As expected, we see that the mesh that resolves the boundary layer best produces the smallest error. The error is computed using the $L^2(\Omega)$ norm and is normalized by the $L^2(\Omega)$ norm of the analytical solution,

$$\frac{\|c(T) - c_h^{n_T}\|_{L^2}}{\|c(T)\|_{L^2}}, \tag{33.20}$$

where n_T is the time step at $t = T$. The mesh resolutions are quantified by the number of vertices close to $z = 0$. In the legend of Figure 33.3, the median of the z distance of all vertices and the total number of vertices in each mesh is presented.

Traces from the actual simulations are plotted in Figure 33.4-33.6. In each figure are three numerical and one analytical solution plotted for each mesh. The numerical solutions are from simulations using

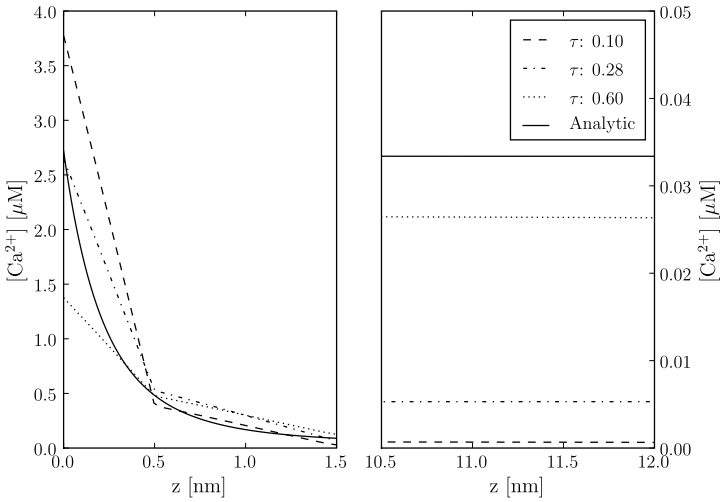


Figure 33.5: The figures show the concentration traces of the numerical solutions from Mesh 2 (see legend of Figure 33.3), for three different τ, together with the analytical solution. The solution traces in the two panels are picked from a line between the points (0,0,0) and (0,0,1.5), for the left panel, and between spatial points (0,0,10.5) and (0,0,12) for the right panel. We see from both panels that the solution for $\tau = 0.10$ gives the poorest approximation. The solution with $\tau = 0.28$ was the solution with smallest global error for this mesh (see Fig 33.3), and this is reflected in the reasonable good fit seen in the left panel, especially at $z = 0$ nm. The solution with $\tau = 0.60$ undershoots the analytical solution at $z = 0$ with $\simeq$1.2 μM. From the right panel we see that all numerical solutions undershoot at $z = 12$ nm. We also see that the trace for $\tau = 0.60$ comes the closest to the analytical solution.

three different τ: 0.1, 0.6 and the L^2-optimal τ (see Figure 33.3). The traces in the figures are from the discrete solution $c_h^{n_T}$, interpolated onto the straight line between the spatial points p_0=(0,0,0) and p_1=(0,0,12).

In Figure 33.4 the traces from Mesh 1 are plotted. Here we see that the numerical solutions are quite poor for some τ. The solution with $\tau = 0.10$ is not correct as it produces negative concentrations; a physiological impossibility. The solution with $\tau = 0.60$ seems more correct, but it undershoots the analytical solution at $z = 0$ with $\simeq$1.7 μM. The solution with $\tau = 0.22$ is the L^2-optimal solution for Mesh 1, and approximates the analytical solution at $z = 0$ well.

In Figure 33.5, the traces from Mesh 2 are presented in two plots. The left plot shows the traces for $z < 1.5$ nm, and the right shows traces for $z > 10.5$ nm. In the left plot, we see the same tendency as in Figure 33.4: an overshoot of the solution with $\tau = 0.10$ and an undershoot of the solution with $\tau = 0.60$. The L^2-optimal solution for $\tau = 0.28$, overshoots the analytical solution for the shown interval in the left plot, but undershoots for the rest of the trace.

In the last figure, Figure 33.6, traces from mesh 3 are presented. The results are also presented in two plots here, corresponding to the same z interval as in Figure 33.5. We see that the solution with $\tau = 0.10$ is again not acceptable in either interval. In the left plot, it clearly overshoots the analytical solution for most of the interval, and then undershoot the analytical solution for the rest of the interval. The solution with $\tau = 0.60$ is improved here compared to the two previous plots. It undershoots the analytical solution at $z = 0$; but stays closer for the rest of the interval as compared to the L^2-optimal solution. The L^2 norm penalizes larger distances between two traces; that is, weighting the error close to $z = 0$ more than the rest. The optimal solution measured in the Max norm is given when $\tau = 50$ (result not shown).

The numerical results tell us that the Streamline upwind Petrov–Galerkin method can be used to stabilize the Finite element solution of the advection–diffusion problem presented in (33.8). Three

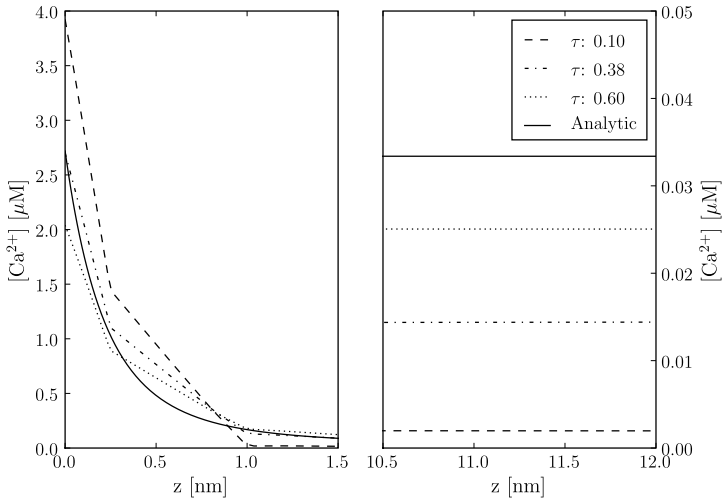


Figure 33.6: The figure shows the concentration traces of the numerical solutions from Mesh 3 (see legend of Figure 33.3), for three different τ, together with the analytical solution. The traces in the two panels were picked from the same lines for Figure 33.5. Again, we see from both panels that the solution with $\tau = 0.10$ give the poorest solution. The solution with $\tau = 0.38$ was the solution with smallest global error for this mesh (see Fig 33.3), and this is reflected in the good fit seen in the left panel, especially at $z = 0$nm. The solution with $\tau = 0.60$ undershoots the analytical solution at $z = 0$ with $\simeq 0.7$ μM. From the right panel, we see that all numerical solutions undershoot at $z = 15$ nm, and the trace with $\tau = 0.60$ here also comes closest to the analytical solution.

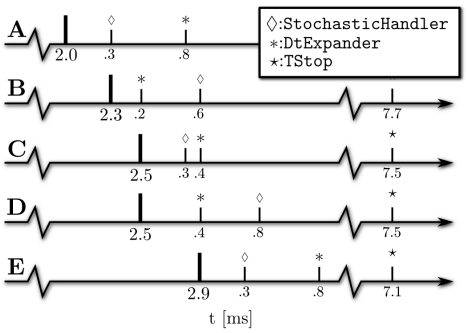


Figure 33.7: Diagram for the time stepping algorithm using 3 discrete objects: `DtExpander`, `StochasticHandler`, `TStop`. The values below the small ticks corresponds to the time to the next event for each of the discrete objects. This time is measured from the last realized event, which is denoted by the thicker tick. See text for details.

different meshes that resolve the boundary layer at $z = 0$ nm differently were used. For each mesh a global τ, which produce an L^2 optimal solution, were obtained. To test convergence rate we also did simulations with homogeneously refined meshes. The largest mesh had $\simeq$180 000 number of vertices. The errors for the optimal τ for each mesh resolution were compared and linear convergence rate was obtained (result not shown).

The largest mesh in our test problems, the one that resolves the boundary layer best, is not large: $\simeq$ 24 000 vertices. The convergence study we performed showed that we could decrease the reported error more by using meshes with better resolution. However, the meshes we ran our simulations on, where physiological small meshes; radius=20nm. A relevant size would need a radius of $\simeq$200 nm. This would create a mesh with $\simeq$2.5 million vertices for the highest resolution we use in this chapter. A mesh with such a size would be a challenge for a serial solver, and parallel solvers need to be employed. The software that will be presented next, `diffsim`, does unfortunately not support parallel solvers.

33.4 `diffsim`: an event-driven simulator

In the DOLFIN scripts above, we show how a simple continuous solver can be built with DOLFIN. By pre-assembling the matrices from (33.14), a flexible system for adding and removing boundary fluxes corresponding to the state of the channels can be constructed. The solving script uses fixed time points for the channel state transitions. At these time points, we minimize Δt so we can resolve the sharp time gradient. In between the channel transitions we expand Δt. This simplistic time stepping scheme has been sufficient to solve the presented example. However, it would be difficult to extend this to incorporate the time stepping involved with the solution of stochastic Markov models and other discrete variables. For such scenarios, an event-driven simulator called `diffsim` has been developed. In the final subsections in this chapter the algorithm underlying the time stepping scheme in `diffsim` will be presented. An example of how one can use `diffsim` to describe and solve a model of Ca^{2+} dynamics in the dyadic cleft is also demonstrated.

33.4.1 Stochastic system

The stochastic evolution of the Markov chain models presented in Section 33.2.3 is determined by a modified Gillespie method (Gillespie, 1977), which resembles one presented in Rüdiger et al. (2007). Here we will not go into detail, but rather focus on the part of the method that has importance for the overall time stepping algorithm.

The solution of the included stochastic Markov chain models is stored in a state vector `S`. Each element in `S` corresponds to one Markov model, and the value reflects which state each model is in. The transitions between these states are modeled stochastically and are computed using a modified Gillespie method. Basically this method gives us which of the states in `S` changes to what state and when. Not all such state transitions are relevant for the continuous system. A transition between two closed states in the LCC model, for instance, will not have any impact on the boundary fluxes, and can be ignored. Only transitions that either open or close a channel (channel transitions), will be recognized. The modified Gillespie method assumes that any continuous variables on which a certain propensity function depends are constant during a time step. The error incurred by this assumption is reduced by taking smaller time steps right after a channel transition as the continuous field is indeed changing dramatically during this time period.

33.4.2 Time stepping algorithm

To simplify the presentation of the time stepping algorithm, we only consider one continuous variable, the Ca^{2+} field. A Python-like pseudo code for the time stepping algorithm is shown in the following script:

Python code

```
# Python-like pseudo code for the time stepping algorithm used in diffsim
while not stop_sim:
    # The next event
    event = min(discrete_objects)
    dt = event.next_time()

    # Step the event and check result
    while not event.step():
        event = min(discrete_objects)
        dt = event.next_time()

    # Update the other discrete objects with dt
    for obj in discrete_objects:
        obj.update_time(dt)
```

```
    # Solve the continuous equation
    ca_field.solve(dt)
    ca_field.send()

    # Distribute the event
    event.send()
```

The framework presented with this pseudo code can be expanded to handle several continuous variables. We define a base class called `DiscreteObject`, which defines the interface for all discrete objects. A key function of a discrete object is to know when its *next event* is due. The `DiscreteObject` that has the smallest next event time gets to define the size of the next Δt. In Python, this is easily achieved by making the `DiscreteObjects` sortable with respect to their next event time. All `DiscreteObjects` are then collected in a list `discrete_objects` (see script below). The `DiscreteObject` with the smallest next event time is then simply `min(discrete_objects)`. An event from a `DiscreteObject` that does not have an impact on the continuous solution will be ignored; for example, a Markov chain model transition that is not a channel transition as noted above. A transition needs to be realized before we can tell if it is a channel transition or not. This is done by *stepping* the `DiscreteObject`; that is, calling the object's `step()` method. If the method returns `False` it will not affect the Ca^{2+} field. We then enter a while loop and a new `DiscreteObject` is picked. If the object returns `True` when stepped we exit the loop and continue. Next, we have to update the other discrete objects with the chosen Δt, solve for the Ca^{2+} field, broadcast the solution, and last but not least, execute the discrete event that is scheduled to happen at Δt.

In Figure 33.7, we show an example of a possible realization of this algorithm. In (A) we have realized a time event at $t = 2.0\,\text{ms}$. The next event to be realized is a stochastic transition, the one with smallest value below the ticks. In (B) this event is realized, and the `StochasticHandler` now shows a new next event time. The event is a channel transition forcing the dt, controlled by the `DtExpander`, to be minimized. `DtExpander` now has the smallest next event time, and is realized in (C). The channel transition that was realized in (B) raised the $[Ca^{2+}]$ in the cleft, which in turn increased the Ca^{2+}-dependent propensity functions in the included Markov models. The time to next event time of the `StochasticHandler` has therefore been updated, and moved forward in (C). Also note that the `DtExpander` has expanded its next event time. In (D), the stochastic transition is realized and updated with a new next event time, but it is ignored as it is not a channel transition. The smallest time step is now the `DtExpander`, and this is realized in (E). In this example, we do not realize the `TStop` event, as it is too far away.

33.4.3 *`diffsim`: an example*

`diffsim` is a versatile, event-driven simulator that incorporates the time stepping algorithm presented in the previous section together with the infrastructure to solve models with one or more diffusional domains defined by a computational mesh. Each such domain can have several diffusive ligands. Custom fluxes can easily be included through the framework. The submodule `dyadiccleft` implements some published Markov models that can be used to simulate the stochastic behavior of a dyad and some convenient boundary fluxes. It also implements the field flux from the lipid bi-layer discussed in Section 33.2.2. The following script runs 10 simulations to collect the time to release, also called the latency, for a dyad:

Python code

```
# An example of how diffsim can be used to simulate the time to RyR release latency, from
# a small dyad who's domain is defined by the mesh in the file cleft_mesh_with_RyR.xml.gz
from diffsim import *
from diffsim.dyadiccleft import *
```

```
from numpy import exp, from file

# Model parameters
c0_bulk = 0.1; D_Ca = 1.e5; Ds_cyt = 50; phi0 = -2.2; tau = 0.28
AP_offset = 0.1; dV = 0.5, ryr_scale = 100; end_sim_when_opend = True

# Setting boundary markers
LCC_markers = range(10,14); RyR_markers = range(100,104); Cyt_marker = 3

# Add a diffusion domain
domain = DiffusionDomain("Dyadic_cleft","cleft_mesh_with_RyR.xml.gz")
c0_vec = c0_bulk*exp(-VALENCE[Ca]*phi0*exp(-domain.mesh().coordinates()[:,-1]))

# Add the ligand with fluxes
ligand = DiffusiveLigand(domain.name(),Ca,c0_vec,D_Ca)
field  = StaticField("Bi_lipid_field",domain.name())
Ca_cyt = CytosolicStaticFieldFlux(field,Ca,Cyt_marker,c0_bulk,Ds_cyt)

# Adding channels with Markov models
for m in LCC_markers:
    LCCVoltageDepFlux(domain.name(), m, activator=LCCMarkovModel_Greenstein)
for m in RyR_markers:
    RyRMarkovModel_Stern("RyR_%d"%m, m, end_sim_when_opend)

# Adding a dynamic voltage clamp that drives the LCC Markov model
AP_time = fromfile("AP_time_steps.txt",sep="\n")
dvc = DynamicVoltageClamp(AP_time,fromfile("AP.txt",sep="\n"),AP_offset,dV)

# Get and set parameters
params = get_params()

params.io.save_data = True
params.Bi_lipid_field.tau = tau
params.time.tstop = AP_time[-1] + AP_offset
params.RyRMarkovChain_Stern.scale = ryr_scale

info(str(params))

# Run 10  simulations
data = run_sim(10,"Dyadic_cleft_with_4_RyR_scale")
mean_release_latency = mean([ run["tstop"] for run in data["time"]])
```

The two Markov models presented in Section 33.2.3 are here used to model the stochastic dynamics of the RyRs and the LCCs. The simulation is driven by a so-called dynamic voltage clamp. With a voltage clamp we can dynamically clamp the voltage to a certain wave form. The wave form can be acquired from experiments. The data that define the voltage clamp are read from a file using utilities from NumPy Python packages.

33.5 Discussion

We have presented a computational model of Ca^{2+} dynamics of the dyadic cleft in heart cells. It consists of a coupled stochastic and continuous system. We have showed how one can use DOLFIN to discretise and solve the continuous system using a finite element method. Because the continuous system is an advection–diffusion equation that produces unstable discretizations, we investigate how one can use the SUPG method for stability. We employ three different meshes each with different resolutions at the boundary layer of the electrical potential, and find an L^2-optimal global stabilization parameter τ for each mesh.

We do not present a solver for the stochastic system. However, we outline a time stepping scheme that can be used to couple the stochastic solver with solver presented for the continuous system. A simulator `diffsim` is briefly introduced, which implements the presented time stepping scheme together with the presented solver for the continuous system.

34 *Electromagnetic waveguide analysis*

By Evan Lezar and David B. Davidson

At their core, Maxwell's equations are a set of differential equations describing the interactions between electric and magnetic fields, charges and currents. These equations provide the tools with which to predict the behavior of electromagnetic phenomena, giving us the ability to use them in a wide variety of applications, including communication and power generation. Due to the complex nature of typical problems in these fields, numeric methods such as the finite element method are often employed.

One of the earliest applications of the finite element method in electromagnetics was in Silvester (1969), where it was applied to the analysis of waveguide structures. These structures are typically bounded structures — although open waveguides do exist — for which a countably infinite number of modes satisfy Maxwell's equations and their associated boundary conditions (Pozar, 2005). The finite element analysis of these structures is concerned with calculating these waveguide modes, which are generally characterized by a complex propagation constant as well as an associated electromagnetic field distribution (which may both be a function of frequency). The formulation adopted in this work is that of Lee et al. (1991) for lossless materials, with an extension to the lossy case presented in Lee (1994). An overview of the state-of-the-art in the field is presented in Davidson (2011). Alternate formulations are discussed subsequently.

Since waveguides are some of the most common structures in microwave engineering, especially in areas where high power and low loss are essential (Pozar, 2005), their analysis is still a topic of much interest. This chapter considers the use of FEniCS in the cutoff and dispersion analysis of these structures. These types of analysis form an important part of the design and optimization of waveguide structures for a particular purpose. In these kinds of waveguide problems, the solution of generalized eigensystems are required with the eigenvalues and eigenvectors of the systems associated with the waveguide modes that are of interest.

The aim of this chapter is to guide the reader through the process followed in implementing solvers for various electromagnetic problems with both cutoff and dispersion analysis considered in depth. To this end, a brief introduction to electromagnetic waveguide theory, the mathematical formulation of these problems, and the specifics of their solutions using the finite element method are presented in Section 34.1. This lays the groundwork for a discussion of the details pertaining to the FEniCS implementation of these solvers, covered in Section 34.2. The translation of the finite element formulation to FEniCS, as well as some post-processing considerations are covered. In Section 34.3, the solution results for three typical waveguide configurations are presented and compared to analytical or previously published data. This serves to validate the implementation and illustrates the kinds of problems that can be solved.

34.1 Formulation

In electromagnetics, the behavior of the electric and magnetic fields are described by Maxwell's equations (Jin, 2002; Smith, 1997). Using these partial differential equations, various boundary value problems can be obtained depending on the problem being solved. In the case of time-harmonic fields, the equation used is the vector Helmholtz wave equation. If the problem is further restricted to a domain surrounded by perfect electrical or magnetic conductors (as is the case in general waveguide problems) the wave equation in terms of the electric field, E, can be written as (Jin, 2002)

$$\nabla \times \frac{1}{\mu_r} \nabla \times E - k_o^2 \epsilon_r E = 0 \qquad \text{in } \Omega_v, \tag{34.1}$$

subject to the boundary conditions

$$\hat{n} \times E = 0 \qquad \text{on } \Gamma_e, \tag{34.2}$$

$$\hat{n} \times \nabla \times E = 0 \qquad \text{on } \Gamma_m, \tag{34.3}$$

with Ω_v representing the interior of the waveguide and Γ_e and Γ_m electric and magnetic walls, respectively. Here, μ_r and ϵ_r are the relative permeability and relative permittivity, respectively. These are material parameters that may be inhomogeneous (varying in space) but only the isotropic case is considered here. In this case, isotropic means that the medium's response is the same for all directions of the electric field vector (Ramo et al., 1994). It should be noted that the formulations discussed here can also be extended to the anisotropic case as in Polycarpou et al. (1996).

In (34.1), k_o is the operating wavenumber which is related to the operating frequency (f_o) by the expression

$$k_o = \frac{2\pi f_o}{c_0}, \tag{34.4}$$

with c_0 the speed of light in free space. This boundary value problem can also be written in terms of the magnetic field as in Jin (2002), but since the discussions that follow are applicable to both formulations, this will not be considered here.

One way to solve the boundary value problem is to find the stationary point of the following variational functional

$$F(E) = \frac{1}{2} \int_{\Omega_v} \left[\frac{1}{\mu_r} (\nabla \times E) \cdot (\nabla \times E) - k_o^2 \epsilon_r E \cdot E \right] \mathrm{d}x, \tag{34.5}$$

which can be found in a number of computational electromagnetic texts, including those by Jin (2002) and Pelosi et al. (1998), as well as the paper by Lee et al. (1991). In the case of the waveguide problems considered here, a number of simplifications can be made to the solution process and these will now be discussed. Note that for this source-free formulation, the $\frac{1}{2}$ factor in (34.5) is superfluous, and is subsequently dropped.

If the guide is sufficiently long, and the z-axis is chosen parallel to its cylinder axis as shown in Figure 34.1, then the z-dependence of the electric field can be assumed to be of the form $e^{-\gamma z}$ with $\gamma = \alpha + j\beta$ a complex propagation constant (Pelosi et al., 1998; Pozar, 2005). Making this assumption and splitting the electric field into transverse ($E_t = \hat{x}E_x + \hat{y}E_y$) and axial ($\hat{z}E_z$) components, results in the following expression for the field

$$E(x, y, z) = [E_t(x, y) + \hat{z}E_z(x, y)]e^{-\gamma z}, \tag{34.6}$$

with x and y the Cartesian coordinates in the cross sectional plane of the waveguide and z the

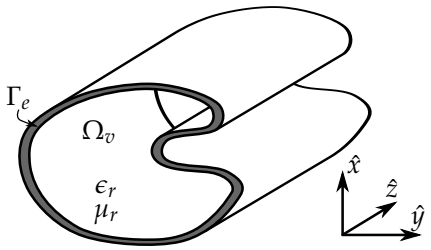


Figure 34.1: A long waveguide with an arbitrary cross section aligned with the z-axis. Note the labels for the domain corresponding to the waveguide interior (Ω_v) as well as the electric wall Γ_e.

coordinate along the length of the waveguide. Here $\hat{x}$, $\hat{y}$, and $\hat{z}$, represent unit vectors in the x, y, and z-direction, respectively. For the purpose of this discussion, also consider the following representation of ∇ in Cartesian coordinates

$$\nabla = \nabla_t + \nabla_z, \tag{34.7}$$

with

$$\nabla_t = \frac{\partial}{\partial x}\hat{x} + \frac{\partial}{\partial y}\hat{y}, \tag{34.8}$$

the transverse gradient, and

$$\nabla_z = \frac{\partial}{\partial z}\hat{z}, \tag{34.9}$$

the partial derivative with respect to z in the z-direction.

By substituting the expression for the field in (34.6) as well as the decomposition of ∇ of (34.7) into the functional of (34.5) and performing a number of vector manipulations, the following modified functional can be obtained

$$F(E) = \int_\Omega \frac{1}{\mu_r}(\nabla_t \times E_t)\cdot(\nabla_t \times E_t) - k_o^2\epsilon_r E_t \cdot E_t + \frac{1}{\mu_r}(\nabla_t E_z + \gamma E_t)\cdot(\nabla_t E_z + \gamma E_t) - k_o^2\epsilon_r E_z E_z \,\mathrm{d}x. \tag{34.10}$$

Note that in this case the integration domain (Ω_v) of (34.5) – representing the entire waveguide interior volume – has been replaced by integration over the waveguide cross section – indicated by the domain Ω in (34.10) – for an arbitrary z position. Functionals similar to the one shown in (34.10) are employed in Lee et al. (1991), Jin (2002), and Pelosi et al. (1998), although in the latter case, this is derived by substituting (34.6) into the original Helmholtz equation of (34.1).

Using two dimensional curl-conforming vector basis functions (N_i) for the discretization of the transverse field (such as the basis functions from the Nédélec function space of the first kind (Nédélec, 1980; Webb, 1993; Monk, 2003)), and nodal scalar basis functions (L_i) for the axial components (Jin, 2002; Pelosi et al., 1998), the discretized field components (indicated by the h subscript) of (34.6) are given by (Jin, 2002; Pelosi et al., 1998)

$$E_{t,h} = \sum_{i=1}^{N_N} (e_t)_i N_i, \tag{34.11}$$

$$E_{z,h} = \sum_{i=1}^{N_L} (e_z)_i L_i. \tag{34.12}$$

Here $(e_t)_i$ and $(e_z)_i$ are the coefficient of the i^{th} vector and scalar basis functions, respectively, while N_N and N_L are the total number of each type of basis function used in the discretization. The letters N and L are chosen for the basis function names as a reminder that the basis functions come from a

Nédélec function space and a Lagrange polynomial space respectively. A discussion on these and other basis functions is presented in Chapter 3.

The formulation used here, where the electric field in the waveguide is expressed as a combination of transverse and axial components, is probably one of the most widely used in practice. A number of other approaches have also been taken, with other vector formulations (most notably that of Davies in Itoh et al. (1996)) discussed by Dillon and Webb (1994). Other formulations, for instance, involve only nodal elements; some use the axial fields as the working variable; and the problem has also been formulated in terms of potentials, rather than fields. A good summary of these may be found in Chapter 9 of Zhu and Cangellaris (2006).

34.1.1 Waveguide cutoff analysis

One of the simplest cases to consider, and often a starting point when testing a new finite element implementation, is waveguide cutoff analysis. When a waveguide is operating at cutoff, the electric field is constant along the z-axis which corresponds to $\gamma = 0$ in (34.6) (Pozar, 2005). Substituting $\gamma = 0$ into (34.10) yields the following functional for cutoff analysis

$$F_c(E) = \int_\Omega \frac{1}{\mu_r}(\nabla_t \times E_t) \cdot (\nabla_t \times E_t) - k_c^2 \epsilon_r E_t \cdot E_t + \frac{1}{\mu_r}(\nabla_t E_z) \cdot (\nabla_t E_z) - k_c^2 \epsilon_r E_z E_z \, \mathrm{d}x. \tag{34.13}$$

The symbol for the operating wavenumber, k_o, has been replaced with k_c, with the c subscript indicating that the quantity of interest is now the cutoff wavenumber. This quantity and the electric field distribution at cutoff are of interest in these kinds of problems.

Substituting the discretized field equations of (34.11) and (34.12) into the functional (34.13) and applying a minimization procedure, yields the following matrix equation (Davidson, 2011)

$$\begin{bmatrix} S_{tt} & 0 \\ 0 & S_{zz} \end{bmatrix} \begin{Bmatrix} e_t \\ e_z \end{Bmatrix} = k_c^2 \begin{bmatrix} T_{tt} & 0 \\ 0 & T_{zz} \end{bmatrix} \begin{Bmatrix} e_t \\ e_z \end{Bmatrix}, \tag{34.14}$$

or simply

$$\left[S\right] \left\{e\right\} = k_c^2 \left[T\right] \left\{e\right\}. \tag{34.15}$$

The matrix equation of (34.14) is in the form of a generalized eigenvalue problem with the square of the cutoff wavenumber the (unknown) eigenvalue. The submatrices S_{oo} and T_{oo} (with $oo = tt, zz$) represent the stiffness and mass matrices common in the finite element literature (Davidson, 2011; Jin, 2002). The subscripts tt and zz indicate transverse and axial components, respectively. The entries of the matrices of (34.14) are defined as (Pelosi et al., 1998; Jin, 2002)

$$(S_{tt})_{ij} = \int_\Omega \frac{1}{\mu_r}(\nabla_t \times N_i) \cdot (\nabla_t \times N_j) \, \mathrm{d}x, \tag{34.16}$$

$$(T_{tt})_{ij} = \int_\Omega \epsilon_r N_i \cdot N_j \, \mathrm{d}x, \tag{34.17}$$

$$(S_{zz})_{ij} = \int_\Omega \frac{1}{\mu_r}(\nabla_t L_i) \cdot (\nabla_t L_j) \, \mathrm{d}x, \tag{34.18}$$

$$(T_{zz})_{ij} = \int_\Omega \epsilon_r L_i L_j \, \mathrm{d}x, \tag{34.19}$$

with $\int_\Omega \cdot \, \mathrm{d}x$ representing integration over the cross section of the waveguide.

In (34.14) the possible cutoff wavenumbers k_c are the square roots of the eigenvalues of the system and the elements of the corresponding eigenvectors are the coefficient of the basis functions as in (34.11) and (34.12). As such, the solution of the eigensystem not only allows for the computation

of the cutoff wavenumbers, but also for the visualization of the fields associated with the modes by substituting the elements of the computed eigenvector into (34.11) and (34.12). It should also be noted that transverse electric (TE) modes will have zeros as coefficients for the scalar basis functions ($\{e_z\} = 0$) whereas transverse magnetic (TM) modes will have $\{e_t\} = 0$, although this condition only holds at cutoff (Pozar, 2005).

34.1.2 Waveguide dispersion analysis

In the case of cutoff analysis discussed in 34.1.1, one attempts to obtain the value of $k_o^2 = k_c^2$ for a given propagation constant γ, namely $\gamma = 0$. For most waveguide design applications however, k_o is specified and the propagation constant is calculated from the resultant eigensystem (Jin, 2002; Pelosi et al., 1998). This calculation can be simplified somewhat by making the following substitution into (34.10) (after multiplying by γ^2)

$$E_{t,\gamma} = \gamma E_t, \tag{34.20}$$

which yields the modified functional

$$F_\gamma(E) = \int_\Omega \frac{1}{\mu_r}(\nabla_t \times E_{t,\gamma}) \cdot (\nabla_t \times E_{t,\gamma}) - k_o^2 \epsilon_r E_{t,\gamma} \cdot E_{t,\gamma} + \gamma^2 \left[\frac{1}{\mu_r}(\nabla_t E_z + E_{t,\gamma}) \cdot (\nabla_t E_z + E_{t,\gamma}) - k_o^2 \epsilon_r E_z E_z \right] \mathrm{d}x. \tag{34.21}$$

Using the same discretization as for the cutoff analysis discussed in the preceding section, the matrix equation associated with the solution of the variational problem is given by (Pelosi et al., 1998)

$$\begin{bmatrix} A_{tt} & 0 \\ 0 & 0 \end{bmatrix} \begin{Bmatrix} e_t \\ e_z \end{Bmatrix} = -\gamma^2 \begin{bmatrix} B_{tt} & B_{tz} \\ B_{zt} & B_{zz} \end{bmatrix} \begin{Bmatrix} e_t \\ e_z \end{Bmatrix}, \tag{34.22}$$

with

$$A_{tt} = S_{tt} - k_o^2 T_{tt}, \tag{34.23}$$

$$B_{zz} = S_{zz} - k_o^2 T_{zz}, \tag{34.24}$$

which is also in the form of a generalized eigenvalue problem with the eigenvalues a function of the square of the complex propagation constant (γ).

The matrices S_{tt}, T_{tt}, S_{zz}, and T_{zz} are identical to those defined for the waveguide cutoff analysis of the previous section, with entries given by (34.16), (34.17), (34.18), and (34.19), respectively. The entries of the other submatrices, B_{tt}, B_{tz}, and B_{zt}, are defined by

$$(B_{tt})_{ij} = \int_\Omega \frac{1}{\mu_r} N_i \cdot N_j \, \mathrm{d}x, \tag{34.25}$$

$$(B_{tz})_{ij} = \int_\Omega \frac{1}{\mu_r} N_i \cdot \nabla_t L_j \, \mathrm{d}x, \tag{34.26}$$

$$(B_{zt})_{ij} = \int_\Omega \frac{1}{\mu_r} \nabla_t L_i \cdot N_j \, \mathrm{d}x. \tag{34.27}$$

A common challenge in electromagnetic eigenvalue problems such as these is the occurrence of spurious modes which are discussed in Jin (2002) and Davidson (2011). These are non-physical modes that fall in the null space of the $\nabla \times \nabla \times$ operator of (34.1) (Bossavit, 1998), with the issue of spurious modes revisited in the work by Fernandes and Raffetto (2002).

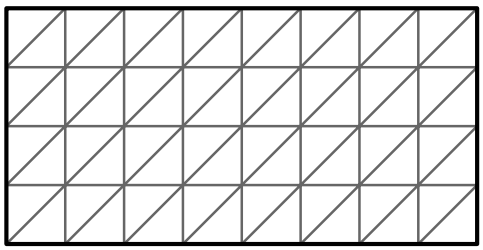


Figure 34.2: An illustration of the finite element mesh used for the rectangular waveguide problems considered here. The mesh corresponds to 64 triangular elements and it should be noted that only the interior of the waveguide is meshed.

One of the strengths of the curl-conforming vector basis functions (edge elements) used in the discretization of the transverse component of the field, is that they allow for a better representation of the null-space in question and improve the modelling of singularities when compared to nodal basis functions (Webb, 1993). This means that the null-space modes can be more readily identified (Davidson, 2011; Jin, 2002). A number of other solutions to the problem have been proposed. These include the use of Lagrange multipliers as in Vardapetyan and Demkowicz (2002), the use of a divergence term to regularize the $\nabla \times \nabla \times$ operator in the functional of (34.5) (Costabel and Dauge, 2002), and the use of a discontinuous Galerkin formulation as presented in Buffa et al. (2007), but are not discussed further in this chapter.

34.2 Implementation

This section considers the details of the implementation of a FEniCS-based solver for waveguide cutoff mode and dispersion curve problems, as described in the preceding section. A number of code snippets illustrate some of the finer points of the implementation.

34.2.1 Formulation

The code listing that follows shows the definitions of the function spaces used in the solution of the cutoff and dispersion problems considered here. As already discussed, the Nédélec basis functions of the first kind are used to approximate the transverse component of the electric field. This ensures the tangential continuity of the discrete transverse field (Jin, 2002). The axial component of the field is modelled using a set of Lagrange basis functions, with the integration domain (Ω) the waveguide cross section. The finite element mesh (generated using the DOLFIN `Rectangle` class) for the rectangular waveguide problems considered here is shown in Figure 34.2.

Python code

```
V_N = FunctionSpace(mesh, "Nedelec 1st kind H(curl)", transverse_order)
V_L = FunctionSpace(mesh, "Lagrange", axial_order)

combined_space = V_N * V_L

(N_i, L_i) = TestFunctions(combined_space)
(N_j, L_j) = TrialFunctions(combined_space)
```

In order to deal with material properties, the `Expression` class is subclassed and the `eval` method is overridden. This is illustrated in the next listing, which shows the implementation of the dielectric properties of a half-filled rectangular guide defined as follows

$$\epsilon_r(x, y) = \begin{cases} 4 & \text{if } y < 0.25\,, \\ 1 & \text{otherwise.} \end{cases} \tag{34.28}$$

This class is then instantiated for the relative permittivity (ϵ_r) and a constant expression is used for the relative permeability (or more specifically its inverse ($\frac{1}{\mu_r} = 1$)). The listing also shows the Expression class used for the square of the operating wavenumber (k_o^2), which is frequency dependent. Note that although it is set to zero, this value can be set for each frequency step as part of the dispersion analysis of a waveguide structure.

Python code

```
class HalfLoadedDielectric(Expression):
    def eval(self, values, x):
        if x[1] < 0.25:
            values[0] = 4.0
        else:
            values[0] = 1.0;

e_r = HalfLoadedDielectric()
one_over_u_r = Expression("1.0")

k_o_squared = Expression("value", {"value" : 0.0})
```

The testing and trial functions shown as well as the desired material properties can now be used to create the forms required for matrix assembly as specified in (34.16) through (34.19), and (34.23) through (34.27). The implementations of the forms are shown in the listing below, and the matrices of (34.14) and (34.22) can be assembled using the required combinations of these forms. It should be noted that the use of the `Expression` class for the representation of k_o^2 means that the forms need not be recompiled each time the operating frequency is changed. This is especially beneficial when the calculation of dispersion curves is considered since the same calculation is performed for a range of operating frequencies.

Python code

```
s_tt = one_over_u_r*dot(curl_t(N_i), curl_t(N_j))
t_tt = e_r*dot(N_i, N_j)

s_zz = one_over_u_r*dot(grad(L_i), grad(L_j))
t_zz = e_r*L_i*L_j

b_tt = one_over_u_r*dot(N_i, N_j)
b_tz = one_over_u_r*dot(N_i, grad(L_j))
b_zt = one_over_u_r*dot(grad(L_i), N_j)

a_tt = s_tt - k_o_squared*t_tt
b_zz = s_zz - k_o_squared*t_zz
```

From (34.2) it follows that the tangential component of the electric field must be zero on perfectly electrical conducting (PEC) surfaces (Smith, 1997). What this means in practice is that the degrees of freedom associated with both the Lagrange and Nédélec basis functions on the boundary must be set to zero. An implementation example for a PEC surface surrounding the entire computational domain is shown in the code listing below as the `ElectricWalls` class. This subdomain is then used to create a Dirichlet boundary condition that can be applied to the constructed matrices before solving the eigenvalue systems.

Python code

```
class ElectricWalls(SubDomain):
    def inside(self, x, on_boundary):
        return on_boundary
```

```
zero = Expression("0.0","0.0","0.0")
dirichlet_bc = DirichletBC(combined_space, zero, ElectricWalls())
```

The boundary condition given in (34.3) is a natural boundary condition for the problems and formulations considered and thus it is not necessary to explicitly enforce it (Pelosi et al., 1998). Such magnetic walls and the symmetry of a problem are often used to decrease the size of the computational domain although this does limit the solution obtained to even modes (Jin, 2002).

Once the required matrices have been assembled and the boundary conditions applied, the resultant eigenproblem can be solved. This can be done by saving the matrices and solving the problem externally, or by making use of the eigensolvers provided by SLEPc – which is discussed in more detail in Chapter 36 – through the FEniCS package.

34.2.2 Post-processing

After the eigenvalue system has been solved, an eigenpair can be post-processed to obtain various quantities of interest. For the cutoff wavenumber, this is a relatively straight-forward process and only involves simple operations on the eigenvalues of the system. For the calculation of dispersion curves and visualization of the resultant field components the process is slightly more complex.

Dispersion curves. For dispersion curves the computed value of the propagation constant ($\gamma = \alpha + j\beta$) is plotted as a function of the operating frequency (f_o). Since γ is a complex variable, a mapping is required to represent the data on a single two-dimensional graph. This is achieved by choosing the f_o-axis to represent the value $\gamma = 0$, effectively dividing the γf_o-plane into two regions. The region above the f_o-axis is used to represent the magnitude of the imaginary part of γ ($|\beta|$), whereas the magnitude of the real part ($|\alpha|$) falls in the lower region. A mode that propagates along the guide for a given frequency will thus lie in the upper half-plane of the plot, an evanescent mode will fall in the lower half-plane, and a complex mode will be represented by a data point above and below the f_o-axis. This procedure is followed in Pelosi et al. (1998) and allows for quick comparisons and validation of results.

Field visualization. In order to visualize the fields associated with a given solution, the basis functions need to be weighted with coefficients corresponding to the entries in an eigenvector obtained from one of the eigenvalue problems. In addition, the transverse or axial components of the field may need to be extracted. An example for plotting the transverse and axial components of the field is given in the code listing below. Here the variable `e` assigned to the function vector is one of the eigenvectors obtained by solving the eigenvalue problem. The `eval` method of the `transverse` and `axial` functions can also be called in order to evaluate the functions at a given spatial coordinate, allowing for further visualization or post-processing options.

Python code

```
f = Function(combined_space, e)

(transverse, axial) = f.split()

plot(transverse)
plot(axial)
```

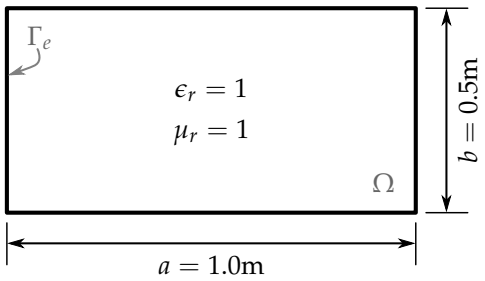


Figure 34.3: A diagram showing the cross section (Ω) and dimensions of a 1m $\times$ 0.5m hollow rectangular waveguide. The electric wall Γ_e, where the zero Dirichlet boundary condition of (34.2) is applied, is also shown.

34.3 Examples

The first of the examples considered is the canonical one of a hollow rectangular waveguide, which has been covered in a multitude of texts on the subject (Davidson, 2011; Jin, 2002; Pelosi et al., 1998; Pozar, 2005). Since the analytical solutions for this structure are known, it provides an excellent benchmark and is a typical starting point for the validation of a computational electromagnetic solver for solving waveguide problems.

The second and third examples are a partially filled rectangular guide and a shielded microstrip line on a dielectric substrate, respectively. In each case results are compared to published results from the literature for validation.

34.3.1 Hollow rectangular waveguide

Figure 34.3 shows the cross section of a hollow rectangular waveguide with dimensions $a = 1$m and $b = 0.5$m. The analytical expressions for the electric field components of a hollow rectangular guide with width a and height b are given by (Pozar, 2005)

$$E_x = \frac{n}{b} A_{mn} \cos\left(\frac{m\pi x}{a}\right) \sin\left(\frac{n\pi y}{b}\right), \tag{34.29}$$

$$E_y = -\frac{m}{a} A_{mn} \sin\left(\frac{m\pi x}{a}\right) \cos\left(\frac{n\pi y}{b}\right), \tag{34.30}$$

for the TE_{mn} (transverse electric) modes. These modes only have electric field components in the plane perpendicular to the direction of propagation (Pozar, 2005) – the waveguide cross section – and correspond with the transverse part (E_t) of the finite element solution. The subscripts mn are used to identify the modes, with m and n non-negative integers subject to the restriction that at least one of them must be nonzero.

The z-directed (axial) electric field corresponds to the TM_{mn} (transverse magnetic) modes and has the form (Pozar, 2005)

$$E_z = B_{mn} \sin\left(\frac{m\pi x}{a}\right) \sin\left(\frac{n\pi y}{b}\right). \tag{34.31}$$

Once again the subscript mn is used to identify the mode, but in this case neither m nor n may be zero. Such a TM mode has components of the magnetic field in the xy-plane, while the electric field has only an axial component. In (34.29), (34.30), and (34.31), A_{mn} and B_{mn} are constants for a given mode.

For a hollow rectangular guide, the propagation constant, γ, has the form

$$\gamma = \sqrt{k_c^2 - k_o^2}, \tag{34.32}$$

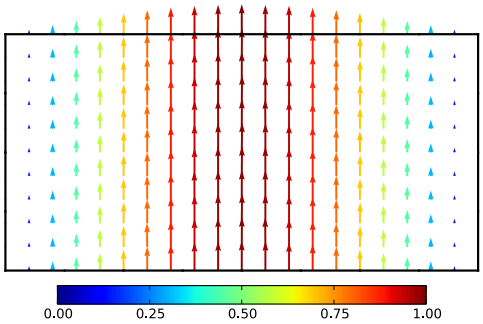


Figure 34.4: The calculated TE_{10} cutoff mode for the 1m $\times$ 0.5m hollow rectangular waveguide shown in Figure 34.3.

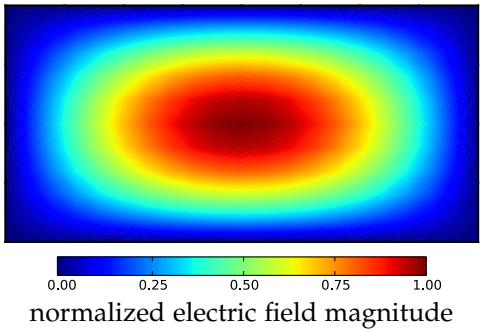


Figure 34.5: The calculated TM_{11} cutoff mode for the 1m $\times$ 0.5m hollow rectangular waveguide shown in Figure 34.3.

with k_o the operating wavenumber dependent on the operating frequency as in (34.4), and

$$k_c^2 = \left(\frac{m\pi}{a}\right)^2 + \left(\frac{n\pi}{b}\right)^2, \tag{34.33}$$

the analytical solution for the square of the cutoff wavenumber for both the TE_{mn} and TM_{mn} modes.

Cutoff analysis. Figure 34.4 and Figure 34.5 show the calculated TE_{10} and TM_{11} cutoff modes, respectively, for the hollow rectangular guide of Figure 34.3.

Table 34.1 gives a comparison of the calculated and analytical values for the square of the cutoff wavenumber of a number of modes for a hollow rectangular guide. As can be seen from the table, there is excellent agreement between the values.

Mode	Analytical [m^{-2}]	Calculated [m^{-2}]	Relative Error
TE_{10}	9.8696	9.8696	1.4452e-06
TE_{01}	39.4784	39.4784	2.1855e-05
TE_{20}	39.4784	39.4784	2.1894e-05
TM_{11}	49.3480	49.4048	1.1514e-03
TM_{21}	78.9568	79.2197	3.3295e-03
TM_{31}	128.3049	129.3059	7.8018e-03

Table 34.1: Comparison of analytical and calculated cutoff wavenumber squared (k_c^2) for various TE and TM modes of a 1m $\times$ 0.5m hollow rectangular waveguide.

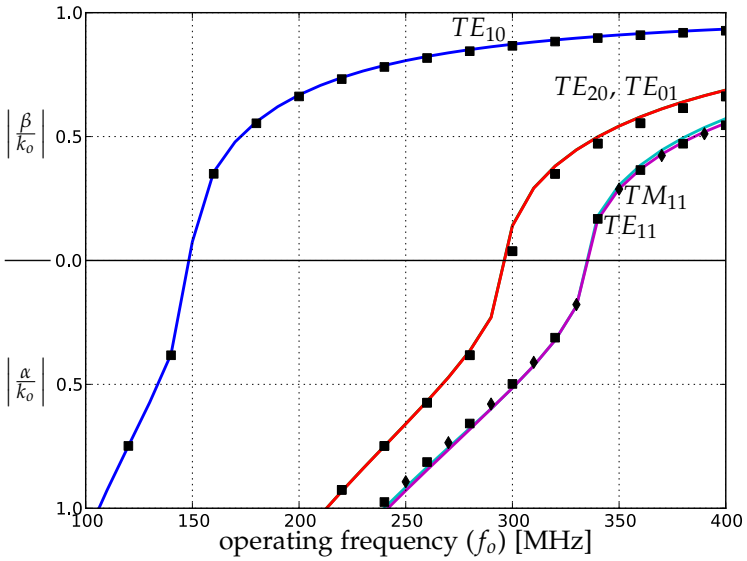


Figure 34.6: Dispersion curves for the first 5 modes of a 1m $\times$ 0.5m hollow rectangular waveguide of Figure 34.3. Markers are used to indicate the analytical results with ■ and ♦ indicating TE and TM modes, respectively. Note that the analytical TE_{01} and TE_{20} form a degenerate pair, as do the TE_{11} and TM_{11} modes.

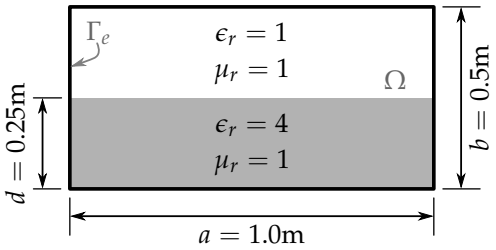


Figure 34.7: A diagram showing the cross section (Ω) and dimensions of a 1m $\times$ 0.5m half-loaded rectangular waveguide. The lower half of the guide is filled with an $\epsilon_r = 4$ dielectric material. The electric wall Γ_e, where the zero Dirichlet boundary condition of (34.2) is applied, is also shown.

Dispersion analysis. When considering the calculation of the dispersion curves for the hollow rectangular waveguide, the mixed formulation as discussed in 34.1.2 is used. The calculated dispersion curves for the first 5 modes of the hollow rectangular guide are shown in Figure 34.6 along with the analytical results. For the rectangular guide, a number of modes are degenerate (see Davidson, 2011, Chapter 10) with the same dispersion and cutoff properties as predicted by (34.32) and (34.33). (As an example consider the TE_{01} and TM_{20} modes that will be degenerate for any rectangular waveguide that is twice as wide as it is high, as is the case here.) There is excellent agreement between the analytical and computed results.

34.3.2 Half-loaded rectangular waveguide

In some cases, a hollow rectangular guide may not be the ideal structure to use due to, for example, limitations on its dimensions. If the guide is filled with a dielectric material with a relative permittivity $\epsilon_r > 1$, the cutoff frequency of the dominant mode will be lowered. Consequently a loaded waveguide will be more compact than a hollow guide for the same dominant mode frequency. Furthermore, in many practical applications, such as impedance matching or phase shifting sections, a waveguide that is only partially loaded is used (Pozar, 2005).

Figure 34.7 shows the cross section of such a guide. The guide considered here has the same dimensions as the hollow rectangular waveguide used in the previous section, but its lower half is filled with an $\epsilon_r = 4$ dielectric material.

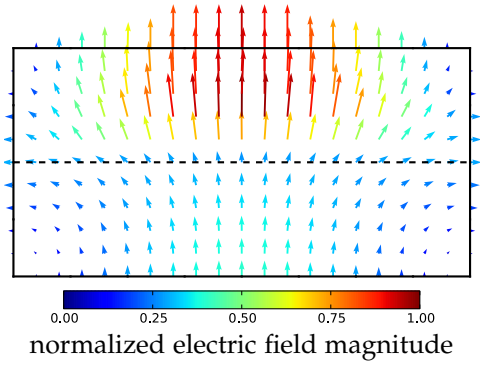


Figure 34.8: The first calculated cutoff mode of a 1m $\times$ 0.5m half-filled rectangular waveguide as shown in Figure 34.7. The dielectric surface is shown as a dashed horizontal line.

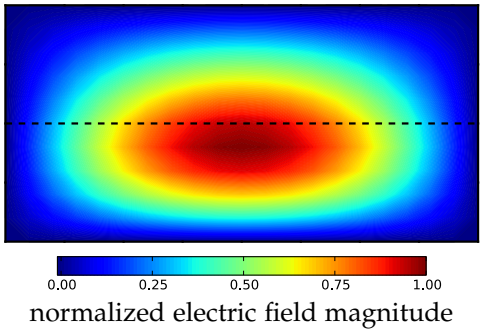


Figure 34.9: The fourth calculated cutoff mode of a 1m $\times$ 0.5m half-filled rectangular waveguide as shown in Figure 34.7. The dielectric surface is shown as a dashed horizontal line.

Cutoff analysis. Figure 34.8 and Figure 34.9 show the first and fourth cutoff modes of the half-loaded guide (shown in Figure 34.7), respectively. Note the concentration of the transverse electric field in the hollow part of the guide. This is due to the fact that the displacement flux, $D = \epsilon E$, must be normally continuous at the dielectric interface (Pozar, 2005; Smith, 1997).

Dispersion analysis. The dispersion curves for the first 4 modes of the half-loaded waveguide are shown in Figure 34.10 with results for the same modes from Jin (2002) provided as reference. Here it can be seen that the cutoff frequency of the dominant mode has decreased and there is no longer the same degeneracy in the modes when compared to the hollow guide of the same dimensions.

34.3.3 Shielded microstrip

Microstrip line is a very popular type of planar transmission line, primarily due to the fact that it can be constructed using photolithographic processes and integrates easily with other microwave components (Pozar, 2005). Such a structure typically consists of a thin conducting strip on a dielectric substrate above a ground plane. In addition, the strip may be shielded by enclosing it in a PEC box to reduce electromagnetic interference. A cross section of a lossless shielded microstrip line is shown in Figure 34.11 with the thickness of the strip, t, exaggerated for clarity. The dimensions used to obtain the results discussed here, are the same as those in Pelosi et al. (1998), and are indicated in the figure.

Since the shielded microstrip structure consists of two conductors, it supports a dominant transverse electromagnetic (TEM) wave that has no axial component of the electric or magnetic field (Pozar, 2005). Such a mode has a cutoff wavenumber of zero and thus propagates for all frequencies (Jin, 2002; Pelosi et al., 1998). Although it can be performed, the cutoff analysis of this structure is not considered here explicitly and only the dispersion analysis is performed. The cutoff wavenumbers for the higher order modes (which are hybrid TE-TM modes (Pozar, 2005)) can however be determined from the dispersion curves by the intersection of a curve with the f_o-axis.

Figure 34.10: Dispersion curves for the first 4 modes of a 1m $\times$ 0.5m half-filled rectangular waveguide as shown in Figure 34.7. Reference values for the first 4 modes from Jin (2002) are shown as ■.

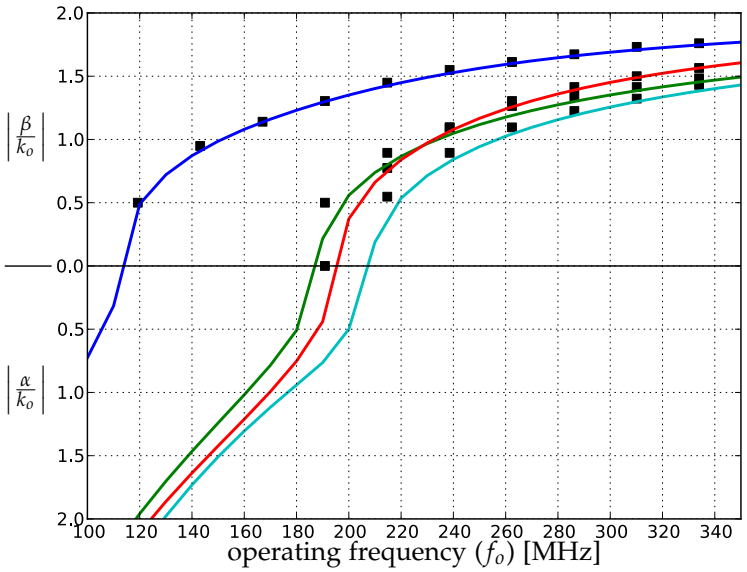
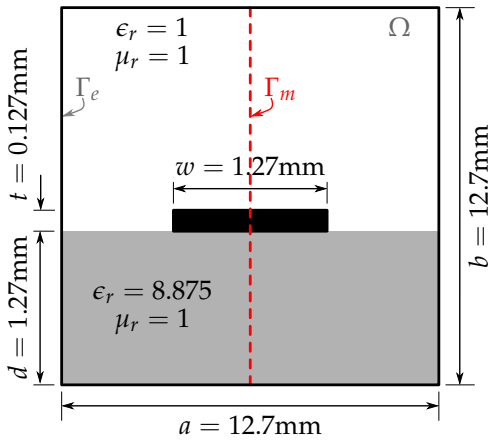


Figure 34.11: A diagram showing the cross section and dimensions of a shielded microstrip line. The microstrip is etched on a dielectric material with a relative permittivity of $\epsilon_r = 8.875$. The plane of symmetry is indicated by a dashed line and is modelled as a magnetic wall (Γ_m) in order to reduce the size of the computational domain. The electric wall (Γ_e) is also shown.



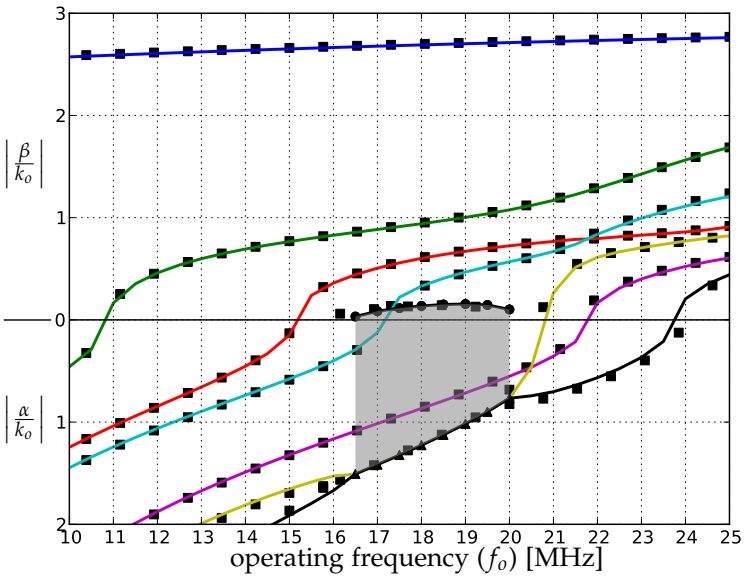


Figure 34.12: Dispersion curves for the first 7 even modes of shielded microstrip line of Figure 34.11 using a magnetic wall to enforce symmetry. Reference values from Pelosi et al. (1998) are shown as ■. The presence of complex mode pairs are indicated by ▲ and ● and highlighted in grey.

Dispersion analysis. The dispersion analysis presented in Pelosi et al. (1998) is repeated here for validation, with the resultant curves shown in Figure 34.12. As is the case with the half-loaded guide, the results calculated with FEniCS agree well with previously published results. In this figure, it is shown that for certain parts of the frequency range of interest, mode six and mode seven have complex propagation constants. Since the matrices in the eigenvalue problem are real valued, the complex eigenvalues – and thus the propagation constants – must occur in complex conjugate pairs as is the case here and reported earlier in Huang and Itoh (1988). It should be noted that for lossy materials (not considered here), complex modes are expected but do not necessarily occur in conjugate pairs (Pelosi et al., 1998).

34.4 Conclusions

In this chapter, the solutions of cutoff and dispersion problems associated with electromagnetic waveguiding structures have been implemented and the results analyzed. In all cases, the results obtained agree well with previously published or analytical results.

It should be noted that although the examples are limited to two-dimensional resonant problems, the formulations presented here can be extended to include three-dimensional eigenvalue problems (where resonant cavities are considered) as well as driven problems in both two and three dimensions. Details can be found in Jin (2002) and Pelosi et al. (1998).

This chapter has also illustrated the ease with which complex formulations can be implemented and how quickly solutions can be obtained. This is largely due to the almost one-to-one correspondence between the expressions at a formulation level and the high-level FEniCS code that is used to implement a particular solution. Even in cases where the required functionality is limited or missing, the use of FEniCS in conjunction with external packages greatly reduces development time.

35 Block preconditioning of systems of PDEs

By Kent-Andre Mardal and Joachim Berdal Haga

In this chapter, we describe the implementation of block preconditioned Krylov solvers for systems of partial differential equations (PDEs) using CBC.Block and the Python interfaces of DOLFIN and Trilinos. We start by reviewing the abstract theory of constructing preconditioners by considering the differential operators as mappings in properly chosen Sobolev spaces, before giving a short overview of CBC.Block. We then present several examples, namely the Poisson problem, the Stokes problem, the time-dependent Stokes problem and finally a mixed formulation of the Hodge Laplacian.

35.1 Abstract framework for constructing preconditioners

This presentation of preconditioning is largely taken from the review paper (Mardal and Winther, 2011), where a more comprehensive mathematical presentation is given. Consider the following abstract formulation of a linear PDE problem: find u in a Hilbert space H such that:

$$\mathcal{A}u = f, \tag{35.1}$$

where $f \in H'$ and H' is the dual space of H. We will assume that the PDE problem is well-posed; that is, $\mathcal{A} : H \to H'$ is a bounded invertible operator in the sense that,

$$\|\mathcal{A}\|_{\mathcal{L}(H,H')} \leq C \quad \text{and} \quad \|\mathcal{A}^{-1}\|_{\mathcal{L}(H',H)} \leq C. \tag{35.2}$$

The reader should notice that this operator is bounded only when viewed as an operator from H to H'. The spectrum of the operator is unbounded and discretizations of the operator will typically have condition numbers that increase in negative powers of h, where h is the characteristic cell size, as the mesh is refined. The remedy for the unbounded spectrum is to introduce a preconditioner. Let the preconditioner B be an operator mapping H' to H such that

$$\|\mathcal{B}\|_{\mathcal{L}(H',H)} \leq C \quad \text{and} \quad \|\mathcal{B}^{-1}\|_{\mathcal{L}(H,H')} \leq C. \tag{35.3}$$

Then $\mathcal{B}\mathcal{A} : H \to H$ and

$$\|\mathcal{B}\mathcal{A}\|_{\mathcal{L}(H,H)} \leq C^2 \quad \text{and} \quad \|(\mathcal{B}\mathcal{A})^{-1}\|_{\mathcal{L}(H,H)} \leq C^2. \tag{35.4}$$

Hence, the spectrum and therefore the condition number of the preconditioned operator is bounded:

$$\kappa(\mathcal{B}\mathcal{A}) = \|\mathcal{B}\mathcal{A}\|_{\mathcal{L}(H,H)}\|(\mathcal{B}\mathcal{A})^{-1}\|_{\mathcal{L}(H,H)} \leq C^4. \tag{35.5}$$

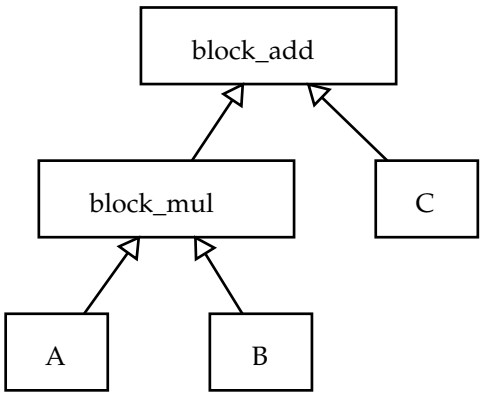


Figure 35.1: Expression tree for the composed matrix $M = AB + C$.

One example of such a preconditioner is the Riesz operator $\mathcal{R}$; that is, the identity mapping between H' and H. In this case

$$\|\mathcal{R}\|_{\mathcal{L}(H',H)} = 1 \quad \text{and} \quad \|\mathcal{R}^{-1}\|_{\mathcal{L}(H,H')} = 1. \tag{35.6}$$

In fact, in most of our examples the preconditioners are approximate Riesz mappings.

Given that the discretized operators $\mathcal{A}_h$ and $\mathcal{B}_h$ are stable; that is,

$$\|\mathcal{A}_h\|_{\mathcal{L}(H,H')} \leq C,\ \|\mathcal{A}_h^{-1}\|_{\mathcal{L}(H',H)} \leq C,\ \|\mathcal{B}_h\|_{\mathcal{L}(H',H)} \leq C,\ \|\mathcal{B}_h^{-1}\|_{\mathcal{L}(H,H')} \leq C, \tag{35.7}$$

then the condition number of the discrete preconditioned operator, $\kappa(\mathcal{B}_h\mathcal{A}_h)$, will be bounded by C^4 independently of h. Furthermore, the number of iterations required by a Krylov solver to reach a certain convergence criterion can typically be bounded by the condition number. Hence, when the condition number of the discrete problem is bounded independent of h, the Krylov solvers will have a convergence rate that is independent of h. If $\mathcal{B}_h$ is similar to $\mathcal{A}_h$ in terms of storage and evaluation, then the solution algorithm is *order-optimal*. We remark that it is crucial that $\mathcal{A}_h$ is a stable operator and we will illustrate what happens for unstable operators in the example concerning Stokes problem. Finally, we will see that $\mathcal{B}_h$ often can be constructed using multigrid techniques. These multigrid preconditioners will be spectrally equivalent with the Riesz mappings.

35.2 Overview of CBC.Block

CBC.Block makes it possible to write matrix operations in mathematical notation, such as $M = AB + C$, where A, B and C are matrices or operators. The algebraic operations are not performed explicitly; instead the operators are stored in a graph as shown in Figure 35.1. When M is called upon to operate on a vector, as in $y = Mx$, the individual operations are performed in the right order on the vector: $u = Bx$, $v = Au$, $w = Cx$, $y = v + w$. Since the matrix product or sum is not created explicitly, the individual operators do not need to have an explicit matrix representation, but may be a DOLFIN matrix, a preconditioner such as ML, or even an inner iterative solver. To enable the construction of this graph CBC.Block injects the methods `__mul__`, `__add__`, and `__sub__` into the `Matrix` and `Vector` classes in DOLFIN. The module also implements block partitioned matrices and vectors. These are pure python objects, and do not use the block matrix in DOLFIN.

When the explicit matrix product is required, such as for input to the ML preconditioner, a method `collapse` is provided that performs the calculations using PyTrilinos. This method requires that all components are actual matrices, not general operators.

The module also provides services to set Dirichlet boundary conditions and perform other transformations of the system, and a range of iterative solvers and preconditioners.

35.3 Numerical examples

In all examples the mesh will be refinements of the unit square or unit cube. The code examples presented in this chapter differ slightly from the source code, in the sense that import statements, safety checks, command-line arguments, definitions of `Functions` and `Subdomains` are often removed to shorten the presentation.

35.3.1 The Poisson problem with homogeneous Neumann conditions

The Poisson equation with Neumann conditions reads: find u such that

$$-\operatorname{div}\operatorname{grad} u = f \text{ in } \Omega, \tag{35.8}$$

$$\frac{\partial u}{\partial n} = g \text{ on } \partial\Omega. \tag{35.9}$$

The corresponding variational problem is: Find $u \in H^1 \cap L_0^2$ such that

$$\int_\Omega \operatorname{grad} u \cdot \operatorname{grad} v \,\mathrm{d}x = \int_\Omega fv \,\mathrm{d}x + \int_{\partial\Omega} gv \,\mathrm{d}s, \quad \forall\, v \in H^1 \cap L_0^2.$$

Let the linear operator $\mathcal{A}$ be defined in terms of the bilinear form,

$$(\mathcal{A}u, v) = \int_\Omega \operatorname{grad} u \cdot \operatorname{grad} v \,\mathrm{d}x.$$

It is well-known that $\mathcal{A}$ is a bounded invertible operator from $H^1 \cap L_0^2$ into its dual space. Furthermore, it is well-known that one can construct multigrid preconditioners for this operator such that the preconditioner is spectrally equivalent with the inverse of $\mathcal{A}$, independent of the characteristic size of the cells in the mesh (Bramble, 1993; Hackbusch, 1994; Trottenberg et al., 2001).

In this example, we use a multigrid preconditioner based on the algebraic multigrid package ML contained in PyTrilinos. Furthermore, we will estimate the eigenvalues of the preconditioned system. We use continuous piecewise linear elements and compute the condition number of the preconditioned system and corresponding number of iteration required for convergence using the conjugate gradient method for various uniform refinements of the unit square.

First of all, the ML preconditioner is constructed as follows,

Python code

```
class ML(block_base):
    def __init__(self, A, pdes=1):
        # create the ML preconditioner
        MLList = {
            "smoother: type"            : "ML symmetric Gauss-Seidel" ,
            "aggregation: type"         : "Uncoupled" ,
            "ML validate parameter list": True,
            }
        self.A = A # Prevent matrix being deleted
        self.ml_prec = MultiLevelPreconditioner(A.down_cast().mat(), 0)
        self.ml_prec.SetParameterList(MLList)
        self.ml_agg = self.ml_prec.GetML_Aggregate()
        self.ml_prec.ComputePreconditioner()

    def matvec(self, b):
        x = self.A.create_vec()
        self.ml_prec.ApplyInverse(b.down_cast().vec(), x.down_cast().vec())
        return x
```

The linear algebra backends uBLAS, PETSc and Trilinos all have a wide range of Krylov solvers. Here, we implement these solvers in Python because we need to store intermediate variables and used them to compute an estimate of the condition number. The following code shows the implementation of the conjugate gradient method using the Python linear algebra interface in DOLFIN:

Python code

```
def precondconjgrad(B, A, x, b, tolerance, maxiter, progress, relativeconv=False):

    r = b - A*x
    z = B*r
    d = z
    rz = inner(r,z)

    iter = 0
    alphas = []
    betas = []
    residuals = [sqrt(rz)]

    if relativeconv:
        tolerance *= residuals[0]

    while residuals[-1] > tolerance and iter <= maxiter:
        z = A*d
        dz = inner(d,z)
        alpha = rz/dz
        x += alpha*d
        r -= alpha*z
        z = B*r
        rz_prev = rz
        rz = inner(r,z)
        beta = rz/rz_prev
        d = z + beta*d

        iter += 1
        progress += 1
        alphas.append(alpha)
        betas.append(beta)
        residuals.append(sqrt(rz))

    return x, residuals, alphas, betas
```

The intermediate variables called alphas and betas can then be used to estimate the condition number of the preconditioned matrix as follows; see Saad (2003). Notice that since the preconditioned conjugate gradient method converges quite fast when using algebraic multigrid (AMG) as a preconditioner, there will be only a small number of alphas and betas. Therefore we use the dense linear algebra tools in NumPy to compute the eigenvalue estimates.

Python code

```
def eigenvalue_estimates(self):
    # eigenvalues estimates in terms of alphas and betas

    import numpy

    n = len(self.alphas)
    M = numpy.zeros([n,n])
    M[0,0] = 1/self.alphas[0]
    for k in range(1, n):
        M[k,k] = 1/self.alphas[k] + self.betas[k-1]/self.alphas[k-1]
        M[k,k-1] = numpy.sqrt(self.betas[k-1])/self.alphas[k-1]
```

```
            M[k-1,k] = M[k,k-1]
        e,v = numpy.linalg.eig(M)
        e.sort()
        return e
```

The following code shows the implementation of a Poisson problem solver, using the above mentioned ML preconditioner and conjugate gradient algorithm. We remark here that it is essential for the convergence of the method that both the start vector and the right-hand side are both in L^2_0. For this reason we subtract the mean value from the right-hand side. The start vector is zero and does therefore have mean value zero.

Python code

```
# Create mesh and finite element
mesh = UnitSquare(N,N)
V = FunctionSpace(mesh, "Lagrange", 1)

# Define variational problem
v = TestFunction(V)
u = TrialFunction(V)
f = Source()
g = Flux()

a = dot(grad(v), grad(u))*dx
L = v*f*dx + v*g*ds

# Assemble matrix and vector
A, b = assemble_system(a,L)

# Remove constant from right-hand side
c = b.array()
c -= sum(c)/len(c)
b[:] = c

# Create preconditioner
B = ML(A)
Ainv = ConjGrad(A, precond=B, tolerance=1e-8)

x = Ainv*b

e = Ainv.eigenvalue_estimates()

print "N=%d iter=%d K=%.3g" % (N, Ainv.iterations, e[-1]/e[0])
```

In Table 35.1 we list the number of iterations for convergence and the estimated condition number of the preconditioned system based on the code shown above. We test different refinements of the unit square and continuous piecewise linear elements, CG_1. The source function is $f = 500\exp(-((x-0.5)^2+(y-0.5)^2)/0.02)$ and the boundary condition is $g = 25\sin(5\pi y)$ for $x = 0$ and zero elsewhere, see also the source code `poisson_neumann.py`.

35.3.2 The Stokes problem

Our next example is the Stokes problem,

$$-\operatorname{div}\operatorname{grad} u - \operatorname{grad} p = f \quad \text{in } \Omega, \tag{35.10}$$

$$\operatorname{div} u = 0 \quad \text{in } \Omega, \tag{35.11}$$

$$u = g \quad \text{on } \partial\Omega. \tag{35.12}$$

h	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}
κ	1.57	1.26	2.09	1.49	1.20
#iterations	8	8	10	9	7

Table 35.1: The estimated condition number κ and the number of iterations for convergence with respect to various uniform mesh refinements for the Poisson problem with Neumann conditions solved with the CG_1 method.

The variational form is:
Find $u, p \in H^1_g \times L^2_0$ such that

$$\int_\Omega \operatorname{grad} u : \operatorname{grad} v \,\mathrm{d}x + \int_\Omega \operatorname{div} u \, q \,\mathrm{d}x + \int_\Omega \operatorname{div} v \, p \,\mathrm{d}x = \int_\Omega f \, v \,\mathrm{d}x, \quad \forall\, v, q \in H^1_0 \times L^2_0.$$

Let the linear operator $\mathcal{A}$ be defined as

$$\mathcal{A} = \begin{pmatrix} A & B^* \\ B & 0 \end{pmatrix}.$$

where

$$(Au, v) = \int_\Omega \operatorname{grad} u : \operatorname{grad} v \,\mathrm{d}x, \tag{35.13}$$

$$(Bu, q) = \int_\Omega \operatorname{div} u \, q \,\mathrm{d}x, \tag{35.14}$$

and B^* is the adjoint of B. Then it is well-known that $\mathcal{A}$ is a bounded operator from $H^1_g \times L^2_0$ to its dual $H^{-1}_g \times L^2_0$, see for example Brezzi (1974); Brezzi and Fortin (1991). Therefore, we construct a preconditioner, $\mathcal{B} : H^{-1}_g \times L^2_0 \to H^1_g \times L^2_0$ defined as

$$\mathcal{B} = \begin{pmatrix} K^{-1} & 0 \\ 0 & L^{-1} \end{pmatrix}.$$

where

$$(Ku, v) = \int_\Omega \operatorname{grad} u : \operatorname{grad} v \,\mathrm{d}x, \tag{35.15}$$

$$(Lp, q) = \int_\Omega p \, q \,\mathrm{d}x. \tag{35.16}$$

We refer to Mardal and Winther (2011) for a mathematical explanation of the derivation of such preconditioners. Notice that the operator $\mathcal{B}$ is positive in contrast to $\mathcal{A}$. Hence, the preconditioned operator $\mathcal{B}\mathcal{A}$ will be indefinite. For both K and L, we use the AMG preconditioner provided by ML/Trilinos as described in the previous example (A simple Jacobi preconditioner would be sufficient for L). For symmetric indefinite problems the *Minimum Residual Method* is the fastest method. Preconditioners of this form has been studied by many (Elman et al., 2005; Rusten and Winther, 1992; Silvester and Wathen, 1993, 1994).

In Table 35.2 we present the number of iterations needed for convergence and estimates on the condition number κ with respect to different discretization methods and different characteristic cell sizes h. The problem we are solving is the so-called lid driven cavity problem; that is, $f = 0$ and $g = (1, 0)$ for $y = 1$ and zero elsewhere. We use different mixed methods, namely the

method	h	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}
$CG_2 - CG_1$	iterations	52	57	62	64	67
$CG_2 - CG_1$	κ	13.6	13.6	13.6	13.6	13.6
$CG_2 - DG_0$	iterations	43	48	55	59	62
$CG_2 - DG_0$	κ	8.5	9.2	9.7	10.3	10.7
$CG_1 - CG_1$	iterations	200+	200+	200+	200+	200+
$CG_1 - CG_1$	κ	696	828	672	651	630
$CG_1 - CG_1$-stab	iterations	41	40	40	39	39
$CG_1 - CG_1$-stab	κ	12.5	12.6	12.7	12.7	12.7

Table 35.2: Estimated condition number κ and number of iterations for convergence with respect to mesh refinements. The methods $CG_2 - CG_1$, $CG_2 - DG_0$, and $CG_1 - CG_1$-stab are stable, while $CG_1 - CG_1$ is not.

$CG_2 - CG_1$, $CG_2 - DG_0$, $CG_1 - CG_1$, and $CG_1 - CG_1$ stabilized. The iteration is stopped when $(\mathcal{B}_h r_k, r_k)/(\mathcal{B}_h r_0, r_0) \leq 10^{-8}$, where r_k is the residual at iteration k. The condition numbers, κ, were estimated using the conjugate gradient method on the normal equation. This condition number will always be less than the real condition number and is probably too low for the last columns for the $CG_1 - CG_1$ method without stabilization. Notice that for the stable methods that satisfy the LBB condition (see also Chapter 36); that is, $CG_2 - CG_1$ and $CG_2 - DG_0$, the number of iterations and the condition number seems to be bounded independently of h. For the unstable $CG_1 - CG_1$ method, the number of iterations and the condition number increases as h decreases, but is here stopped. However, for the stabilized method, $CG_1 - CG_1$-stab, where the pressure is stabilized by

$$\int_\Omega \operatorname{div} u\, q - \alpha h^2 \operatorname{grad} p \cdot \operatorname{grad} q \,\mathrm{d}x,$$

with $\alpha = 0.01$, the number of iterations and the condition number appear to be bounded.

We will now describe the code in detail. In this case, the preconditioner consists of two preconditioners. The following shows how to implement this block preconditioner based on the ML preconditioner defined in the previous example.

Python code

```
mesh = UnitSquare(N,N)
def CG(n):
    return ("DG",0) if n==0 else ("CG",n)

V = VectorFunctionSpace(mesh, *CG(vorder))
Q = FunctionSpace(mesh, *CG(porder))

f = Constant((0,0))
g = Constant(0)
alpha = Constant(alpha)
h = CellSize(mesh)

v,u = TestFunction(V), TrialFunction(V)
q,p = TestFunction(Q), TrialFunction(Q)

A  = assemble(inner(grad(v), grad(u))*dx)
B  = assemble(div(v)*p*dx)
C  = assemble(div(u)*q*dx)
D  = assemble(-alpha*h*h*dot(grad(p), grad(q))*dx)
M1 = assemble(p*q*dx)
b0 = assemble(inner(v, f)*dx)
```

```
b1 = assemble(q*g*dx)

AA = block_mat([[A, B],
                [C, D]])
bc = block_bc([DirichletBC(V, BoundaryFunction(), Boundary()), None])
b = block_vec([b0, b1])
bc.apply(AA, b)

BB = block_mat([[ML(A),  0],
                [0, ML(M1)]])


AAinv = MinRes(AA, precond=BB, tolerance=1e-8)
x = AAinv * b

x.randomize()

AAi = CGN(AA, precond=BB, initial_guess=x, tolerance=1e-8, maxiter=1000)
AAi * b

e = AAi.eigenvalue_estimates()

print "N=%d iter=%d K=%.3g" % (N, AAinv.iterations, sqrt(e[-1]/e[0]))
```

We refer to `stokes.py` for the complete code.

35.3.3 *The time-dependent Stokes problem*

Our next example is the time-dependent Stokes problem,

$$u - k \operatorname{div} \operatorname{grad} u - \operatorname{grad} p = f \quad \text{in } \Omega, \tag{35.17}$$

$$\operatorname{div} u = 0 \quad \text{in } \Omega, \tag{35.18}$$

$$u = 0 \quad \text{on } \partial\Omega. \tag{35.19}$$

Here k is the time stepping parameter.

The variational form is:
Find $u, p \in H_0^1 \times L_0^2$ such that

$$\int_\Omega u \cdot v \,\mathrm{d}x + k \int_\Omega \operatorname{grad} u : \operatorname{grad} v \,\mathrm{d}x + \int_\Omega \operatorname{div} u \, q \,\mathrm{d}x + \int_\Omega \operatorname{div} v \, p \,\mathrm{d}x = \int_\Omega f \, v \,\mathrm{d}x, \quad \forall v, q \in H_0^1 \times L_0^2.$$

Let

$$\mathcal{A} = \begin{pmatrix} A & B^* \\ B & 0 \end{pmatrix}.$$

where

$$(Au, v) = \int_\Omega u \cdot v \,\mathrm{d}x + k \int_\Omega \operatorname{grad} u : \operatorname{grad} v \,\mathrm{d}x, \tag{35.20}$$

$$(Bu, q) = \int_\Omega \operatorname{div} u \, q \,\mathrm{d}x, \tag{35.21}$$

This operator changes character as k varies. For $k = 1$ the problem behaves like Stokes problem, with a non-harmful low order term. However as k approaches zero the problems change to the mixed

$k\backslash h$	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}
1.0	13.6	13.6	13.6	13.7	13.7
0.1	13.4	13.5	13.6	13.6	13.6
0.01	12.8	13.2	13.4	13.5	13.6
0.001	11.0	12.3	13.0	12.3	13.5

Table 35.3: The convergence with respect to k and mesh refinements for the time-dependent Stokes problem when using the $\mathrm{CG}_2 - \mathrm{CG}_1$ method.

formulation of a Poisson equation; that is,

$$u - \operatorname{grad} p = f, \quad \text{in } \Omega, \tag{35.22}$$

$$\operatorname{div} u = 0, \quad \text{in } \Omega. \tag{35.23}$$

This problem is not a well-defined operator from $H_0^1 \times L_0^2$ into its dual. Instead, it is a mapping from $H(\operatorname{div}) \times L_0^2$ to its dual. However, as pointed out in Mardal and Winther (2004) this operator can also be seen as an operator $L^2 \times H^1$ to its dual. In fact, in Mardal et al. (2002); Mardal and Winther (2004) it was shown that A is a bounded operator from $L^2 \cap k^{1/2} H_0^1 \times H^1 \cap L_0^2 + k^{-1/2} L_0^2$ to its dual space with a bounded inverse. Furthermore, the bounds are uniform in k. Therefore, we construct a preconditioner B, such that

$$\mathcal{B} : L^2 \cap k^{1/2} H_0^1 \times H^1 \cap L_0^2 + k^{-1/2} L_0^2 \to L^2 + k^{-1/2} H^{-1} \times H^{-1} \cap L_0^2 + k^{1/2} L_0^2.$$

Such a $\mathcal{B}$ can be defined as

$$\mathcal{B} = \begin{pmatrix} K^{-1} & 0 \\ 0 & L^{-1} + M^{-1} \end{pmatrix}.$$

where

$$(Ku, v) = \int_\Omega u \cdot v + k \operatorname{grad} u : \operatorname{grad} v \, \mathrm{d}x, \tag{35.24}$$

$$(Lp, q) = \int_\Omega k^{-1} pq \, \mathrm{d}x, \tag{35.25}$$

$$(Mp, q) = \int_\Omega \operatorname{grad} p \cdot \operatorname{grad} q \, \mathrm{d}x. \tag{35.26}$$

Again we refer to Mardal and Winther (2011) and references therein, for an overview and more comprehensive mathematical derivation of the construction of such preconditioners. Preconditioners of this form has been studied by many; see for example Cahouet and Chabard (1988); Elman et al. (2005); Mardal and Winther (2004, 2011); Turek (1999).

Creating the preconditioner in this example is completely analogous to the Stokes example except that we need three matrices based on three bilinear forms:

Python code

```
# function spaces, trial and test functions, boundary conditions etc. are previously defined

A = assemble((dot(u,v) + k*inner(grad(u),grad(v)))*dx)
B = assemble(div(v)*p*dx)
C = assemble(div(u)*q*dx)
b = assemble(dot(f, v)*dx)

AA = block_mat([[A, B],
```

```
                [C, 0]])
bb = block_vec([b, 0])

M = assemble(kinv*p*q*dx)
L = assemble(dot(grad(p),grad(q))*dx)

prec = block_mat([[ML(A),      0     ],
                  [0,     ML(L)+ML(M)]])
```

In Table 35.3, we show the condition number for the time-dependent Stokes problem discretized with the $CG_2 - CG_1$ method for various uniform mesh refinements and values of k. We have the same boundary conditions as for the Stokes problem; that is, $f = 0$ and $g = (1, 0)$ for $y = 1$ and zero elsewhere. Clearly, the condition number appears to be bounded by ≈ 14, although the asymptotic limit is not reached for small k on these coarse meshes. The complete code can be found in `timestokes.py`

35.3.4 *Mixed form of the Hodge Laplacian*

The final example is a mixed formulation of the Hodge Laplacian,

$$\operatorname{rot}\operatorname{curl} u - \operatorname{grad} p = f \quad \text{in } \Omega, \tag{35.27}$$
$$\operatorname{div} u - p = 0 \quad \text{in } \Omega, \tag{35.28}$$
$$u \times n = 0 \quad \text{on } \partial\Omega, \tag{35.29}$$
$$p = 0 \quad \text{on } \partial\Omega. \tag{35.30}$$

The variational form is:
Find $u, p \in H_0(\operatorname{curl}) \times H_0^1$ such that

$$\int_\Omega \operatorname{curl} u \cdot \operatorname{curl} v \,\mathrm{d}x - \int_\Omega \operatorname{grad} p \, v \,\mathrm{d}x = \int_\Omega f v \,\mathrm{d}x \quad \forall\, v \in H_0(\operatorname{curl}), \tag{35.31}$$
$$\int_\Omega u \operatorname{grad} q \,\mathrm{d}x - \int_\Omega pq \,\mathrm{d}x = 0 \quad \forall\, q \in H_0^1. \tag{35.32}$$

Hence, it is natural to consider a preconditioner for $H(\operatorname{curl})$ problems (in addition to H^1 preconditioners). Such preconditioners have been considered by many (Arnold et al., 1997, 2000; Hiptmair, 1997, 1999). One important observation in these papers is that point-wise smoothers are not appropriate for geometric multigrid methods. Furthermore, for algebraic multigrid methods, extra care has to be taken for the aggregation step (Gee et al., 2006; Hu et al., 2006).

Let

$$\mathcal{A} = \begin{pmatrix} A & B^* \\ B & -C \end{pmatrix},$$

where,

$$(Au, v) = \int_\Omega \operatorname{curl} u \cdot \operatorname{curl} v \,\mathrm{d}x, \tag{35.33}$$
$$(Bp, v) = -\int_\Omega \operatorname{grad} p \, v \,\mathrm{d}x, \tag{35.34}$$
$$(Cp, q) = -\int_\Omega p \, q \,\mathrm{d}x. \tag{35.35}$$

Then $\mathcal{A} : H_0(\operatorname{curl}) \times H_0^1 \to H^{-1}(\operatorname{curl}) \times H^{-1}$, where $H^{-1}(\operatorname{curl})$ is the dual of $H_0(\operatorname{curl})$.

However, if we for the moment forget about the boundary conditions, we can obtain the Laplacian form by eliminating p from (35.27)-(35.28); that is,

$$\text{rot}\,\text{curl}\,u - \text{div}\,\text{grad}\,u = f.$$

Hence, the problem is elliptic in nature and modulo boundary conditions, $\mathcal{A} : H^1 \times L^2 \to H^{-1} \times L^2$.

To avoid constructing a $H(\text{curl})$ preconditioner we will employ the observation that this is a vector Laplacian. Let the discrete operator be

$$\mathcal{A} = \begin{pmatrix} A & B^* \\ B & -C \end{pmatrix},$$

where we assume that the discrete system has been obtained by using a stable finite element method. We eliminate the p to obtain the matrix

$$K = A + B^* C^{-1} B,$$

A problem here is that C^{-1} is a dense matrix. However, the diagonal of C is spectrally equivalent with C and a cheap approximation of C^{-1} is hence to invert the diagonal. We then obtain the following approximation of L:

$$L = A + B^* (\text{diag}(C))^{-1} B,$$

The matrix L is then in some sense a vector Laplacian, incorporating the mixed discretization technique. To test the efficiency of this preconditioner compared with more straightforward applications of AMG, we compare a couple of different problems. First, we test the preconditioners for the A and the L operators; that is, we estimate the condition number for the systems $P_1 A$ and $P_2 L$, where P_1 and P_2 is simply the algebraic multigrid preconditioners for A and L, respectively. Then we test the preconditioners

$$\mathcal{B}_1 = \begin{pmatrix} A & 0 \\ 0 & D \end{pmatrix}.$$

Here, D is a discrete Laplacian. The other preconditioner is

$$\mathcal{B}_2 = \begin{pmatrix} L & 0 \\ 0 & C \end{pmatrix}.$$

The following code demonstrate the construction of $\mathcal{B}_2$, $\mathcal{B}_1$ can be created in a similar fashion as described earlier.

Python code

```
V = FunctionSpace(mesh, "N1curl", 1)
Q = FunctionSpace(mesh, "Lagrange", 1)

v,u = TestFunction(V), TrialFunction(V)
q,p = TestFunction(Q), TrialFunction(Q)

A = assemble(dot(u,v)*dx + dot(curl(v), curl(u))*dx)
B = assemble(dot(grad(p),v)*dx)
C = assemble(dot(grad(q),u)*dx)
D = assemble(p*q*dx)

AA = block_mat([[A,  B], [C, -D]])
bb = block_vec([0,0])
L = collapse(A + B*InvDiag(D)*C)
```

h	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}
$P_1 A$	15.5	40.7	155	618	2370
$P_2 L$	1.7	2.2	5.5	18.4	68.9
$\mathcal{B}_1 \mathcal{A}$	4.1	6.8	14.9	44.1	148
$\mathcal{B}_2 \mathcal{A}$	5.6	8.8	23.2	76.7	287

Table 35.4: The estimated condition number κ with respect to various uniform mesh refinements and preconditioners for the mixed formulation of the Hodge Laplacian.

The complete code can be found in `hodge.py`.

In Table 35.4 we list the estimated condition numbers on various uniform mesh refinements on the unit cube. We use the lowest order Nédélec elements of first kind (Nédélec, 1980) combined with continuous piecewise linears. In this example we use homogeneous boundary conditions and $f = 0$, but we use a random start vector. Clearly, the simplest preconditioner P_1 does not work well for A, as compared with P_2 for L. However, it seems that for the fully coupled system, both preconditioners work quite well. The reason is probably that the P_1 preconditioner is poor main on gradients, but these gradients are closely related to p.

35.4 Conclusions

In this chapter we have demonstrated that advanced solution algorithms can be developed relatively easily by using CBC.Block and the Python interfaces of DOLFIN and Trilinos. The `cbc.block` module allows rather complicated block-partitioned preconditioners to be written in a simple form since it represent the linear operators as a graph. The Python linear algebra interface in DOLFIN allow us to write Krylov solvers and customize them in the language which these algorithms are typically expressed in books. Furthermore, it is relatively simple to employ state–of–the–art algebraic multigrid algorithms in Python using Trilinos. We remark that an alternative to PyTrilinos is PyAMG (Bell et al., 2011) which can be used together with the DOLFIN Python interface.

We have shown the implementation of block preconditioners for a few selected problems. Block preconditioners have been used in a variety of applications, we refer to Mardal and Winther (2011) and the references therein for a more complete discussion on this topic. For an overview of similar and alternative preconditioning techniques; see for example Benzi et al. (2005); Elman et al. (2005); Hiptmair (2006); Kirby (2010b).

36 *Automated testing of saddle point stability conditions*

By Marie E. Rognes

Over the last five decades, there has been a substantial body of research on the theory of mixed finite element methods. Mixed finite element methods are finite element methods where two or more finite element spaces are used to approximate separate variables. These methods have often been applied to saddle point problems arising from constrained minimization problems. Examples include the Stokes equations, the equations of Darcy flow (or the mixed Laplacian) or the Hellinger–Reissner formulation for linear elasticity. For equations involving several variables, and where elimination of any of the variables is not a viable option, the usefulness of such methods is evident. For other equations, discretizations based on the introduction of additional variables may have improved properties. The goal of this chapter is to demonstrate that one may automate the examination of the stability of any given discretization.

36.1 Background

For any discretization of a variational problem, stability is crucial to ensure well-posedness. For coercive problems, the discrete stability may often be easily ensured. For mixed discretizations of saddle point problems on the other hand, stability may be a nontrivial affair. Indeed, the mixed finite element spaces must usually be carefully chosen. The stability theory for mixed finite element discretizations originates from the work of Babuška (1973) and Brezzi (1974) in the early 1970's. Brezzi established two conditions ensuring the stability of a mixed finite element discretization of a canonical saddle point problem. Since then, many papers (and books) have been devoted to the identification and construction of specific stable mixed finite elements for specific saddle point problems (Arnold et al., 2006a; Brezzi et al., 1985a; Brezzi and Falk, 1991; Brezzi and Fortin, 1991; Raviart and Thomas, 1977; Taylor and Hood, 1973). Some of the analytical results are well known, such as the stability of the Taylor–Hood elements for the Stokes equations (Brezzi and Falk, 1991; Stenberg, 1984; Taylor and Hood, 1973). Others, such as the reduced stability of the $[\mathrm{CG}_1]^2 \times \mathrm{DG}_0$ elements on criss-cross triangulations for the mixed Laplacian (Boffi et al., 2000), may be less so.

As stated above, the goal of this chapter is to demonstrate that the process of numerically examining the stability of any given discretization can be automated. For a given discretization, the Brezzi constants are computable through a set of eigenvalue problems. These eigenvalue problems have previously been used to numerically study the stability of certain discretizations (Arnold and Rognes, 2009; Chapelle and Bathe, 1993; Qin, 1994). However, automation of this task has not been previously considered in the literature. A secondary aim is to show that the automation process is fairly easy given a software framework supporting the following components: a suitable range of different finite element spaces, easy support of bilinear forms defining equations and inner products,

and finally, a linear algebra backend with support for generalized, possibly singular, eigenvalue problems. The components of the FEniCS project provide these tools.

An automated stability tester provides several advantages. First, the notion of saddle point stability goes from something rather abstract to something rather hands-on. Moreover, even a novice user can easily get an overview of the available stable (or unstable) finite elements for a given equation. For research purposes, it provides a tool for the careful examination of discretizations that have stability properties depending on the tessellation structure. In particular, this framework has been used to study the stability properties of Lagrange elements for the mixed Laplacian (Arnold and Rognes, 2009).

This chapter is organized as follows. For motivational purposes, a simple example illustrating the importance of discrete stability is presented in Section 36.2. The subsequent two sections summarize the discrete stability theory of Babuška and Brezzi and how the stability constants involved can be computed through a set of eigenvalue problems. In Section 36.5, a strategy for the automation of numerical stability testing is presented. In particular, a light-weight Python module, ASCoT (Rognes, 2009), constructed on top of DOLFIN (Logg and Wells, 2010), is described. This module is freely available as a FEniCS Application at `https://launchpad.net/ascot`. The use and capabilities of this framework are demonstrated when applied to two classical examples: the mixed Laplacian and the Stokes equations in Section 36.6. Finally, Section 36.7 provides some concluding remarks and a discussion of limitations.

36.2 Why does discrete stability matter?

The following simple example illustrates that discrete stability is indeed crucial for the approximation of saddle point problems. Let $\Omega = (0,1)^2$ be the unit square in $\mathbb{R}^2$, and take $f = -2\pi^2 \sin(\pi x)\sin(\pi y)$. Consider the following mixed formulation of the Poisson problem with homogeneous Dirichlet boundary conditions: for the given data $f \in L^2(\Omega)$, find $\sigma \in H(\mathrm{div}, \Omega)$, and $u \in L^2(\Omega)$ such that

$$\begin{aligned} \langle \sigma, \tau \rangle + \langle \mathrm{div}\, \tau, u \rangle &= 0 \qquad \forall\, \tau \in H(\mathrm{div}, \Omega), \\ \langle \mathrm{div}\, \sigma, v \rangle &= \langle f, v \rangle \quad \forall\, v \in L^2(\Omega). \end{aligned} \tag{36.1}$$

This problem is well-posed: such solutions exist, are unique and depend continuously on the given data. In particular, $u = \sin(\pi x)\sin(\pi y)$ and $\sigma = \mathrm{grad}\, u$ solve (36.1).

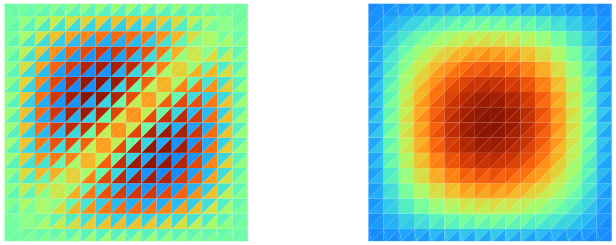
Next, let $\mathcal{T}_h$ be a uniform triangulation of the unit square that is formed by dividing the domain into $n \times n$ subsquares (with h the maximal triangle diameter) and dividing each square by the diagonal with positive slope. Given a pair of finite element spaces $\Sigma_h \times V_h$ defined relative to this tessellation, the equations (36.1) can be discretized in the standard manner: find $\sigma_h \in \Sigma_h$ and $u_h \in V_h$ such that

$$\begin{aligned} \langle \sigma_h, \tau \rangle + \langle \mathrm{div}\, \tau, u_h \rangle &= 0 \qquad \forall\, \tau \in \Sigma_h, \\ \langle \mathrm{div}\, \sigma_h, v \rangle &= \langle f, v \rangle \quad \forall\, v \in V_h. \end{aligned} \tag{36.2}$$

The final question becomes what finite element spaces Σ_h and V_h to choose. As we shall see, the well-posedness of the discrete problem will heavily rely on the choice of spaces.

First, let us consider a naive choice; namely, taking the space of continuous piecewise linear vector fields defined relative to $\mathcal{T}_h$ for the space Σ_h and the space of continuous piecewise linears for V_h. This choice turns out to be a rather bad one: the finite element matrix associated with this pair will be singular! Hence, there does not exist a discrete solution (σ_h, u_h) with this choice of $\Sigma_h \times V_h$.

As a second attempt, we keep the space of continuous piecewise linear vector fields for Σ_h, but replace the previous space V_h by the space of piecewise constant functions. This pair might appear to be a more attractive alternative: there does indeed exist a discrete solution (σ_h, u_h). However, the



(a) Bad approximation (b) Good approximation

Figure 36.1: The scalar variable approximation u_h for two choices of mixed finite element spaces for the mixed Laplacian. The data are as defined immediately above (36.1). The element spaces are $[\mathrm{CG}_1]^2 \times \mathrm{DG}_0$ in (a) and $\mathrm{RT}_1 \times \mathrm{DG}_0$ in (b). (The scales are less relevant for the current purpose and have therefore been omitted.)

discrete solution is not at all satisfactory. In particular, the approximation of the scalar variable u_h is highly oscillatory, see Figure 36.1(a), and hence it is a poor approximation to the correct solution.

The above two alternatives give unsatisfactory results because the discretizations defined by the element spaces are both unstable. A stable low order element pairing is the combination of the lowest order Raviart–Thomas elements and the space of piecewise constants (Raviart and Thomas, 1977). The corresponding u_h approximation is plotted in Figure 36.1(b). This approximation looks qualitatively correct.

The reason for the instabilities of the first two choices, and the stability of the third choice, may not be immediately obvious. The goal of this chapter is to construct a framework that automates this stability identification procedure, by characterizing the stability properties of a finite element discretization automatically and accurately. We will return to this example in Section 36.6 where we give a more careful characterization of the stability properties of the above sample elements.

36.3 Discrete stability

In order to automatically characterize the stability of a discretization, we need a precise definition of discrete stability and preferably conditions for such to hold. In this section, the Babuška and Brezzi stability conditions are described and motivated in the general abstract setting. The material presented here is largely taken from the classical references Babuška (1973); Brezzi (1974); Brezzi and Fortin (1991).

For a Hilbert space W, we denote the norm on W by $\|\cdot\|_W$ and the inner product by $\langle\cdot,\cdot\rangle_W$. Assume that c is a symmetric, bilinear form on W and that L is a continuous, linear form on W. We will consider the following canonical variational problem: find $u \in W$ such that

$$c(u, v) = L(v) \quad \forall\, v \in W. \tag{36.3}$$

Assume that c is continuous; that is, there exists a positive constant C such that

$$|c(u, v)| \leqslant C\, \|u\|_W \|v\|_W \quad \forall\, u, v \in W. \tag{36.4}$$

If additionally there exists a positive constant γ such that

$$c(u, u) \geqslant \gamma \|u\|_W^2, \tag{36.5}$$

the form c is by definition coercive. This is indeed the case for many variational formulations of partial differential equations arising from standard minimization problems. On the other hand, for many constrained minimization problems, such as those giving rise to saddle point problems, the corresponding c is not coercive. Fortunately, the coercivity condition is sufficient, but not necessary. A

weaker condition suffices: there exists a positive constant γ such that

$$0 < \gamma = \inf_{0 \neq u \in W} \sup_{0 \neq v \in W} \frac{|c(u,v)|}{\|u\|_W \|v\|_W}. \tag{36.6}$$

If the continuous c satisfies (36.6), there exists a unique $u \in W$ solving (36.3) (Babuška, 1973).

Now, we turn to consider discretizations of (36.3). Let $W_h \subset W$ be a finite dimensional subspace, and consider the discrete problem: find $u_h \in W_h$ such that

$$c(u_h, v) = L(v) \quad \forall\, v \in W_h. \tag{36.7}$$

For the discrete system to be well-posed, analogous conditions as for the continuous case must be satisfied. Note that c restricted to W_h is continuous *a fortiori*. However, the discrete analogue of (36.6) does not trivially hold. In order to guarantee that (36.7) has a unique solution, we must also have that there exists a positive constant γ_0 such that

$$0 < \gamma_0 \leqslant \gamma_h = \inf_{0 \neq u \in W_h} \sup_{0 \neq v \in W_h} \frac{|c(u,v)|}{\|u\|_W \|v\|_W}. \tag{36.8}$$

Moreover, in order to have uniform behavior in the limit as $h \to 0$, we must have that $\gamma_h \geqslant \gamma_0 > 0$ for all $h > 0$; that is, that γ_h is bounded from below independently of h (Babuška, 1973).

The condition (36.8) has a simple interpretation in the linear algebra perspective. Taking a basis $\{\phi_i\}_{i=1}^n$ for W_h, in combination with the Ansatz $u_h = u_j \phi_j$, we obtain the standard matrix formulation of (36.7):

$$C_{ij} u_j = L(\phi_i) \quad i = 1, \ldots, n, \tag{36.9}$$

where $C_{ij} = c(\phi_j, \phi_i)$. The Einstein notation, in which summation over repeated indices is implied, has been used here. This system will have a unique solution if the matrix C is non-singular, or equivalently, if the eigenvalues of C are nonzero. In the special case where c is coercive, all eigenvalues will in fact be positive. Moreover, we must ensure that the generalized eigenvalues (generalized with respect to the inner product on W) do not approach zero as $h \to 0$. This is precisely what is implied by the condition (36.8).

36.3.1 Stability conditions for saddle point problems

We now turn to consider the special case of abstract saddle point problems. In this case, the stability condition (36.8) can be rephrased in an alternative, but equivalent form.

Assume that V and Q are Hilbert spaces, that a is a continuous, symmetric, bilinear form on $V \times V$, that b is a continuous, bilinear form on $V \times Q$, and that L is a continuous linear form on $V \times Q$. A saddle point problem has the following canonical form: find $u \in V$ and $p \in Q$ such that

$$a(u,v) + b(v,p) + b(u,q) = L((v,q)) \quad \forall\, v \in V, q \in Q. \tag{36.10}$$

The system (36.10) is clearly a special case of (36.3) with the following identifications: let $W = V \times Q$, endow the product space with the norm $\|(v,q)\|_W = \|v\|_V + \|q\|_Q$, and label

$$c((u,p),(v,q)) = a(u,v) + b(v,p) + b(u,q). \tag{36.11}$$

Assuming that the condition (36.6) is satisfied, the above system admits a unique solution $(u,p) \in V \times Q$.

As in the general case, we aim to discretize (36.10), but now using a pair of conforming finite

element spaces V_h and Q_h. Letting $W_h = V_h \times Q_h$, we obtain the following special form of (36.7): find $u_h \in V_h$ and $p_h \in Q_h$ satisfying:

$$a(u_h, v) + b(v, p_h) + b(u_h, q) = L((v, q)) \quad \forall\, v \in V_h, q \in Q_h. \tag{36.12}$$

Again, the well-posedness of the discrete problem follows from the general theory. Applying the definition of (36.8) to (36.10), we define the Babuška constant γ_h:

$$\gamma_h = \inf_{0 \neq (u,p) \in W_h} \sup_{0 \neq (v,q) \in W_h} \frac{|a(u, v) + b(v, p) + b(u, q)|}{(\|u\|_V + \|p\|_Q)(\|v\|_V + \|q\|_Q)} \tag{36.13}$$

In particular, the discrete problem is well-posed if the Babuška stability condition holds; namely, if $\gamma_h \geqslant \gamma_0 > 0$ for any $h > 0$.

The previous deliberations simply summarized the general theory applied to the particular variational form defined by (36.10). However, the special structure of (36.10) also offers an alternative characterization. The single Babuška stability condition can be split into a pair of stability conditions as follows (Brezzi, 1974). Define

$$\alpha_h = \inf_{0 \neq u \in Z_h} \sup_{0 \neq v \in Z_h} \frac{a(u, v)}{\|u\|_V \|v\|_V}, \tag{36.14}$$

$$\beta_h = \inf_{0 \neq q \in Q_h} \sup_{0 \neq v \in V_h} \frac{b(v, q)}{\|v\|_V \|q\|_Q}, \tag{36.15}$$

where

$$Z_h = \{v \in V_h \,|\, b(v, q) = 0 \quad \forall\, q \in Q_h\}. \tag{36.16}$$

We shall refer to α_h as the Brezzi coercivity constant and β_h as the Brezzi inf-sup constant. The Brezzi stability conditions state that these must stay bounded above zero for all $h > 0$. The Brezzi conditions are indeed equivalent to the Babuška condition (Brezzi, 1974). However, for a specific saddle point problem and a given pair of function spaces, it might be easier to verify the two Brezzi conditions than the single Babuška condition. In summary, these conditions enable a concise characterization of the stability of discretizations of saddle point problems.

Definition 36.1 *A family of finite element discretizations $\{V_h \times Q_h\}_h$ is* stable in $V \times Q$ *if the Brezzi coercivity and inf-sup constants $\{\alpha_h\}_h$ and $\{\beta_h\}_h$ (or equivalently the Babuška inf-sup constants $\{\gamma_h\}_h$) are bounded from below by a positive constant independent of h.*

Throughout this chapter, the term *a family of discretizations* refers to a collection of finite element discretizations parametrized over a family of meshes.

There are families of discretizations that are not stable in the sense defined above, but possess a certain reduced stability. For a pair $V_h \times Q_h$, we can define the space of spurious modes $N_h \subseteq Q_h$:

$$N_h = \{q \in Q_h \,|\, b(v, q) = 0 \quad \forall\, v \in V_h\}. \tag{36.17}$$

It can be shown that the Brezzi inf-sup constant is positive if and only if there are no nontrivial spurious modes; that is, if $N_h = \{0\}$ (Qin, 1994). On the other hand, if N_h is nontrivial, one may, loosely speaking, think of the space Q_h as a bit too large. In that case, it may be natural to replace Q_h by the reduced space $N_h^{\perp}$, the orthogonal complement of N_h in Q_h. This idea motivates the definition

of the reduced Brezzi inf-sup constant, relating to the stability of $V_h \times N_h^\perp$:

$$\tilde{\beta}_h = \inf_{0 \neq q \in N_h^\perp} \sup_{0 \neq v \in V_h} \frac{b(v,q)}{\|v\|_V \|q\|_Q}, \tag{36.18}$$

and the definition of reduced stable below. By definition, $\tilde{\beta}_h \neq 0$. The identification of reduced stable discretizations can be interesting from a theoretical viewpoint. Further, such could be used for practical purposes after a filtration of the spurious modes.

Definition 36.2 *A family of discretizations $\{V_h \times Q_h\}_h$ is* reduced stable in $V \times Q$ *if the Brezzi coercivity constants $\{\alpha_h\}_h$ and the reduced Brezzi inf-sup constants $\{\tilde{\beta}_h\}_h$ are bounded from below by a positive constant independent of h.*

36.4 Eigenvalue problems associated with saddle point stability

For a given variational problem, the Brezzi conditions provide a method to inspect the stability of a family of conforming discretizations, defined relative to a family of meshes. However, it seems hardly feasible to automatically verify these conditions in their current form. Fortunately and as we shall see in this section, there is an alternative characterization of the Babuška and Brezzi constants: each stability constant will be related to the smallest (in modulus) eigenvalue of a certain eigenvalue problem. The automatic testing of the stability of a given discretization family can therefore be based on the computation and inspection of certain eigenvalues.

We begin by considering the Babuška inf-sup constant for the element pair $V_h \times Q_h$. It can be easily seen that the Babuška inf-sup constant $\gamma_h = |\lambda_{\min}|$ where $\lambda_{\min}$ is the smallest in modulus eigenvalue of the generalized eigenvalue problem (Arnold and Rognes, 2009; Malkus, 1981): find $0 \neq (u_h, p_h) \in V_h \times Q_h$ and $\lambda \in \mathbb{R}$ such that

$$a(u_h, v) + b(v, p_h) + b(u_h, q) = \lambda \left(\langle u_h, v \rangle_V + \langle p_h, q \rangle_Q \right) \quad \forall\, v \in V, q \in Q. \tag{36.19}$$

By the same arguments, the Brezzi coercivity constant α_h is the smallest in modulus eigenvalue of the following generalized eigenvalue problem: find $0 \neq u_h \in Z_h$ and $\lambda \in \mathbb{R}$ satisfying

$$a(u_h, v) = \lambda \langle u_h, v \rangle_V \tag{36.20}$$

For the spaces V_h and Q_h, a basis is normally known. For Z_h however, this is usually not the case. (If it had been, the space Z_h might have been better to compute with in the first place.) Therefore, the eigenvalue problem (36.20) is not that easily constructed in practice.

Instead, one may consider an alternative generalized eigenvalue problem: find $0 \neq (u_h, p_h) \in V_h \times Q_h$ and $\lambda \in \mathbb{R}$ satisfying

$$a(u_h, v) + b(v, p_h) + b(u_h, q) = \lambda \langle u_h, v \rangle_V \tag{36.21}$$

It can be shown that the smallest in modulus eigenvalue of the above eigenvalue problem and the smallest in modulus eigenvalue of (36.20) agree (Arnold and Rognes, 2009). Therefore $\alpha_h = |\lambda_{\min}|$ when $\lambda_{\min}$ is the smallest in modulus eigenvalue of (36.21). The eigenvalue problem (36.21) involves the spaces V_h and Q_h and is therefore more tractable. One word of caution however: if there exists a $q \in Q_h$ such that $b(v, q) = 0$ for all $v \in V_h$, then any λ is an eigenvalue of (36.21). Thus, the problem (36.21) is ill-posed if such q exists. The case where such q exists is precisely the case where the Brezzi inf-sup constant is zero.

Finally, the Brezzi inf-sup constant β_h is the square root of the smallest eigenvalue $\lambda_{\min}$ of the following eigenvalue problem (Malkus, 1981; Qin, 1994): find $0 \neq (u_h, p_h) \in V_h \times Q_h$ and $\lambda \in \mathbb{R}$ satisfying

$$\langle u_h, v \rangle_V + b(v, p_h) + b(u_h, q) = -\lambda \langle p_h, q \rangle_Q \tag{36.22}$$

The eigenvalues of (36.22) are all non-negative. Any eigenvector associated with a zero eigenvalue corresponds to a spurious mode. Further, the square root of the smallest nonzero eigenvalue will be the reduced Brezzi inf-sup constant (Qin, 1994).

36.5 Automating the stability testing

The mathematical framework is now in place. For a given variational formulation, given inner product(s), and a family of function spaces, the eigenvalue problem (36.19) or the problems (36.21) and (36.22) can be used to numerically check stability. The eigenvalue problem (36.22) applied to the Stokes equations was used in this context by Qin (1994) and Chapelle and Bathe (1993). A fully automated approach has not been previously available though. This is perhaps not so strange, as an automated approach would be rather challenging to implement within many finite element libraries. However, DOLFIN provides ample and suitable tools for this task. In particular, the UFL form language, the collection of finite element spaces supported by FIAT/FFC, and the available SLEPc eigenvalue solvers provide the required functionality.

The definition of an abstract saddle point problem (36.10) and the definition of stability of discretizations of such, Definition 36.1, provide a natural starting point. Based on these definitions, the testing of stability relies on the following input.

- The bilinear forms a and b defining a variational saddle point problem.
- The function spaces V and Q through the inner products $\langle \cdot, \cdot \rangle_V$ and $\langle \cdot, \cdot \rangle_Q$.
- A family of finite element function spaces $\{W_h\}_h = \{V_h \times Q_h\}_h$ parametrized over the mesh size h.

We pause to remark that since (36.10) is a special case of the canonical form (36.3), one may consider the Babuška constant only. However, for the analysis of saddle point problems, the separate behavior of the individual Brezzi constants may be interesting. For this reason, we focus on the Brezzi stability conditions and the decomposed variational form here.

The following strategy presents itself naturally in order to attempt to characterize the stability of a discretization family. With the above information, one can proceed in the following steps

1. For each function space W_h, construct the eigenvalue problems associated with the Brezzi conditions
2. Solve the eigenvalue problems and identify the appropriate eigenvalues corresponding to the Brezzi constants.
3. Based on the behavior of the Brezzi constants with respect to h, the discretization family should be classified, see Definitions 36.1 and 36.2, as
   (a) Stable
   (b) Unstable
   (c) Unstable, but reduced stable

The above strategy is implemented in the automated stability condition tester ASCoT (Rognes, 2009). ASCoT is a Python module dependent on DOLFIN compiled with SLEPc. It is designed to automatically evaluate the stability of a discretization family, and in particular, the stability of mixed finite element methods for saddle point problems. ASCoT can be imported as any Python module:

Python code

```
from ascot import *
```

The remainder of this section describes how the afore described strategy is implemented in ASCoT. Emphasis is placed on the form of the input, the construction and solving of the eigenvalue problems, and the classification of stability based on the stability constants.

Before continuing however, it is necessary to point out a limitation of the numerical testing. The mathematical definition of stability is indeed based on taking the limit as $h \to 0$. However, it is hardly feasible to examine an infinite family of function spaces $\{W_h\}_{h\in\mathbb{R}^+}$ numerically. In practice, one can only consider a finite set of spaces $\{W_{h_i}\}_{i\in(0,\dots,N)}$. Therefore, this strategy can only give numerical evidence, which must be interpreted using appropriate heuristics.

36.5.1 Defining input

ASCoT relies on the variational form language defined by UFL and DOLFIN for the specification of forms, inner products and function spaces. In order to illustrate, we take the discrete mixed Laplacian introduced in (36.2) as an example.

First and foremost, consider the specification of the forms a and b. Recall that discrete saddle point stability is not a property relating to a single set of function spaces, but rather a property relating to a family of function spaces. In the typical DOLFIN approach, forms are specified in terms of basis functions on a single function space. For our purposes, this seems like a less ideal approach. Instead, to be able to specify the forms independently of the function spaces, we can take advantage of the Python λ functionality. For the mixed Laplacian, the forms a and b read $a = a(u,v) = \langle u, v\rangle$ and $b = b(v,q) = \langle \operatorname{div} v, q\rangle$. These should be specified as

Python code

```
# Define a and b forms:
a = lambda u, v: dot(u, v)*dx
b = lambda v, q: div(v)*q*dx
```

The above format is advantageous as it separates the definition of the forms from the function spaces. Hence, the user needs not specify basis functions on each of the separate function spaces: ASCoT handles the initialization of the appropriate basis functions.

Second, the inner products $\langle\cdot,\cdot\rangle_V$ and $\langle\cdot,\cdot\rangle_Q$ must be provided. The inner products are bilinear forms and can therefore be viewed as a special case of the above. For the mixed Laplacian, the appropriate inner products are $\langle u, v\rangle_{\text{div}} = \langle u, v\rangle + \langle \operatorname{div} u, \operatorname{div} v\rangle$ and $\langle p, q\rangle_0 = \langle p, q\rangle$. The corresponding code reads

Python code

```
# Define inner products:
Hdiv = lambda u, v: (dot(u, v) + div(u)*div(v))*dx
L2 = lambda p, q: dot(p, q)*dx
```

Third, the function spaces have to be specified. In particular, a list of function spaces corresponding to a set of meshes should be defined. For the testing of the mixed function space consisting of continuous piecewise linear vector fields $[\text{CG}_1]^2$, combined with continuous piecewise linears CG_1, for a set of diagonal triangulations of the unit square, one can do as follows:

Python code

```
# Construct a family of mixed function spaces
meshsizes = [2, 4, 6, 8, 10]
meshes = [UnitSquare(n, n) for n in meshsizes]
W_hs = [VectorFunctionSpace(mesh, "Lagrange", 1)*FunctionSpace(mesh, "Lagrange", 1)
        for mesh in meshes]
```

Note that the reliability of the computed stability characterization increases with the number of meshes and their refinement level.

The stability of the above can now be tested. The main entry point function provided by ASCoT is `test_stability`. This function takes three arguments: a list of forms, a list of inner products and a list of function spaces:

Python code

```
result = test_stability((a, b), (Hdiv, L2), W_hs)
```

A `StabilityResult` is returned. The instructions carried out by this function and the properties of the `StabilityResult` are described in the subsequent paragraphs.

36.5.2 Constructing and solving eigenvalue problems

For the testing of saddle point problems, specified by the two forms a and b, it is assumed that the user wants to check the Brezzi conditions. In order to test these conditions, the Brezzi constants; that is, the Brezzi coercivity and Brezzi inf-sup constants, must be computed for each of the function spaces. ASCoT provides functionality for the computation of these constants: the functions `compute_brezzi_coercivity` and `compute_brezzi_infsup`.

Let us take a closer look at the implementation of `compute_brezzi_infsup`. The input consists of the form b, the inner products m and n, and a function space W_h (and optionally, an essential boundary condition `bc`). The aim is to construct the eigenvalue problem given by (36.22) and then solve this problem efficiently. To accomplish this, the basis functions on the function space W_h are defined first. The left and right-hand sides of the eigenvalue problems are specified through the forms defined by (36.22). These forms are sent to an `EigenProblem`, and the resulting eigenvalues are then used to initialize an `InfSupConstant`. The `InfSupConstant` class is a part of the characterization machinery and will be discussed further in the next subsection.

Python code

```
def compute_brezzi_infsup(b, (m, n), W, bc=None):
    """
    For a given form b: V x Q \rightarrow \R and inner products m and
    n defining V and Q respectively and a function space W = V_h x
    Q_h, compute the Brezzi inf-sup constant.
    """
    # Define forms for eigenproblem
    (u, p) = TrialFunctions(W)
    (v, q) = TestFunctions(W)
    lhs = m(u, v) + b(v, p) + b(u, q)
    rhs = - n(p, q)

    # Get parameters
    params = ascot_parameters["brezzi_infsup"]
    num = ascot_parameters["number_of_eigenvalues"]

    # Compute eigenvalues
    eigenvalues = EigenProblem(lhs, rhs, params, bc).solve(num)
    return InfSupConstant(W.mesh().hmax(), eigenvalues, operator=sqrt)
```

The computation of the Brezzi coercivity constant takes a virtually identical form, only differing in the definition of the left and right-hand sides (`lhs` and `rhs`). If only a single form c and a single inner product m is specified, the Babuška condition is tested by similar constructs.

The `EigenProblem` class is a simple wrapper class for the DOLFIN `SLEPcEigenSolver`, taking either a single form, corresponding to a standard eigenvalue problem, or two forms, corresponding to a generalized eigenvalue problem. The eigenvalue problems generated by the Babuška and Brezzi conditions are all generalized eigenvalue problems. For both the Brezzi conditions, the right-hand side matrix will always be singular. The left-hand side matrix may or may not be singular depending on the discretization. For the Babuška conditions, the right-hand side matrix should never be singular, however the left-hand side matrix may be.

SLEPc provides a collection of eigenvalue solvers that can handle generalized, possibly singular eigenvalue problems (Hernandez et al., 2005, 2009). The type of eigenvalue solver can be specified through the DOLFIN parameter interface. For our purposes, two solver types are particularly relevant: the LAPACK and the Krylov-Schur solvers. The LAPACK solver is a direct method. This solver is very robust. However, it computes all of the eigenvalues, and it is thus only suited for relatively small problems. In contrast, the Krylov-Schur method offers the possibility of only computing a given number of eigenvalues. Since the Brezzi constants are related to the eigenvalue closest to zero, it seems meaningful to only compute the eigenvalue of smallest magnitude. This solver is therefore set as the default solver type in ASCoT. Unfortunately, the Krylov-Schur solver is less robust for singular problems: it may fail to converge. A partial remedy may be to apply a shift-and-invert spectral transform with an appropriate shift factor to the eigenvalue problem. For more details on spectral transformations in SLEPc, see Hernandez et al. (2009). ASCoT applies a shift-and-invert transform with a small shift factor by default for the Brezzi and Babuška inf-sup problems.

36.5.3 Characterizing the discretization

After the eigenvalues and thus the stability constants are computed for the family of function spaces, all that remains is to interpret these constants. ASCoT provides three classes intended to represent and interpret the behavior of the stability constants: `InfSupConstant`, `InfSupCollection` and `StabilityResult`.

An `InfSupConstant` represents a single inf-sup constant. It is initialized using a mesh size h, a set of values, and an optional operator. The values typically correspond to the computed eigenvalues. If supplied, the operator is applied to the eigenvalues. For instance, ASCoT supplies a square root operator when computing the Brezzi inf-sup constant. The object can return the inf-sup constant and, if computed, the reduced inf-sup constant and the number of zero eigenvalues. The latter two items are most useful for careful analysis purposes.

A collection of `InfSupConstants` forms an `InfSupCollection`. An `InfSupCollection`'s main purpose is to identify whether or not the stability condition associated with the inf-sup constants holds. The method `is_stable` returns a boolean answer. The stability condition will not hold if any of the inf-sup constants is zero, and it will probably not hold if the inf-sup constants seem to decay with the mesh size h. The rate of decay r_i between two subsequent constants c_i and c_{i+1} is defined as:

$$r_i = \frac{\log_2(c_i) - \log_2(c_{i+1})}{\log_2(h_i) - \log_2(h_{i+1})} \tag{36.23}$$

where h_i is the corresponding mesh size. Currently, ASCoT classifies a discretization as stable if there are no singularities (no zero eigenvalues for all meshes), and the decay rates are below 1 and consistently decrease, or the rate corresponding to the finest mesh is less than a heuristically chosen small number.

Finally, the `StabilityResult` class holds a list of possibly several `InfSupCollections`, each corresponding to a separate inf-sup condition, such as the Brezzi coercivity and the Brezzi inf-sup condition. The `StabilityResult` identifies a discretization as stable if all stability conditions are satisfied, and as unstable otherwise.

36.6 Examples

In this section, we apply the automated stability testing framework to two classical saddle point problems: the mixed Laplacian and the Stokes equations. The behavior of the various mixed finite elements observed in Section 36.2 will be explained and classical analytical results reproduced. The complete code is available from the demo directory of the ASCoT module.

36.6.1 Mixed Laplacian

We can now return to the mixed Laplacian example described in Section 36.2 and inspect the Brezzi stability properties of the element spaces involved, namely $[\mathrm{CG}_1]^2 \times \mathrm{CG}_1$, $[\mathrm{CG}_1]^2 \times \mathrm{DG}_0$ and $\mathrm{RT}_1 \times \mathrm{DG}_0$. The example considered a family of diagonal triangulations of the unit square. The complete code required to test the stability of the first discretization family was presented piecewise in Section 36.5.1. The stability result can be inspected as follows:

Python code

```
print result
for condition in result.conditions:
    print condition
```

The following output appears:

Output

```
<Mixed element: (<Mixed element: (<CG1 on a <triangle of degree 1>>,
<CG1 on a <triangle of degree 1>>)>, <CG1 on a <triangle of degree 1>>)>

Not computing Brezzi coercivity constants because of singularity
Discretization family is: Unstable. Singular. Decaying.

InfSupCollection: beta_h
singularities =  [2, 2, 2, 2, 2]
reduced =        [0.56032, 0.35682, 0.24822, 0.18929, 0.15251]
rates  =         [0.651, 0.895, 0.942, 0.968]

Empty InfSupCollection: alpha_h
```

ASCoT characterizes this discretization family as unstable. For the Brezzi inf-sup eigenvalue problems, there are 2 zero eigenvalues for each mesh. Hence, the Brezzi inf-sup constant is zero, and moreover, the element matrix will be singular. This is precisely what we observed in the introductory example: there was no solution to the discrete system of equations. Moreover, the reduced inf-sup constant is also decaying with the mesh size at a rate that seems to be increasing towards $\mathcal{O}(h)$. So, there is no hope of recovering a stable method by filtering out the spurious modes. Since each Brezzi inf-sup constant is zero, the Brezzi coercivity eigenvalue problems are not computationally well-posed, and thus these constants have not been computed.

The second family of elements considered in Section 36.2 was the combination of continuous piecewise linear vector fields and piecewise constants. Using the same code as before, just replacing the finite element spaces, we obtain the following results:

Output

```
<Mixed element: (<Mixed element: (<CG1 on a <triangle of degree 1>>,
<CG1 on a <triangle of degree 1>>)>, <DG0 on a <triangle of degree 1>>)>
Discretization family is: Unstable. Decaying.

InfSupCollection: beta_h
values =         [0.96443, 0.84717, 0.71668, 0.60558, 0.51771]
rates  =         [0.187, 0.413, 0.586, 0.703]

InfSupCollection: alpha_h
values =         [1, 1, 1, 1, 1]
rates  =         [-1.35e-14, 6.13e-14, 3.88e-13, 4.05e-13]
```

Look at the Brezzi inf-sup constants first. In this case, there are no singular values, and hence the Brezzi inf-sup constants are positive. However, the constants seem to decay with the mesh size at increasing rates. Extrapolating, we can suppose that the constants β_h depend on the mesh size h and decay towards zero with h. ASCoT accordingly labels the discretization as unstable. Since there are no singular values, the Brezzi coercivity problem is well-posed. The Brezzi coercivity constants have therefore been computed. We see that the Brezzi coercivity constant is equal to one for all of the meshes tested. This is also easily deduced: the divergence of the velocity space is included in the pressure space and hence the Brezzi coercivity constant is indeed one for all meshes. Since neither constant is singular, we expect the discrete system of equations to be solvable – as we indeed saw in Section 36.2. The problem with this method hence only lies in the decaying Brezzi inf-sup constant. However, the instability did indeed manifest itself in the discrete approximation see Figure 36.1(a).

Finally, we can inspect a stable method, namely the lowest order Raviart–Thomas space combined with the space of piecewise constants:

Output

```
<Mixed element: (<RT1 on a <triangle of degree 1>>,
<DG0 on a <triangle of degree 1>>)>
Discretization family is: Stable.

InfSupCollection: beta_h
values =         [0.97682, 0.97597, 0.97577, 0.97569, 0.97566]
rates  =         [0.00126, 0.000508, 0.000265, 0.000162]

InfSupCollection: alpha_h
values =         [1, 1, 1, 1, 1]
rates  =         [5.6e-11, 1.39e-08, 1.64e-08, 2.24e-07]
```

ASCoT characterizes this mixed element method as stable. It is indeed proven so (Raviart and Thomas, 1977). The Brezzi coercivity constant is equal to 1 for all meshes tested and hence bounded from below. The Brezzi inf-sup constant definitely seems to be bounded from below. The constant will actually converge to the value $\sqrt{2}\pi(1+2\pi^2)^{-1/2}$, see Arnold and Rognes (2009). The satisfactory result observed in Figure 36.1(b) is thus agreement with the general theory.

Caveat emptor. It is worth noting that the stability properties of some mixed elements can vary dramatically. Here is one example: take the combination of continuous linear vector fields and piecewise constants for the mixed Laplacian. As we have seen above, this element family is non-singular on the diagonal mesh family, but the Brezzi inf-sup constants decay. However, if we inspect a family of criss-cross meshes, specified in DOLFIN using

Python code

```
meshes = [UnitSquare(n, n, "crossed") for n in meshsizes]
```

with the mesh sizes as before, the results are different:

Output

```
Discretization family is: Unstable. Singular. Reduced stable.

InfSupCollection: beta_h
singularities = [4, 16, 36, 64, 100]
reduced =        [0.97832, 0.97637, 0.97595, 0.97579, 0.97572]
rates  =         [0.00288, 0.00106, 0.000543, 0.000328]
```

For this mesh family, the Brezzi inf-sup constants are zero and thus the method is singular. (In fact, there are n^2 spurious modes for this element on this mesh (Qin, 1994).) However, the reduced Brezzi inf-sup constants seem to be bounded from below, and so the method could theoretically be stabilized by a removal of the spurious modes. For a careful study of the stability of Lagrange elements for the mixed Laplacian on various mesh families, see Arnold and Rognes (2009).

The results may be more different than illustrated above. A truly stable method will be stable for any admissible tessellation family, but there are methods that are stable on some mesh families, but not in general. Therefore, if determining whether a mixed element is appropriate or not, the discretization should be tested on more than a single mesh family.

36.6.2 Stokes

The Stokes equations is another classical and highly relevant saddle point problem. For simplicity, we here consider the following discrete formulation: find the velocity $u_h \in V_h$, and the pressure $p_h \in Q_h$ such that

$$\begin{aligned} \langle \operatorname{grad} u_h, \operatorname{grad} v \rangle + \langle \operatorname{div} v, p_h \rangle &= \langle f, v \rangle \qquad \forall\, v \in V_h, \\ \langle \operatorname{div} u_h, q \rangle &= 0 \quad \forall\, q \in Q_h. \end{aligned} \tag{36.24}$$

The previous example demonstrated that it is feasible, even easy, to test stability for any given family of discretizations. Taking this a step further, we can generate a set of all available conforming function spaces on a family of meshes, and test the stability of each. With this aim in mind, ASCoT provides some functionality for creating combinations of mixed function spaces given information on the value dimension of the spaces, the polynomial degree, the meshes and the desired regularity. For instance, to generate all available H^1-conforming vector fields of polynomial degree between 1 and 4 matched with L^2-conforming functions of polynomial degrees between 0 and 3 on a given set of meshes, define

Python code

```
specifications = {"value_dimensions": (2, 1),
                  "degrees": ((i,j) for i in range(1,5) for j in range(i)),
                  "spaces": ("H1", "L2")}
spaces = create_spaces(meshes, **specifications)
```

For the equations (36.24), the Brezzi coercivity condition always holds as long as V_h does not contain the constant functions. Therefore, it suffices to examine the Brezzi inf-sup condition. For simplicity though, we here examine the V_h spaces with no essential boundary conditions prescribed. With spaces generated as above, this can be accomplished as follows:

Python code

```
# Define b form
b = lambda v, q: div(v)*q*dx
```

1. $[\mathrm{CG}_2]^2 \times \mathrm{CG}_1$
2. $[\mathrm{CG}_3]^2 \times \mathrm{CG}_1$
3. $[\mathrm{CG}_3]^2 \times \mathrm{CG}_2$
4. $[\mathrm{CG}_4]^2 \times \mathrm{CG}_1$
5. $[\mathrm{CG}_4]^2 \times \mathrm{CG}_2$
6. $[\mathrm{CG}_4]^2 \times \mathrm{CG}_3$
7. $[\mathrm{CG}_2]^2 \times \mathrm{DG}_0$
8. $[\mathrm{CG}_3]^2 \times \mathrm{DG}_0$
9. $[\mathrm{CG}_3]^2 \times \mathrm{DG}_1$
10. $[\mathrm{CG}_4]^2 \times \mathrm{DG}_0$
11. $[\mathrm{CG}_4]^2 \times \mathrm{DG}_1$
12. $[\mathrm{CG}_4]^2 \times \mathrm{DG}_2$
13. $[\mathrm{CG}_4]^2 \times \mathrm{DG}_3$

Figure 36.2: List of elements identified as satisfying the Brezzi inf-sup condition for the Stokes equations on a family of diagonal triangulations of the unit square.

```
# Define inner products:
H1 = lambda u, v: (dot(u, v) + inner(grad(u), grad(v)))*dx
L2 = lambda p, q: dot(p, q)*dx

# Test Brezzi inf-sup condition for the generated spaces
for W_hs in spaces:
    beta_hs = [compute_brezzi_infsup(b, (H1, L2), W_h) for W_h in W_hs]
    result = StabilityResult(InfSupCollection(beta_hs, "beta_h"))
```

Finally, ASCoT provides an optimized mode where only the stability of a discretization family is detected and not possible reduced stabilities. This mode is off by default, but can easily be turned on:

Python code

```
ascot_parameters["only_stable"] = True
```

Applying the above to the diagonal mesh family used in the previous example and printing those elements that are classified as stable result in the list of mixed elements summarized in Figure 36.2. The first item on this list is the lowest order Taylor–Hood element, while the third and sixth items are the next elements of the Taylor–Hood family: $[\mathrm{CG}_{k+1}]^2 \times \mathrm{CG}_k$ for $k \geqslant 1$. These mixed elements are indeed stable for any family of tessellations consisting of more than three triangles (Taylor and Hood, 1973; Stenberg, 1984; Brezzi and Falk, 1991). The seventh item on the list is the $[\mathrm{CG}_2]^2 \times \mathrm{DG}_0$ element (Crouzeix and Raviart, 1973), while the 9'th and 12'th item are the next order elements of the $[\mathrm{CG}_{k+1}]^2 \times \mathrm{DG}_{k-1}$ family, which again is truly stable for $k \geqslant 1$. The 13'th item on this list, $[\mathrm{CG}_4]^2 \times \mathrm{DG}_3$ is the lowest order Scott–Vogelius element. This element is the lowest order element of the Scott–Vogelius family $[\mathrm{CG}_k]^2 \times \mathrm{DG}_{k-1}$ for $k \geqslant 4$. Note that these elements for $k = 1, 2, 3$ are not on the list — as they should not: these lower order mixed elements are indeed unstable on this tessellation family (Qin, 1994). The stability of the remaining elements follow from the previous results: if the Brezzi inf-sup condition holds for a family $\{V_h \times Q_h\}$, by definition it will also hold for the families $\{V_h \times P_h\}$ for $P_h \subseteq Q_h$.

In conclusion, the elements identified are indeed known to be stable, and the list comprises all the stable conforming finite elements for the Stokes equations on this tessellation family that are available in FFC and generated by the `create_spaces` function.

36.7 Conclusions

This chapter describes an automated strategy for the testing of stability conditions for mixed finite element discretizations. The strategy has been implemented as a very light-weight Python module, ASCoT, on top of DOLFIN. The implementation is light-weight because of the powerful tools provided by the DOLFIN module, in particular the flexible form language provided through UFL/FFC, the availability of arbitrary order mixed finite elements of various families, and the SLEPc eigenvalue solvers.

We have seen that the automated stability tester has successfully identified available stable and unstable elements when applied to the Stokes equations for a diagonal tessellation family. Moreover, the framework has been used to identify previously unknown stability properties for lower order Lagrange elements for the mixed Laplacian (Arnold and Rognes, 2009).

There are however some limitations. First, numerical evidence is not analytical evidence. The tester makes a stability conjecture based on the computed constants. The conjecture may in some cases be erroneous, and the reliability of this conjecture may be low if only a few meshes are considered. Second, solving generalized, singular eigenvalue problems can be nontrivial. For the Brezzi coercivity constants, the Krylov-Schur solver easily fails to converge even with an applied shift-and-invert spectral transform. In such a case, one must either return to use a LAPACK-type solver or consider the Babuška constant directly.

List of authors

The following authors have contributed to this book.

Martin Sandve Alnæs
Chapters 14, 15, 16 and 17

Center for Biomedical Computing at Simula Research Laboratory, Norway
Department of Informatics, University of Oslo, Norway

This work is supported by an Outstanding Young Investigator grant from the Research Council of Norway, NFR 162730. This work is also supported by a Center of Excellence grant from the Research Council of Norway to the Center for Biomedical Computing at Simula Research Laboratory.

Stuart R. Clark
Chapter 31

Computational Geoscience Department at Simula Research Laboratory, Norway

This work is funded by a research grant from Statoil.

David B. Davidson
Chapter 34

Department of Electrical and Electronic Engineering, Stellenbosch University, South Africa

This work is supported by grants from the National Research Foundation, South Africa, as well as the Centre for High Performance Computing, South Africa as part of a flagship project.

Rodrigo Vilela De Abreu
Chapter 28

School of Computer Science and Communication, KTH, Sweden

Cem Degirmenci
Chapter 18

School of Computer Science and Communication, KTH, Sweden

Joachim Berdal Haga
Chapter 35

Center for Biomedical Computing at Simula Research Laboratory, Norway

This work is supported by a Center of Excellence grant from the Research Council of Norway to the Center for Biomedical Computing at Simula Research Laboratory.

Johan Hake
Chapters 10, 19 and 33

Department of Bioengineering, UCSD, USA
Center for Biomedical Computing at Simula Research Laboratory, Norway

This work is supported by the National Biomedical Computational Resource (NIH grant 5P41RR08605-17). This work is also supported by a Center of Excellence grant from the Research Council of Norway to the Center for Biomedical Computing at Simula Research Laboratory.

Johan Hoffman
Chapters 18 and 28

School of Computer Science and Communication, KTH, Sweden

Johan Jansson
Chapters 18 and 28

School of Computer Science and Communication, KTH, Sweden

Niclas Jansson
Chapters 18 and 28

School of Computer Science and Communication, KTH, Sweden

Claes Johnson
Chapter 28

School of Computer Science and Communication, KTH, Sweden

Robert C. Kirby
Chapters 2, 3, 4, 5, 8, 9, 12, 13 and 20

Department of Mathematics and Statistics, Texas Tech University, USA

Matthew Gregg Knepley
Chapter 20

Computation Institute, University of Chicago, Chicago, USA
Department of Molecular Biology and Physiology, Rush University Medical Center, Chicago, USA
Mathematics and Computer Science Division, Argonne National Laboratory, Chicago, USA

This work is supported by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

Hans Petter Langtangen
Chapter 1

Center for Biomedical Computing at Simula Research Laboratory, Norway
Department of Informatics, University of Oslo, Norway

This work is supported by a Center of Excellence grant from the Research Council of Norway to the Center for Biomedical Computing at Simula Research Laboratory.

Evan Lezar
Chapter 34

Department of Electrical and Electronic Engineering, Stellenbosch University, South Africa

This work is supported by grants from the National Research Foundation, South Africa, as well as the Centre for High Performance Computing, South Africa as part of a flagship project.

Svein Linge
Chapter 24

Telemark University College and the Center for Biomedical Computing at Simula Research Laboratory, Norway

This work is supported by a Center of Excellence grant from the Research Council of Norway to the Center for Biomedical Computing at Simula Research Laboratory.

Anders Logg
Chapters 2, 3, 5, 6, 8, 9, 10, 11, 12, 16, 21 and 23

Center for Biomedical Computing at Simula Research Laboratory, Norway
Department of Informatics, University of Oslo, Norway

This work is supported by an Outstanding Young Investigator grant from the Research Council of Norway, NFR 180450. This work is also supported by a Center of Excellence grant from the Research Council of Norway to the Center for Biomedical Computing at Simula Research Laboratory.

Nuno D. Lopes
Chapter 25

ADEM, Área Departamental de Matemática, ISEL, Instituto Superior de Engenharia de Lisboa, Portugal
Departamento de Matemática, FCT–UNL, Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa, Portugal
CMAF, Centro de Matemática e Aplicações Fundamentais, Lisboa, Portugal

This work is supported by ISEL, Instituto Superior de Engenharia de Lisboa.

Alf Emil Løvgren
Chapter 24

Nofas Management AS, Norway

This work is supported by a Center of Excellence grant from the Research Council of Norway to the Center for Biomedical Computing at Simula Research Laboratory.

Kent-Andre Mardal
Chapters 4, 6, 14, 15, 16, 19, 21, 22, 23, 24 and 35

Center for Biomedical Computing at Simula Research Laboratory, Norway
Department of Informatics, University of Oslo, Norway

This work is supported by the NFR Grant 209951 and a Center of Excellence grant from the Research Council of Norway to the Center for Biomedical Computing at Simula Research Laboratory.

Mikael Mortensen
Chapters 21 and 22

Norwegian Defence Research Establishment (FFI), Norway

Harish Narayanan
Chapters 21 and 27

Center for Biomedical Computing at Simula Research Laboratory, Norway

This work is supported by an Outstanding Young Investigator grant from the Research Council of Norway, NFR 180450. This work is also supported by a Center of Excellence grant from the Research Council of Norway to the Center for Biomedical Computing at Simula Research Laboratory.

Murtazo Nazarov
Chapter 18

School of Computer Science and Communication, KTH, Sweden

Mehdi Nikbakht
Chapter 30

Faculty of Civil Engineering and Geosciences, Delft University of Technology, The Netherlands

Support from the Netherlands Technology Foundation STW, the Netherlands Organisation for Scientific Research and the Ministry of Public Works and Water Management is gratefully acknowledged.

Pedro J. S. Pereira
Chapter 25

ADEM, Área Departamental de Matemática, ISEL, Instituto Superior de Engenharia de Lisboa, Portugal
CEFITEC, Centro de Física e Investigação Tecnológica, FCT–UNL, Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa, Portugal

Johannes Ring
FEniCS packaging, buildbots, benchbots and build system

Center for Biomedical Computing at Simula Research Laboratory, Norway

This work is supported by a Center of Excellence grant from the Research Council of Norway to the Center for Biomedical Computing at Simula Research Laboratory.

Marie E. Rognes
Chapters 3, 11, 31 and 36

Center for Biomedical Computing at Simula Research Laboratory, Norway

This work is supported by an Outstanding Young Investigator grant from the Research Council of Norway, NFR 180450. This work is also supported by a Center of Excellence grant from the Research Council of Norway to the Center for Biomedical Computing at Simula Research Laboratory.

Hans Joachim Schroll
Chapter 32

Department of Mathematics and Computer Science, University of Southern Denmark, Odense, Denmark

This work is funded by a research grant from Statoil. The work has been conducted at Simula Research Laboratory, Norway.

L. Ridgway Scott
Chapter 20

Departments of Computer Science and Mathematics, Institute for Biophysical Dynamics and the Computation Institute, University of Chicago, USA

This work is supported in part by NSF grant DMS-0920960.

Kristoffer Selim
Chapter 29

Center for Biomedical Computing at Simula Research Laboratory, Norway
Department of Informatics, University of Oslo, Norway

This work is supported by an Outstanding Young Investigator grant from the Research Council of Norway, NFR 180450. This work is also supported by a Center of Excellence grant from the Research Council of Norway to the Center for Biomedical Computing at Simula Research Laboratory.

Susanne Støle-Hentschel
Chapter 24

Det Norske Veritas AS

This work is supported by a Center of Excellence grant from the Research Council of Norway to the Center for Biomedical Computing at Simula Research Laboratory.

Andy R. Terrel
Chapters 3, 9 and 20

Enthought Inc., Austin, Texas, USA

Luís Trabucho
Chapter 25

Departamento de Matemática, FCT–UNL, Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa, Portugal
CMAF, Centro de Matemática e Aplicações Fundamentais, Lisboa, Portugal

This work is partially supported by Fundação para a Ciência e Tecnologia, Financiamento Base 2008-ISFL-1-209.

Kristian Valen-Sendstad
Chapters 21 and 23

Simula School of Research and Innovation
Center for Biomedical Computing at Simula Research Laboratory, Norway

This work is supported by a Center of Excellence grant from the Research Council of Norway to the Center for Biomedical Computing at Simula Research Laboratory.

Lyudmyla Vynnytska
Chapter 31

Computational Geoscience Department at Simula Research Laboratory, Norway

This work is funded by a research grant from Statoil.

Garth N. Wells
Chapters 6, 7, 10, 11, 20, 26 and 30

Department of Enginering, University of Cambridge, United Kingdom

Ilmar M. Wilbers
Chapter 14

Center for Biomedical Computing at Simula Research Laboratory, Norway

This work is supported by a Center of Excellence grant from the Research Council of Norway to the Center for Biomedical Computing at Simula Research Laboratory.

Kristian B. Ølgaard
Chapters 7, 11 and 26

Faculty of Civil Engineering and Geosciences, Delft University of Technology, The Netherlands

This work is supported by the Netherlands Technology Foundation STW, the Netherlands Organisation for Scientific Research and the Ministry of Public Works and Water Management.

GNU Free Documentation License

Version 1.3, 3 November 2008

Copyright © 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc.

`http://fsf.org`

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

0. Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. 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.

1. Applicability and definitions

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

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

A "**Secondary Section**" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

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

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

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

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

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

The "**publisher**" means any person or entity that distributes copies of the Document to the public.

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

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

2. 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.

3. *Copying in quantity*

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

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

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. *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:

A. 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.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. 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.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. 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.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. 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.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. *Combining documents*

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. *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.

7. *Aggregation with independent works*

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. *Translation*

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that

you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. Termination

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. 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. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. Relicensing

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in

San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

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 1.3 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".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with ... Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

References

S. Adams and B. Cockburn. A mixed finite element method for elasticity in three dimensions. *Journal of Scientific Computing*, 25(3):515–521, 2005. URL `http://dx.doi.org/10.1007/s10915-004-4807-3`.

M. Ainsworth and J. Coyle. Hierarchic finite element bases on unstructured tetrahedral meshes. *International Journal for Numerical Methods in Engineering*, 58(14):2103–2130, 2003. URL `http://dx.doi.org/10.1002/nme.847`.

M. Ainsworth and J. T. Oden. A unified approach to *a posteriori* error estimation using element residual methods. *Numerische Mathematik*, 65(1):23–50, 1993. URL `http://dx.doi.org/10.1007/BF01385738`.

M. Ainsworth and J. T. Oden. A Posteriori *Error Estimation in Finite Element Analysis*. Wiley and Sons, New York, 2000.

J. Alastruey, K. H. Parker, J. Peiro, S. M. Byrd, and S. J. Sherwin. Modelling the circle of Willis to assess the effects of anatomical variations and occlusions on cerebral flows. *Journal of Biomechanics*, 40(8):1794–1805, 2007. URL `http://dx.doi.org/10.1016/j.jbiomech.2006.07.008`.

B. Alberts, D. Bray, J. Lewis, M. Raff, K. Roberts, and J. D. Watson. *Molecular Biology of the Cell*. Garland, 4th edition, 2002.

V. Alfred, R. Sethi, and D. U. Jeffrey. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, Massachusetts, 1986.

M. S. Alnæs, A. Logg, K.-A. Mardal, O. Skavhaug, and H. P. Langtangen. *UFC Specification and User Manual*, 2007. URL `http://fenicsproject.org`.

M. S. Alnæs and A. Logg. *UFL Specification and User Manual*, 2009. URL `http://fenicsproject.org`.

M. S. Alnæs and K.-A. Mardal. *SyFi*, 2009. URL `http://fenicsproject.org`.

M. S. Alnæs and K.-A. Mardal. On the efficiency of symbolic computations combined with code generation for finite element methods. *ACM Trans. Math. Softw.*, 37:6:1–6:26, 2010. URL `http://dx.doi.org/10.1145/1644001.1644007`.

M. S. Alnæs, A. Logg, K.-A. Mardal, O. Skavhaug, and H. P. Langtangen. Unified framework for finite element assembly. *International Journal of Computational Science and Engineering*, 4(4):231–244, 2009.

N. Alperin, M. Mazda, T. Lichtor, and S. H. Lee. From cerebrospinal fluid pulsation to noninvasive intracranial compliance and pressure measured by MRI flow studies. *Current Medical Imaging Reviews*, 2:117–129, 2006.

J. H. Argyris, I. Fried, and D. W. Scharpf. The TUBA family of plate elements for the matrix displacement method. *The Aeronautical Journal of the Royal Aeronautical Society*, 72:701–709, 1968.

D. N. Arnold. An interior penalty finite element method with discontinuous elements. *SIAM J. Numer. Anal.*, 19(4), 1982. URL http://dx.doi.org/10.1137/0719052.

D. N. Arnold and R. S. Falk. A uniformly accurate finite element method for the Reissner–Mindlin plate. *SIAM J. Num. Anal.*, 26:1276–1290, 1989. URL http://dx.doi.org/10.1137/0726074.

D. N. Arnold and M. E. Rognes. Stability of Lagrange elements for the mixed Laplacian. *Calcolo*, 46(4): 245–260, 2009. URL http://dx.doi.org/10.1007/s10092-009-0009-6.

D. N. Arnold and R. Winther. Mixed finite elements for elasticity. *Numer. Math.*, 92(3):401–419, 2002. URL http://dx.doi.org/10.1007/s002110100348.

D. N. Arnold, F. Brezzi, and J. Douglas, Jr. PEERS: a new mixed finite element for plane elasticity. *Japan J. Appl. Math.*, 1(2):347–367, 1984a. URL http://dx.doi.org/10.1007/BF03167064.

D. N. Arnold, F. Brezzi, and M. Fortin. A stable finite element for the Stokes equations. *Calcolo*, 21(4): 337–344, 1984b. URL http://dx.doi.org/10.1007/BF02576171.

D. N. Arnold, R. S. Falk, and R. Winther. Preconditioning in $H(\mathrm{div})$ and applications. *Mathematics of Computation*, 66(219):957–984, 1997. URL http://dx.doi.org/10.1090/S0025-5718-97-00826-0.

D. N. Arnold, R. S. Falk, and R. Winther. Multigrid in $H(\mathrm{div})$ and $H(\mathrm{curl})$. *Numerische Mathematik*, 85(2):197–217, 2000. URL http://dx/doi.org/10.1007/PL00005386.

D. N. Arnold, R. S. Falk, and R. Winther. Finite element exterior calculus, homological techniques, and applications. *Acta Numerica*, 15:1–155, 2006a. URL http://dx.doi.org/10.1017/S0962492906210018.

D. N. Arnold, R. S. Falk, and R. Winther. Differential complexes and stability of finite element methods. II. The elasticity complex. In *Compatible spatial discretizations*, volume 142 of *IMA Vol. Math. Appl.*, pages 47–67. Springer, New York, 2006b.

D. N. Arnold, R. S. Falk, and R. Winther. Mixed finite element methods for linear elasticity with weakly imposed symmetry. *Mathematics of Computation*, 76(260):1699–1723, 2007.

D. N. Arnold, G. Awanou, and R. Winther. Finite elements for symmetric tensors in three dimensions. *Math. Comp.*, 77(263):1229–1251, 2008.

O. Axelsson. A general incomplete block-matrix factorization method. *Linear Algebra and its Applications*, 74:179–190, 1986. ISSN 0024-3795.

I. Babuška. The finite element method with Lagrangian multipliers. *Numer. Math.*, 20:179–192, 1973. URL http://dx.doi.org/10.1007/BF01436561.

I. Babuška and W. C. Rheinboldt. A posteriori error estimates for the finite element method. *Int. J. Numer. Meth. Engrg.*, pages 1597–1615, 1978.

I. Babuška, U. Banerjee, and J. E. Osborn. Survey of meshless and generalized finite element methods: A unified approach. *Acta Numerica*, 12:1–125, 2003.

B. Bagheri and L. R. Scott. About Analysa. Technical Report TR–2004–09, University of Chicago, Department of Computer Science, 2004. URL http://www.cs.uchicago.edu/research/publications/techreports/TR-2004-09.

S. Balay, K. Buschelman, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang. PETSc web page, 2001. URL http://www.mcs.anl.gov/petsc.

S. Balay, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 2.1.5, Argonne National Laboratory, 2004.

W. Bangerth, R. Hartmann, and G. Kanschat. deal.II — a general-purpose object-oriented finite element library. *ACM Trans. Math. Softw.*, 33(4), 2007. URL `http://dx.doi.org/10.1145/1268776.1268779`.

R. E. Bank and A. Weiser. Some a posteriori error estimators for elliptic partial differential equations. *Mathematics of Computation*, pages 283–301, 1985.

G. K. Batchelor. *An Introduction to Fluid Dynamics*. Cambridge University Press, 1967.

C. Bauer, A. Frink, and R. Kreckel. Introduction to the GiNaC framework for symbolic computation within the C++ programming language. *Journal of Symbolic Computation*, 33(1):1–1, 2002. URL `http://dx.doi.org/10.1006/jsco.2001.0494`.

D. M. Beazley. SWIG: An easy to use tool for integrating scripting languages with C and C++. In *Proceedings of the 4th conference on USENIX Tcl/Tk Workshop*, volume 4, Berkeley, 1996. USENIX Association. URL `http://portal.acm.org/citation.cfm?id=1267498.1267513`.

E. B. Becker, G. F. Carey, and J. T. Oden. *Finite Elements: An Introduction*. Prentice–Hall, Englewood–Cliffs, 1981.

R. Becker and R. Rannacher. An optimal control approach to a posteriori error estimation in finite element methods. *Acta Numerica*, 10:1–102, 2001.

S. Beji and K. Nadaoka. A formal derivation and numerical modelling of the improved Boussinesq equations for varying depth. *Ocean engineering*, 23(8):691, 1996. ISSN 0029-8018.

W. N. Bell, L. N. Olson, and J. Schroder. PyAMG: Algebraic multigrid solvers in Python, 2011. URL `http://www.pyamg.org`.

T. Belytschko and T. Black. Elastic crack growth in finite elements with minimal remeshing. *International Journal for Numerical Methods in Engineering*, 45(5):601–620, 1999.

F. Bengzon and M. G. Larson. Adaptive finite element approximation of multiphysics problems: A fluid–structure interaction model problem. *International Journal for Numerical Methods in Engineering*, 2010.

M. Benzi, G. H. Golub, and J. Liesen. Numerical solution of saddle point problems. *Acta Numerica*, 14: 1–137, 2005.

D. M. Bers. *Excitation-Contraction Coupling and Cardiac Contractile Force*. Kluwer Academic Publishers, Dordrecht, 2nd edition, 2001.

D. J. Beuckelmann and W. G. Wier. Mechanism of release of calcium from sarcoplasmic reticulum of guinea-pig cardiac cells. *J. Physiol.*, 405:233–255, Nov 1988.

W. B. Bickford. *A First Course in the Finite Element Method*. Irwin, 1994.

P. Bientinesi, J. A. Gunnels, M. E. Myers, E. S. Quintana-Ortí, and R. A. van de Geijn. The science of deriving dense linear algebra algorithms. *ACM Transactions on Mathematical Software*, 31(1):1–26, 2005. URL `http://dx.doi.org/10.1145/1055531.1055532`.

H. B. Bingham, P. A. Madsen, and D. R. Fuhrman. Velocity potential formulations of highly accurate Boussinesq-type models. *Coastal Engineering*, Article in Press, 2008.

C. Bischof, A. Carle, G. Corliss, and A. Griewank. ADIFOR: Automatic differentiation in a source translator environment. In *Papers from the international symposium on Symbolic and algebraic computation*, pages 294–302, Berkeley, California, 1992. URL http://dx.doi.org/10.1145/143242.143335.

C. H. Bischof, P. D. Hovland, and B. Norris. Implementation of automatic differentiation tools. In *Proceedings of the 2002 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*, pages 98–107, Portland, Oregon, 2002. URL http://dx.doi.org/10.1145/503032.503047.

H. Blackburn. Semtex, 2011. URL http://users.monash.edu.au/~bburn/semtex.html.

H. M. Blackburn and S. J. Sherwin. Formulation of a Galerkin spectral element-Fourier method for three-dimensional incompressible flows in cylindrical geometries. *J. Comput. Phys.*, 197:759–778, 2004. URL http://dx.doi.org/10.1016/j.jcp.2004.02.013.

B. Blankenbach, F. Busse, U. Christensen, et al. A benchmark comparison for mantle convection codes. *Geophys. J. Int.*, 98:23–28, 1989.

D. Boffi. Three-dimensional finite element methods for the Stokes problem. *SIAM J. Numer. Anal.*, 34 (2):664–670, 1997. URL http://dx.doi.org/10.1137/S0036142994270193.

D. Boffi, F. Brezzi, and L. Gastaldi. On the problem of spurious eigenvalues in the approximation of linear elliptic problems in mixed form. *Math. Comp.*, 69(229):121–140, 2000.

J. Bonet and R. D. Wood. *Nonlinear Continuum Mechanics for Finite Element Analysis*. Cambridge University Press, 1997.

A. Bossavit. *Computational Electromagnetics: Variational Formulations, Complementarity, Edge Elements*. Academic Press, 1998.

L. Boussel, V. Rayz, C. McCulloch, A. Martin, G. Acevedo-Bolton, M. Lawton, R. Higashida, W. S. Smith, W. L. Young, and D. Saloner. Aneurysm growth occurs at region of low wall shear stress: Patient-specific correlation of hemodynamics and growth in a longitudinal study. *Stroke*, 39(11): 2997–3002, 2008. URL http://dx.doi.org/10.1161/STROKEAHA.108.521617.

D. Braess. *Finite Elements*. Cambridge University Press, third edition, 2007. ISBN 978-0-521-70518-9.

J. H. Bramble. *Multigrid Methods*, volume 294 of *Pitman Research Notes in Mathematical Sciences*. Longman Scientific & Technical, Essex, England, 1993.

J. H. Bramble and M. Zlámal. Triangular elements in the finite element method. *Mathematics of Computation*, pages 809–820, 1970.

A. Brandt. Multi-level adaptive solutions to boundary-value problems. *Mathematics of Computation*, pages 333–390, 1977.

S. C. Brenner and L. R. Scott. *The Mathematical Theory of Finite Element Methods*, volume 15 of *Texts in Applied Mathematics*. Springer, New York, third edition, 2008. URL http://dx.doi.org/10.1007/978-0-387-75934-0.

F. Brezzi. On the existence, uniqueness and approximation of saddle-point problems arising from Lagrangian multipliers. *RAIRO Anal. Numér.*, R–2:129–151, 1974.

F. Brezzi and R. S. Falk. Stability of higher-order Hood-Taylor methods. *SIAM J. Numer. Anal.*, 28(3): 581–590, 1991.

F. Brezzi and M. Fortin. *Mixed and Hybrid Finite Element Methods*, volume 15 of *Springer Series in Computational Mathematics*. Springer-Verlag, New York, 1991.

F. Brezzi, J. Douglas, Jr., and L. D. Marini. Two families of mixed finite elements for second order elliptic problems. *Numer. Math.*, 47(2):217–235, 1985a.

F. Brezzi, J. Douglas, Jr., and L. D. Marini. Variable degree mixed methods for second order elliptic problems. *Mat. Apl. Comput.*, 4(1):19–34, 1985b.

F. Brezzi, J. Douglas, R. Duran, and M. Fortin. Mixed finite elements for second order elliptic problems in three variables. *Numerische Matematik*, 51:237–250, 1987a.

F. Brezzi, J. Douglas, Jr., M. Fortin, and L. D. Marini. Efficient rectangular mixed finite elements in two and three space variables. *RAIRO Modél. Math. Anal. Numér.*, 21(4):581–604, 1987b.

A. N. Brooks and T. J. R. Hughes. Streamline upwind/Petrov–Galerkin formulations for convection dominated flows with particular emphasis on the incompressible Navier–Stokes equations. *Computer Methods in Applied Mechanics and Engineering*, 32(1-3):199–259, Sept. 1982. URL `http://dx.doi.org/10.1016/0045-7825(82)90071-8`.

A. Buffa, P. Houston, and I. Perugia. Discontinuous Galerkin computation of the Maxwell eigenvalues on simplicial meshes. *Journal of Computational and Applied Mathematics*, 204(2):317 – 333, 2007. URL `http://dx.doi.org/DOI:10.1016/j.cam.2006.01.042`.

K. Burke, B. Steinberger, T. H. Torsvik, and M. A. Smethurst. Plume generation zones at the margins of large low shear velocity provinces on the core-mantle boundary. *Earth and Planetary Science Letters*, 265:49–60, 2008.

J. Cahouet and J.-P. Chabard. Some fast 3D finite element solvers for the generalized Stokes problem. *International Journal for Numerical Methods in Fluids*, 8(8):869–895, 1988.

C. Canuto, M. Y. Hussaini, A. Quarteroni, and T. A. Zang. *Spectral Methods*. Springer-Verlag, 2007. URL `http://dx.doi.org/10.1007/978-3-540-30728-0`.

C. A. P. Castigliano. *Théorie de l'équilibre des systèmes élastiques et ses applications*. A.F. Negro ed., Torino, 1879.

J. R. Cebral, M. A. Castro, J. E. Burgess, R. S. Pergolizzi, M. J. Sheridan, and C. M. Putman. Characterization of cerebral aneurysms for assessing risk of rupture by using patient-specific computational hemodynamics models. *American Journal of Neuroradiology*, 26(10):2550–2559, 2005. URL `http://www.ajnr.org/cgi/content/abstract/26/10/2550`.

O. Čertík, F. Seoane, P. Peterson, et al. sympy: Python library for symbolic mathematics, 2009. URL `http://sympy.org`.

CGAL. Software package. URL `http://www.cgal.org`.

D. Chapelle and K.-J. Bathe. The inf-sup test. *Comput. & Structures*, 47(4-5):537–545, 1993.

G. Chavent and P. Lemmonier. Identification de la non–linearite' d'une equation parabolique quasi–lineare. *Applied Math. and Opt.*, 26:121–162, 1974.

Y. Chen and P. L.-F. Liu. Modified Boussinesq equations and associated parabolic models for water wave propagation. *J. Fluid Mech.*, 288:351–381, 1994.

J. Chessa, P. Smolinski, and T. Belytschko. The extended finite element method (XFEM) for solidification problems. *International Journal for Numerical Methods in Engineering*, 53(8):1959–1977, 2002.

A. J. Chorin. Numerical solution of the Navier-Stokes equations. *Math. Comp.*, 22:745–762, 1968.

I. Christie, D. Griffiths, A. R. Mitchell, and O. C. Zienkiewicz. Finite element methods for second order differential equations with significant first derivatives. *International Journal for Numerical Methods in Engineering*, 10(6):1389–1396, 1976.

V. V. Chudanov, A. G. Popkov, A. G. Churbanov, P. N. Vabishchevich, and M. M. Makarov. Operator-splitting schemes for the streamfunction-vorticity formulation. *Comput. Fluids*, v24(7):771–786, 2007.

P. G. Ciarlet. Lectures on the finite element method. *Lectures on Mathematics and Physics, Tata Institute of Fundamental Research, Bombay*, 49, 1975.

P. G. Ciarlet. *Numerical Analysis of the Finite Element Method*. Les Presses de l'Universite de Montreal, 1976.

P. G. Ciarlet. *The Finite Element Method for Elliptic Problems*, volume 40 of *Classics in Applied Mathematics*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 2002.

P. G. Ciarlet and P.-A. Raviart. General Lagrange and Hermite interpolation in $\mathbf{R}^n$ with applications to finite element methods. *Arch. Rational Mech. Anal.*, 46:177–199, 1972.

R. Cimrman et al. SfePy: Simple Finite Elements in Python, 2008. URL `http://sfepy.org`.

G. Compère, J.-F. Remacle, J. Jansson, and J. Hoffman. Transient mesh adaptivity applied to large domain deformations. *To appear in Int. J. Numer. Meth. Eng.*, 2009.

COMSOL. Comsol multiphysics, 2009. URL `http://www.comsol.com`.

M. Costabel and M. Dauge. Weighted Regularization of Maxwell Equations in Polyhedral Domains. *Numerische Mathematik*, 93(2):239–277, 2002. ISSN 0029-599X.

R. Courant. Variational methods for the solution of problems of equilibrium and vibrations. *Bull. Amer. Math. Soc.*, pages 1–23, 1943.

M. Crouzeix and R. S. Falk. Nonconforming finite elements for the Stokes problem. *Mathematics of Computation*, 52:437–456, 1989. URL `http://dx.doi.org/10.2307/2008475`.

M. Crouzeix and P. A. Raviart. Conforming and non-conforming finite element methods for solving the stationary Stokes equations. *RAIRO Anal. Numer.*, 7:33–76, 1973.

Cython. Software package. URL `http://cython.org`.

D. B. Davidson. *Computational Electromagnetics for RF and Microwave Engineers*. Cambridge University Press, Cambridge, second edition, 2011. ISBN 9780521518918.

P. F. Davies, A. Remuzzi, E. J. Gordon, C. F. Dewey, and M. Gimbrone. Turbulent fluid shear stress induces vascular endothelial cell turnover in vitro. *Proc. Natl. Acad. Sci. USA*, 83:2114–2117, 1986.

T. A. Davis. Algorithm 832: Umfpack v4.3 — an unsymmetric-pattern multifrontal method. *ACM Transactions on Mathematical Software*, 30(2):196–199, 2004. URL `http://dx.doi.org/10.1145/992200.992206`.

DDD. Software package. URL `http://www.gnu.org/software/ddd`.

R. V. de Abreu, N. Jansson, and J. Hoffman. Adaptive computation of aeroacoustic sources for a 4-wheel rudimentary landing gear benchmark problem. In *Proc. Workshop on Benchmark Problems Airframe Noise Computation (BANC-I)*, 2010.

D. A. di Pietro, S. L. Forte, and N. Parolini. Mass preserving finite element implementations of the level set method. *Applied Numerical Mathematics*, 56:1179–1195, 2006.

B. M. Dillon and J. P. Webb. A comparison of formulations for the vector finite element analysis of waveguides. *IEEE Trans. Microwave Theory Tech.*, 42(2):308–316, February 1994.

Distutils. distutils âĂŤ Building and installing Python modules. URL `http://docs.python.org/library/distutils.html`.

V. Domínguez and F.-J. Sayas. Algorithm 884: A simple Matlab implementation of the Argyris element. *ACM Transactions on Mathematical Software*, 35(2):16:1–16:11, July 2008. URL `http://dx.doi.org/10.1145/1377612.1377620`.

J. Donea and A. Huerta. *Finite Element Methods for Flow Problems*. Wiley, 2003.

J. Donea, S. Giuliani, and J. Halleux. An arbitrary Lagrangian–Eulerian finite element method for transient dynamic fluid–structure interactions. *Computer Methods in Applied Mechanics and Engineering*, 33(1-3):689–723, 1982.

J. Donea, A. Huerta, J. P. Ponthot, and A. Rodrıguez-Ferran. Arbitrary Lagrangian–Eulerian methods. *Encyclopedia of Computational Mechanics*, 1:1–25, 2004.

W. Dörfler. A convergent adaptive algorithm for Poisson's equation. *SIAM Journal on Numerical Analysis*, 33(3):1106–1124, 1996.

M. Dubiner. Spectral methods on triangles and other domains. *Journal of Scientific Computing*, 6(4): 345–390, 1991.

P. Dular and C. Geuzaine. *GetDP Reference Manual*, 2005.

D. A. Dunavant. High degree efficient symmetrical Gaussian quadrature rules for the triangle. *International Journal for Numerical Methods in Engineering*, 21(6):1129–1148, 1985. ISSN 1097-0207.

T. Dunne. An Eulerian approach to fluid–structure interaction and goal-oriented mesh adaptation. *International Journal for Numerical Methods in Fluids*, 51(9-10):1017–1039, 2006.

T. Dunne. *Adaptive Finite Element Approximation of Fluid–Structure Interaction Based on Eulerian and Arbitrary Lagrangian-Eulerian Variational Formulations*. PhD thesis, Ruprecht-Karls-UniversitÃd't, Heidelberg, 2007.

T. Dunne and R. Rannacher. Adaptive finite element approximation of fluid–structure interaction based on an Eulerian variational formulation. *Lecture Notes In Computational Science And Engineering*, 53:110, 2006.

D. Dutykh and F. Dias. Dissipative Boussinesq equations. *Comptes Rendus Mecanique*, 335:559, 2007. URL `doi:10.1016/j.crme.2007.08.003`.

S. V. Eden, W. J. Meurer, B. N. Sanchez, L. D. Lisabeth, M. A. Smith, D. L. Brown, and L. B. Morgenstern. Gender and ethnic differences in subarachnoid hemorrhage. *Neurology*, 71:731–735, 2008.

H. C. Elman, D. J. Silvester, and A. J. Wathen. *Finite Elements and Fast Iterative Solvers: With Applications in Incompressible Fluid Dynamics*. Numerical Mathematics and Scientific Computation. Oxford University Press, Oxford, 2005.

M. S. Engelman, R. L. Sani, P. M. Gresho, and M. Bercovier. Consistent vs. reduced integration penalty methods for incompressible media using several old and new elements. *International Journal for Numerical Methods in Fluids*, 2(1):25–42, 1982. ISSN 1097-0363. doi: 10.1002/fld.1650020103. URL `http://dx.doi.org/10.1002/fld.1650020103`.

K. Eriksson and C. Johnson. Adaptive finite element methods for parabolic problems III: Time steps variable in space. *in preparation*.

K. Eriksson and C. Johnson. Adaptive finite element methods for parabolic problems I: A linear model problem. *SIAM J. Numer. Anal.*, 28, No. 1:43–77, 1991.

K. Eriksson and C. Johnson. Adaptive finite element methods for parabolic problems II: Optimal error estimates in $l_\infty l_2$ and $l_\infty l_\infty$. *SIAM Journal on Numerical Analysis*, 32:706, 1995a.

K. Eriksson and C. Johnson. Adaptive finite element methods for parabolic problems IV: Nonlinear problems. *SIAM Journal on Numerical Analysis*, pages 1729–1749, 1995b.

K. Eriksson and C. Johnson. Adaptive finite element methods for parabolic problems V: Long-time integration. *SIAM J. Numer. Anal.*, 32:1750–1763, 1995c.

K. Eriksson, D. Estep, P. Hansbo, and C. Johnson. Introduction to adaptive methods for differential equations. *Acta Numerica*, 4:105–158, 1995.

K. Eriksson, D. Estep, P. Hansbo, and C. Johnson. *Computational Differential Equations*. Cambridge University Press, 1996.

K. Eriksson, C. Johnson, and S. Larsson. Adaptive finite element methods for parabolic problems VI: Analytic semigroups. *SIAM J. Numer. Anal.*, 35:1315–1325, 1998.

A. Ern and J.-L. Guermond. *Theory and Practice of Finite Elements*, volume 159 of *Applied Mathematical Sciences*. Springer-Verlag, 2004.

C. R. Ethier and D. A. Steinmann. Exact fully 3D Navier–Stokes solution for benchmarking. *Internat. J. Numer. Methods Fluids*, 19:369 – 375, 1994.

A. J. Evans, F. Iwai, T. A. Grist, H. D. Sostman, L. W. Hedlund, C. E. Spritzer, R. Negro-Vilar, C. A. Beam, and N. J. Pelc. Magnetic resonance imaging of blood flow with a phase subtraction technique. in vitro and in vivo validation. *Investigative Radiology*, 28(2), 1993.

F2PY. Software package. URL `http://cens.ioc.ee/projects/f2py2e`.

R. D. Falgout and U. M. Yang. Hypre: A library of high performance preconditioners. In *Proceedings of the International Conference on Computational Science-Part III*, pages 632–641. Springer-Verlag, 2002.

M. Farhloul and M. Fortin. Dual hybrid methods for the elasticity and the Stokes problems: a unified approach. *Numer. Math.*, 76(4):419–440, 1997.

V. Feigin. Stroke epidemiology in the developing world. *The Lancet*, 365(9478):2160–2161, 2005. URL `http://dx.doi.org/10.1016/S0140-6736(05)66755-4`.

P. Fernandes and M. Raffetto. Characterization of spurious-free finite element methods in electromagnetics. *COMPEL - The International Journal for Computation and Mathematics in Electrical and Electronic Engineering*, 21:147–164, 2002. URL http://dx.doi.org/10.1108/03321640210410814.

M. D. Ford, N. Alperin, S. H. Lee, D. W. Holdsworth, and D. A. Steinman. Characterization of volumetric flow rate waveforms in the normal internal cartoid and vertebral arteries. *Physiological Measurement*, 26:477–488, 2005.

L. Formaggia and F. Nobile. A stability analysis for the arbitrary Lagrangian Eulerian formulation with finite elements. *East-West Journal of Numerical Mathematics*, 7:105–132, 1999.

S. A. Forth, M. Tadjouddine, J. D. Pryce, and J. K. Reid. Jacobian code generated by source transformation and vertex elimination can be as efficient as hand-coding. *ACM Trans. Math. Softw.*, 30(3): 266–299, 2004. URL http://dx.doi.org/10.1145/1024074.1024076.

C. Franzini-Armstrong, F. Protasi, and V. Ramesh. Shape, size, and distribution of Ca^{2+} release units and couplons in skeletal and cardiac muscles. *Biophys. J.*, 77(3):1528–1539, Sep 1999. URL http://www.biophysj.org/cgi/content/full/77/3/1528.

Y. C. Fung. *Biodynamics: Circulation*. Springer-Verlag, 1984.

Y. C. Fung. *Biomechanics: Mechanical Properties of Living Tissues*. Springer-Verlag, 1993.

B. G. Galerkin. Series solution of some problems in elastic equilibrium of rods and plates. *Vestnik inzhenerov i tekhnikov*, 19:897–908, 1915.

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

GDB. Software package. URL http://www.gnu.org/software/gdb.

M. W. Gee, C. M. Siefert, J. J. Hu, R. S. Tuminaro, and M. G. Sala. ML 5.0 smoothed aggregation user's guide. Technical Report SAND2006-2649, Sandia National Laboratories, 2006.

W. J. German and S. P. W. Black. Cervical ligation for internal carotid aneurysms. *Journal of Neurosurgery*, 23(6):572–577, 1965. URL http://dx.doi.org/10.3171/jns.1965.23.6.0572.

C. Geuzaine and J.-F. Remacle. Gmsh - a 3d mesh generator.

R. Giering and T. Kaminski. Recipes for adjoint code construction. *ACM Trans. Math. Softw.*, 24(4): 437–474, 1998. URL http://dx.doi.org/10.1145/293686.293695.

M. Giles and E. Süli. Adjoint methods for PDEs: a posteriori error analysis and postprocessing by duality. *Acta Numer.*, 11:145–236, 2002.

D. T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *J. Phys. Chem.*, 81(25): 2340–2361, 1977. URL http://dx.doi.org/10.1021/j100540a008.

V. Girault and P.-A. Raviart. *Finite element methods for Navier-Stokes equations*, volume 5 of *Springer Series in Computational Mathematics*. Springer-Verlag, Berlin, 1986.

M. S. Gockenbach. *Understanding and Implementing the Finite Element Method*. Society for Industrial and Applied Mathematics, 2006.

K. Goda. A multistep technique with implicit difference schemes for calculating two- or three-dimensional cavity flows. *Journal of Computational Physics*, 30(1):76–95, 1979. URL http://dx.doi.org/10.1016/0021-9991(79)90088-3.

G. H. Golub and C. F. Van Loan. *Matrix Computations*. Johns Hopkins Studies in the Mathematical Sciences. Johns Hopkins University Press, Baltimore, third edition, 1996.

D. C. Grahame. The electrical double layer and the theory of electrocapillarity. *Chem Rev*, 41(3): 441–501, Dec 1947.

T. Grätsch and K. Bathe. Goal-oriented error estimation in the analysis of fluid flows with structural interactions. *Computer Methods in Applied Mechanics and Engineering*, 195(41-43):5673–5684, 2006.

H. Gray. *Anatomy, Descriptive and Surgical*. Lea Brothers, 1897.

J. L. Greenstein and R. L. Winslow. An integrative model of the cardiac ventricular myocyte incorporating local control of Ca^{2+} release. *Biophys. J.*, 83(6):2918–2945, Dec 2002. URL http://dx.doi.org/10.1016/S0006-3495(02)75301-0.

A. Griewank. On automatic differentiation. In M. Iri and K. Tanabe, editors, *Mathematical Programming: Recent Developments and Applications*, pages 83–108. Kluwer Academic Publishers, 1989.

F. F. Grinstein, L. G. Margolin, and W. J. Rider. *Implicit Large Eddy Simulation: Computing Turbulent Fluid Dynamics*. Cambridge University Press, New York, 2007.

J. L. Guermond and J. Shen. A new class of truly consistent splitting schemes for incompressible flows. *J. Comput. Phys.*, 192(1):262–276, 2003. URL http://dx.doi.org/10.1016/j.jcp.2003.07.009.

J. L. Guermond, P. Minev, and J. Shen. An overview of projection methods for incompressible flows. *Comput. Methods Appl. Mech. Engrg*, 41:112–134, 2006.

A. Guia, M. D. Stern, E. G. Lakatta, and I. R. Josephson. Ion concentration-dependence of rat cardiac unitary l-type calcium channel conductance. *Biophys J*, 80(6):2742–2750, 2001. URL http://dx.doi.org/10.1016/S0006-3495(01)76242-X.

S. Gupta, M. Soellinger, P. Boesiger, D. Pulikako, and V. Kurtcuoglu. Three-dimensional computational modeling of subject -specific cerebrospinal fluid flow in the subarachnoid space. *J. Biomech. Eng.*, 131, 2009.

M. E. Gurtin. *An Introduction to Continuum Mechanics*. Academic Press, 1981.

W. Hackbusch. *Iterative Solution of Large Sparse Systems of Equations*. Springer-Verlag, 1994.

E. Hairer and G. Wanner. *Solving Ordinary Differential Equations I — Nonstiff Problems*. Springer Series in Computational Mathematics, vol. 8, 1991a.

E. Hairer and G. Wanner. *Solving Ordinary Differential Equations. II*, volume 14 of *Springer Series in Computational Mathematics*. Springer-Verlag, Berlin, 1991b.

P. Hansbo. A Crank–Nicolson type space-time finite element method for computing on moving meshes. *Journal of Computational Physics*, 159:274–289, 2000.

P. Hansbo and M. G. Larson. Discontinuous Galerkin and the Crouzeix–Raviart element: application to elasticity. *Math. Model. Numer. Anal.*, 37:63–72, 2003. URL http://dx.doi.org/10.1051/m2an:2003020.

T. Hansen and G. Wollman. US secure hash algorithm 1 (SHA1). URL `http://www.apps.ietf.org/rfc/rfc3174.html`.

U. Hansen and A. Ebel. Time-dependent thermal convection – a possible explanation for a multiscale flow in the earth's mantle. *Geophysical Journal*, 94:181–191, 1988.

U. Hansen and D. A. Yuen. Numerical simulation of thermal-chemical instabilities at the core-mantle boundary. *Nature*, 334:237–240, 1988.

V. Haughton, F. R. Korosec, J. E. Medow, M. T. Dolar, and B. J. Iskandar. Peak systolic and diastolic CSF velocity in the foramen magnum in adult patients with Chiari I malformations and in normal control participants. *Am. J. Neuroradiol.*, 24:169–176, 2003.

J. D. Heiss, N. Patronas, H. L. DeVroom, T. Shawker, R. Ennis, W. Kammerer, A. Eidsath, T. Talbot, J. Morris, E. Eskioglu, and E. H. Oldfield. Elucidating the pathophysiology of syringomyelia. *J. Neurosurg.*, 91:532–562, 1999.

S. Hentschel, K.-A. Mardal, A. E. Løvgren, S. Linge, and V. Haughton. Characterization of cyclical CSF flow in the foramen magnum and upper cervical spinal canal with MR flow imaging and computational fluid dynamics. *American Journal of Neuroradiology*, 31(6):997 – 1002, 2010.

J. Hermansson and P. Hansbo. A variable diffusion method for mesh smoothing. *Communications in Numerical Methods in Engineering*, 19(11):897–908, 2003.

V. Hernandez, J. E. Roman, and V. Vidal. SLEPc: A scalable and flexible toolkit for the solution of eigenvalue problems. *ACM Transactions on Mathematical Software*, 31(3):351–362, Sept. 2005.

V. Hernandez, J. E. Roman, E. Romero, A. Tomas, and V. Vidal. SLEPc users manual. Technical Report DSIC-II/24/02 - Revision 3.0.0, D. Sistemas Informáticos y Computación, Universidad Politécnica de Valencia, 2009.

M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley. An overview of the Trilinos project. *ACM Trans. Math. Softw.*, 31(3): 397–423, 2005. URL `http://dx.doi.org/10.1145/1089014.1089021`.

M. R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *J. Res. Nat. Bur. Stand*, 49(6):409–436, 1952.

M. L. Hetland. *Practical Python*. APress, 2002.

H. M. Hilber, T. J. R. Hughes, and R. L. Taylor. Improved numerical dissipation for time integration algorithms in structural dynamics. *Earthquake Engineering and Structural Dynamics*, 5(3):283–292, 1977.

B. Hille. *Ion Channels of Excitable Membranes*. Sinauer, Sunderland, MA, 2001.

R. Hiptmair. Multigrid method for $H(\mathrm{div})$ in three dimensions. *Electronic Transactions on Numerical Analysis*, 6:133–152, 1997.

R. Hiptmair. Multigrid method for Maxwell's equations. *SIAM Journal on Numerical Analysis*, 36(1): 204–225, 1999. ISSN 0036-1429.

R. Hiptmair. Operator preconditioning. *Computers & Mathematics with Applications*, 52(5):699–706, 2006.

J. Hoffman. Computation of mean drag for bluff body problems using adaptive DNS/LES. *SIAM J. Sci. Comput.*, 27(1):184–207, 2005.

J. Hoffman. Adaptive simulation of the turbulent flow past a sphere. *J. Fluid Mech.*, 568:77–88, 2006.

J. Hoffman. Efficient computation of mean drag for the subcritical flow past a circular cylinder using general Galerkin G2. *Int. J. Numer. Meth. Fluids*, 2009.

J. Hoffman and N. Jansson. A computational study of turbulent flow separation for a circular cylinder using skin friction boundary conditions. In *Proceedings of Workshop for Quality and Reliability of Large-Eddy Simulations II*, 2009.

J. Hoffman and C. Johnson. A new approach to computational turbulence modeling. *Comput. Methods Appl. Mech. Engrg.*, 195:2865–2880, 2006.

J. Hoffman and C. Johnson. *Computational Turbulent Incompressible Flow*. Springer, 2007. URL `http://dx.doi.org/10.1007/978-3-540-46533-1`.

J. Hoffman and C. Johnson. Resolution of d'Alembert's paradox. *J. Math. Fluid Mech.*, Online First December 2008, 2008.

J. Hoffman, J. Jansson, and M. Stöckli. Unified continuum modeling of fluid–structure interaction. *Mathematical Models and Methods in Applied Sciences*, 2011.

J. Hoffman, J. Jansson, and R. V. de Abreu. Adaptive modeling of turbulent flow with residual based turbulent kinetic energy dissipation. *Comput. Meth. Appl. Mech. Eng.*, 200(37-40):2758 – 2767, 2011a. ISSN 0045-7825.

E. Hofmann, M. Warmuth-Metz, M. Bendszus, and L. Solymosi. Phase-contrast MR imaging of the cervical CSF and spinal cord: volumetric motion analysis in patients with Chiari i malformation. *AJNR Am J Neuroradiol*, 21:151–158, 2000.

G. A. Holzapfel. *Nonlinear Solid Mechanics: A Continuum Approach for Engineering*. John Wiley & Sons, 2000.

J. Hron and S. Turek. A monolithic FEM solver for ALE formulation of fluid structure interaction with configurations for numerical benchmarking. In M. Papadrakakis, E. Onate, and B. Schrefler, editors, *Computational Methods for Coupled Problems in Science and Engineering*, volume First Edition, page 148, 2005. Konferenzband 'First International Conference on Computational Methods for Coupled Problems in Science and Engineering' (Santorini, May 25th - 27th).

J. Hu, R. Tuminaro, P. Bochev, C. Garasi, and A. Robinson. Toward an h-independent algebraic multigrid method for Maxwell's equations. *SIAM Journal on Scientific Computing*, 27(5):1669–1688, 2006.

J. Huang, L. Guo, and Z. Shi. Vibration analysis of Kirchhoff plates by the Morley element method. *Journal of Computational and Applied Mathematics*, 213(1):14–34, 2008.

W. X. Huang and T. Itoh. Complex modes in lossless shielded microstrip lines. *IEEE Transactions on Microwave Theory and Techniques*, 36(1):163–165, 1988.

T. J. R. Hughes. *The Finite Element Method: Linear Static and Dynamic Finite Element Analysis*. Prentice-Hall, 1987.

T. J. R. Hughes, L. P. Franca, and M. Balestra. A new finite element formulation for computational fluid dynamics: V. Circumventing the Babuška-Brezzi condition: a stable Petrov-Galerkin formulation of the Stokes problem accommodating equal-order interpolations. *Computer Methods in Applied Mechanics and Engineering*, 59:85–99, 1986.

T. J. R. Hughes, J. A. Cottrell, and Y. Bazilevs. Isogeometric analysis: CAD, finite elements, NURBS, exact geometry and mesh refinement. *Computer Methods in Applied Mechanics and Engineering*, 194 (39–41):4135–4195, 2005.

J. D. Humphrey. *Cardiovascular Solid Mechanics*. Springer, 2001.

Hypre. Software package. URL `http://www.llnl.gov/CASC/hypre/`.

M. G. Imhof and A. K. Sharma. Seismostratigraphic inversion: Appraisal, ambiguity, and uncertainty. *Geophysics*, 72(4):R51–R65, 2007.

T. Ingebrigtsen, M. K. Morgan, K. Faulder, L. Ingebrigtsen, T. Sparr, and H. Schirmer. Bifurcation geometry and the presence of cerebral artery aneurysms. *J Neurosurg.*, 101:108–113, 2004. URL `http://dx.doi.org/10.3171/jns.2004.101.1.0108`.

A. Ismail-Zadeh and P. Tackley. *Computational Methods for Geodynamics*. Cambridge University Press, 2010.

T. Itoh, P. Silvester, and G. Pelosi. *Finite Element Software for Microwave Engineering*. John Wiley & Sons, Inc. New York, NY, USA, 1996. ISBN 0471126365.

M. S. Jafri, J. J. Rice, and R. L. Winslow. Cardiac Ca^{2+} dynamics: the roles of ryanodine receptor adaptation and sarcoplasmic reticulum load. *Biophys. J.*, 74(3):1149–68, Mar 1998. URL `http://www.biophysj.org/cgi/content/full/74/3/1149`.

P. Jäger, P. Steinmann, and E. Kuhl. Modeling three-dimensional crack propagation - a comparison of crack path tracking strategies. *International Journal for Numerical Methods in Engineering*, 76(9): 1328–1352, 2008.

J. Jansson. Performance optimization of unicorn. Technical report, 2009.

N. Jansson. *High performance adaptive finite element methods for turbulent fluid flow*. Licentiate thesis, Royal Institute of Technology, School of Computer Science and Engineering, 2011. TRITA-CSC-A 2011:02.

N. Jansson, J. Hoffman, and J. Jansson. Performance of Dolfin and Unicorn on modern high-performance distributed memory architectures. Technical Report KTH-CTL-4012, Computational Technology Laboratory, 2010. `http://www.publ.kth.se/trita/ctl-4/012/`.

G. T. Jarvis and J. P. Lowman. Survival times of subducted slab remnants in numerical models of mantle flow. *Earth and Planetary Science Letters*, 260(1–2):23–36, 2007. URL `http://dx.doi.org/10.1016/j.epsl.2007.05.009`.

J. Jiang, K. Johnson, K. Valen-Sendstad, K.-A. Mardal, O. Wieben, and C. Strother. Flow characteristics in a canine aneurysm model: A comparison of 4-D accelerated phase-contrast MR measurements and computational fluid dynamics simulations. 2011. Accepted in Medical Physics.

J. Jin. *The Finite Element Method in Electromagnetics*. John Wiley & Sons, Inc., New York, 2nd edition, 2002. ISBN 0471438189.

V. John and P. Knobloch. On spurious oscillations at layers diminishing (SOLD) methods for convection-diffusion equations: Part I – A review. *Computer Methods in Applied Mechanics and Engineering*, 196 (17-20):2197 – 2215, 2007. URL http://dx.doi.org/10.1016/j.cma.2006.11.013.

R. S. Johnson. *A Modern Introduction to the Mathematical Theory of Water Waves*. Cambridge Texts in Applied Mathematics. Cabridge University Press, Cambridge, 1997.

E. Jones, T. Oliphant, P. Peterson, et al. SciPy: Open source scientific tools for Python, 2009. URL http://scipy.org.

J. Karczmarczuk. Functional differentiation of computer programs. *Higher-Order and Symbolic Computation*, 14(1):35–57, 2001. URL http://dx.doi.org/10.1023/A:1011501232197.

G. E. Karniadakis and S. J. Sherwin. *Spectral/hp Element Methods for Computational Fluid Dynamics*. Numerical Mathematics and Scientific Computation. Oxford University Press, Oxford, second edition, 2005.

M. Keegan, D. Ridzal, and P. Bochev. Sparse-grid integration in finite element spaces. *CSRI Summer Proceedings*, page 32, 2008.

B. L. N. Kennett and H.-P. Bunge. *Geophysical Continua*. Cambridge University Press, 2008.

J. T. Kirby. Funwave homepage, 1998. URL http://chinacat.coastal.udel.edu/programs/funwave/funwave.html.

R. C. Kirby. Algorithm 839: FIAT, a new paradigm for computing finite element basis functions. *ACM Trans. Math. Softw.*, 30:502–516, 2004. URL http://dx.doi.org/10.1145/1039813.1039820.

R. C. Kirby. *FIAT*, 2006a. URL http://fenicsproject.org.

R. C. Kirby. Optimizing FIAT with level 3 BLAS. *ACM Trans. Math. Softw.*, 32:223–235, 2006b. URL http://dx.doi.org/10.1145/1141885.1141889.

R. C. Kirby. Singularity-free evaluation of collapsed-coordinate orthogonal polynomials. *ACM Trans. Math. Softw.*, 37:5:1–5:16, 2010a. URL http://dx.doi.org/10.1145/1644001.1644006.

R. C. Kirby. From functional analysis to iterative methods. *SIAM Review*, 52:269–293, 2010b. URL http://dx.doi.org/10.1137/070706914.

R. C. Kirby. Fast simplicial finite element algorithms using Bernstein polynomials. *Numerische Mathematik*, 117(4):631–652, 2011.

R. C. Kirby and A. Logg. A compiler for variational forms. *ACM Trans. Math. Softw.*, 32:417–444, 2006. URL http://dx.doi.org/10.1145/1163641.1163644.

R. C. Kirby and A. Logg. Efficient compilation of a class of variational forms. *ACM Transactions on Mathematical Software*, 33(3), 2007.

R. C. Kirby and A. Logg. Benchmarking domain-specific compiler optimizations for variational forms. *ACM Transactions on Mathematical Software*, 35(2):1–18, 2008. URL http://dx.doi.org/10.1145/1377612.1377614.

R. C. Kirby and L. R. Scott. Geometric optimization of the evaluation of finite element matrices. *SIAM J. Sci. Comput.*, 29:827–841, 2007. URL http://dx.doi.org/10.1137/060660722.

R. C. Kirby, M. G. Knepley, A. Logg, and L. R. Scott. Optimizing the evaluation of finite element matrices. *SIAM J. Sci. Comput.*, 27(6):741–758, 2005. ISSN 1064-8275.

R. C. Kirby, A. Logg, L. R. Scott, and A. R. Terrel. Topological optimization of the evaluation of finite element matrices. *SIAM Journal on Scientific Computing*, 28(1):224–240, 2006. URL `http://dx.doi.org/10.1137/050635547`.

J. Kiusalaas. *Numerical Methods in Engineering with Python*. Cambridge University Press, second edition, 2009.

M. G. Knepley and D. A. Karpeev. Mesh algorithms for PDE with Sieve I: Mesh distribution. *Scientific Programming*, 17(3):215–230, 2009.

X. Koh, B. Srinivasan, H. S. Ching, and A. Levchenko. A 3D Monte Carlo analysis of the role of dyadic space geometry in spark generation. *Biophys. J.*, 90(6):1999–2014, 2006. URL `http://dx.doi.org/10.1529/biophysj.105.065466`.

G. L. Kongable, G. Lanzino, T. P. Germanson, L. L. Truskowski, W. M. Alves, J. C. Torner, and N. F. Kassell. Gender-related differences in aneurysmal subarachnoid hemorrhage. *Journal of Neurosurgery*, 84:43–48, 1996.

J. Korelc. Automatic generation of finite-element code by simultaneous optimization of expressions. *Theoretical Computer Science*, 187(1-2):231–248, 1997. URL `http://dx.doi.org/10.1016/S0304-3975(97)00067-4`.

J. Korelc. Multi-language and multi-environment generation of nonlinear finite element codes. *Engineering with Computers*, 18(4):312–327, 2002. URL `http://dx.doi.org/10.1007/s003660200028`.

J. Krejza, P. Szydlik, D. S. Liebeskind, J. Kochanowicz, O. Bronov, Z. Mariak, and E. R. Melhem. Age and sex variability and normal reference values for the vMCA/vICA index. *American Journal of Neuroradiology*, 26(4):730–735, 2005. URL `http://www.ajnr.org/cgi/content/abstract/26/4/730`.

A. Kuchling. Python's dictionary implementation: Being all things to all people. In A. Oram and G. Wilson, editors, *Beautiful Code*, chapter 18, pages 293–302. O'Reilly, 2007. URL `http://portal.acm.org/citation.cfm?id=1407867`.

J. D. Lambert. *Numerical Methods for Ordinary Differential Systems*. John Wiley & Sons Ltd., Chichester, 1991.

L. Landweber. An iteration formula for Fredholm integral equations of the first kind. *Amer. J. Math.*, 73:615–624, 1951.

G. A. Langer and A. Peskoff. Calcium concentration and movement in the diadic cleft space of the cardiac ventricular cell. *Biophys. J.*, 70(3):1169–1182, 1996. URL `http://dx.doi.org/10.1016/S0006-3495(96)79677-7`.

M. Langner, D. Cafiso, S. Marcelja, and S. McLaughlin. Electrostatics of phosphoinositide bilayer membranes. Theoretical and experimental results. *Biophys. J.*, 57(2):335–349, 1990. URL `http://dx.doi.org/10.1016/S0006-3495(90)82535-2`.

H. P. Langtangen. *Python Scripting for Computational Science*, volume 3 of *Texts in Computational Science and Engineering*. Springer, third edition, 2008. URL `http://dx.doi.org/10.1007/978-3-540-73916-6`.

H. P. Langtangen. *A Primer on Scientific Programming with Python*, volume 6 of *Texts in Computational Science and Engineering*. Springer, second edition, 2011. URL http://dx.doi.org/10.1007/978-3-642-18366-9.

H. P. Langtangen and K.-A. Mardal. Using Diffpack from Python scripts. In H. P. Langtangen and A. Tveito, editors, *Advanced Topics in Computational Partial Differential Equations - Numerical Methods and Diffpack Programming*, Lecture Notes in Computational Science and Engineering, pages 321–360. Springer-Verlag, 2003.

H. P. Langtangen and G. Pedersen. Computational models for weakly dispersive nonlinear water waves. *Computer Methods in Applied Mechanics and Engineering*, 160(3-4):337 – 358, 1998. URL http://dx.doi.org/10.1016/S0045-7825(98)00293-X.

P. Lascaux and P. Lesaint. Some nonconforming finite elements for the plate bending problem. *Rev. Française Automat. Informat. Recherche Operationnelle RAIRO Analyse Numérique*, 9(R-1):9–53, 1975.

P. D. Lax and A. N. Milgram. Parabolic equations. *Annals of Mathematics Studies*, 33:167–190, 1954.

J.-F. Lee. Finite element analysis of lossy dielectric waveguides. *IEEE Transactions on Microwave Theory Techniques*, 42(5):1025–1031, June 1994.

J.-F. Lee, D.-K. Sun, and Z. J. Cendes. Full-wave analysis of dielectric waveguides using tangential vector finite elements. *IEEE Trans. Microwave Theory Tech.*, 39(8):1262–1271, August 1991.

A. Lenardic and W. M. Kaula. A numerical treatment of geodynamic viscous flow problems involving the advection of material interfaces. *J. Geophys. Res.*, 98(B5):8243–8260, 1993.

X. Li. An overview of SuperLU: Algorithms, implementation, and user interface. *ACM Transactions on Mathematical Software*, 31(3):302–325, 2005.

B. Q. Lin. *Discontinuous Finite Elements in Fluid Dynamics and Heat Transfer*. Computational Fluid and Solid Mechanics. Springer-Verlag, 2006. URL http://dx.doi.org/10.1007/1-84628-205-5.

H. Lindekleiv, K. Valen-Sendstad, M. F. Morgan, K.-A. Mardal, K. Faulder, J. Magnus, K. Waterloo, B. Romner, and T. Ingebrigtsen. Sex differences in intracranial arterial bifurcations. *Gender Medicine*, 7(2):149–155, 2010. URL http://dx.doi.org/10.1016/j.genm.2010.03.003.

S. Linge, V. Haughton, A. E. Løvgren, K.-A. Mardal, and H. P. Langtangen. CSF flow dynamics at the craniovertebral junction studied with an idealized model of the subarachnoid space and computational flow analysis. *American Journal of Neuroradiology*, 31(1):185–192, 2010. URL http://dx.doi.org/10.3174/ajnr.A1766.

S. Linge, V. Haughton, A. E. Løvgren, K.-A. Mardal, A. Helgeland, and H. P. Langtangen. Effect of tonsilar herniation on cyclic CSF flow studied with computational flow analysis. *American Journal of Neuroradiology*, 32(8):1474–1481, 2011. URL http://dx.doi.org/10.3174/ajnr.A2496.

A. Logg. Multi-adaptive time-integration. *Applied Numerical Mathematics*, 48(3–4):339–354, 2004. URL http://dx.doi.org/10.1016/j.apnum.2003.11.004.

A. Logg. Automating the Finite Element Method. *Arch. Comput. Methods Eng.*, 2(14):93–138, 2007.

A. Logg. Efficient representation of computational meshes. *International Journal of Computational Science and Engineering*, 4(4):283–295, 2009.

A. Logg and K.-A. Mardal. Exterior, 2009. URL http://fenicsproject.org.

A. Logg and G. N. Wells. DOLFIN: Automated finite element computing. *ACM Transactions on Mathematical Software*, 32(2):1–28, 2010. URL `http://dx.doi.org/10.1145/1731022.1731030`.

K. Long. Sundance, a rapid prototyping tool for parallel PDE-constrained optimization. In *Large-Scale PDE-Constrained Optimization*, Lecture Notes in Computational Science and Engineering. Springer-Verlag, 2003.

K. Long. Sundance 2.0 tutorial. Technical Report TR–2004–09, Sandia National Laboratories, 2004a.

K. Long. Efficient discretization and differentiation of partial differential equations through automatic functional differentiation, 2004b. http://www.autodiff.org/ad04/abstracts/Long.pdf.

W. T. Longstreth, L. M. Nelson, and G. T. D. Koepsell andvan Belle. Subarachnoid hemorrhage and hormonal factors in women: A population-based case-control study. *Ann Intern Med*, 121(3):168–173, 1994.

N. Lopes. *Relatório do Curso de Formação avançada: Alguns Modelos de Propagação de Ondas Marítimas*. PhD report, Departamento de Matemática, Faculdade de Ciências da Universidade de Lisboa, 2007.

N. D. Lopes, P. J. S. Pereira, and L. Trabucho. A numerical analysis of a class of generalized Boussinesq-type equations using continuous/discontinuous FEM. *International Journal for Numerical Methods in Fluids*, 2011 URL `http://dx.doi.org/10.1002/fld.2631`.

E. J. López, N. M. Nigro, and M. A. Storti. Simultaneous untangling and smoothing of moving grids. *International Journal for Numerical Methods in Engineering*, 76(7):994–1019, 2008.

F. Loth, M. A. Yardimci, and N. Alperin. Hydrodynamic modeling of cerebrospinal fluid motion within the spinal cavity. *J. Biomech. Eng.*, 123:71–79, 2001.

J. Lotz, C. Meier, A. Leppert, and M. Galanski. Cardiovascular flow measurement with phase-contrast MR imaging: Basic facts and implementation. *Radiographics*, 22(3):651–671, 2002.

F. Løvholt and G. Pedersen. Instabilities of Boussinesq models in non-uniform depth. *International Journal for Numerical Methods in Fluids*, 61(6):606–637, 2009. ISSN 0271-2091.

J. Lubliner. *Plasticity Theory*. Dover Publications, 2008.

M. Lutz. *Programming Python*. O'Reilly, third edition, 2006.

M. Lutz. *Learning Python*. O'Reilly, third edition, 2007.

P. J. Lynett and P. L.-F. Liu. Coulwave model page, 2004. URL `https://ceprofs.civil.tamu.edu/plynett/COULWAVE/default.htm`.

Y. Maday, A. Patera, and E. Rønquist. The $P^N \times P^{N-2}$ method for the approximation of the Stokes problem. Technical report, Department of Mechanical Engineering, Massachusetts Institute of Technology, 1992.

P. A. Madsen and Y. Agnon. Accuracy and convergence of velocity formulations for water waves in the framework of Boussinesq theory. *J. Fluid Mech.*, 477:285–319, 2003.

P. A. Madsen, R. Murray, and O. R. Sørensen. A new form of the Boussinesq equations with improved linear dispersion characteristics. *Coastal Engineering*, 15(4):371 – 388, 1991. URL `http://dx.doi.org/10.1016/0378-3839(91)90017-B`.

P. A. Madsen, H. B. Bingham, and H. A. Schäffer. Boussinesq-type formulations for fully nonlinear and extremely dispersive water waves: derivation and analysis. *Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, 459(2033):1075–1104, 2003. URL `http://dx.doi.org/10.1098/rspa.2002.1067`.

M. R. Malik, T. R. Zang, and M. Y. Hussaini. A spectral collocation method for the Navier–Stokes equations. Technical Report 172365, ICASE report no. 84-19., NASA, 1984.

D. S. Malkus. Eigenproblems associated with the discrete LBB condition for incompressible finite elements. *Internat. J. Engrg. Sci.*, 19(10):1299–1310, 1981.

D. S. Malkus. *The Finite Element Method*, chapter 4.II, pages 276 – 303. Dover, 2000.

T. A. Manteuffel. An incomplete factorization technique for positive definite linear systems. *Mathematics of Computation*, 34(150):473–497, 1980.

G. Marckmann and E. Verron. Comparison of hyperelastic models for rubber-like materials. *Rubber chemistry and technology*, 79(5):835–858, 2006.

K.-A. Mardal and R. Winther. Uniform preconditioners for the time dependent Stokes problem. *Numerische Mathematik*, 98(2):305–327, 2004. ISSN 0029-599X.

K.-A. Mardal and R. Winther. Preconditioning discretizations of systems of partial differential equations. *Numerical Linear Algebra with Applications*, 18(1):1–40, 2011. URL `http://dx.doi.org/10.1002/nla.716`.

K. A. Mardal, X.-C. Tai, and R. Winther. A robust finite element method for Darcy-Stokes flow. *SIAM J. Numer. Anal.*, 40(5):1605–1631, 2002.

A. Martelli. *Python in a Nutshell*. O'Reilly, second edition, 2006.

A. Martelli and D. Ascher. *Python Cookbook*. O'Reilly, second edition, 2005.

MayaVi2. Software package. URL `http://code.enthought.com/projects/mayavi/`.

B. W. McCormick. *Aerodynamics, Aeronautics and Flight Mechanics*. John Wiley and Sons, second edition, 1995.

S. G. McLaughlin, G. Szabo, and G. Eisenman. Divalent ions and the surface potential of charged phospholipid membranes. *J. Gen. Physiol.*, 58(6):667–687, Dec 1971. URL `http://www.jgp.org/cgi/reprint/58/6/667`.

A. K. McNamara and S. Zhong. Thermochemical structures beneath Africa and the Pacific Ocean. *Nature*, 437:1136–1139, 2005.

A. K. McNamara, E. J. Garnero, and S. Rost. Tracking deep mantle reservoirs with ultra-low velocity zones. *Earth and Planetry Science Letters*, 299:1–9, 2010.

M. K. McNutt. Superswells. *Rev. Geophys.*, 36:211–244, 1998.

MeshBuilder. Software package. URL `http://launchpad.net/meshbuilder`.

C. N. Mhurchu, C. Anderson, K. Jamrozik, G. Hankey, and D. Dunbabin. Hormonal factors and risk of aneurysmal subarachnoid hemorrhage: An international population-based, case-control study. *Stroke*, 32:606–612, 2001.

T. Milhorat and P. A. Bolognese. Tailored operative technique for chiari type i malformations using intraoperative color doppler ultrasonography. *Neurosurgery*, 53(4):899–906, 2003.

W. Ming and J. Xu. The Morley element for fourth order elliptic equations in any dimensions. *Numerische Mathematik*, 103(1):155–169, 2006.

N. Moës, J. Dolbow, and T. Belytschko. A finite element method for crack growth without remeshing. *International Journal for Numerical Methods in Engineering*, 46(1):231–150, 1999.

P. Monk. *Finite Element Methods for Maxwell's Equations*. Oxford University Press, Oxford, 2003. ISBN 0198508883.

N. L. Montague, H. L. Kellogg, and M. Manga. High Rayleigh number thermo-chemical models of a dense boundary layer in D''. *Geophys. Res. Lett.*, 25(13):2345–2348, 1998.

L. S. D. Morley. The triangular equilibrium element in the solution of plate bending problems. *Aero. Quart.*, 19:149–169, 1968.

L. S. D. Morley. The constant-moment plate-bending element. *The Journal of Strain Analysis for Engineering Design*, 6(1):20–24, 1971.

M. Mortensen, H. P. Langtangen, and J. Myre. CBC.RANS – a new flexible, programmable software framework for computational fluid dynamics. In H. I. Andersson and B. Skallerud, editors, *Sixth National Conference on Computational Mechanics (MekIT'11)*. Tapir, 2011.

M. Mortensen, H. P. Langtangen, and G. N. Wells. A FEniCS-based programming framework for modeling turbulent flow by the Reynolds-averaged Navier–Stokes equations. *Advances in Water Resources*, 34(9):1082–1101, 2011. URL `http://dx.doi.org/10.1016/j.advwatres.2011.02.013`.

MTL4. Software package. URL `http://www.mtl4.org`.

C. J. L. Murray and A. D. Lopez. Alternative projections of mortality and disability by cause 1990-2020: Global burden of disease study. *The Lancet*, 349(9064):1498 –1504, 1997. URL `http://dx.doi.org/10.1016/S0140-6736(96)07492-2`.

J. C. Nagtegaal, D. M. Parks, and J. R. Rice. On numerically accurate finite element solutions in the fully plastic range. *Computer Methods in Applied Mechanics and Engineering*, 4(2):153–177, 1974. URL `http://dx.doi.org/10.1016/0045-7825(74)90032-2`.

M. Nazarov. *An adaptive finite element method for the compressible Euler equations*. Licentiate thesis, 2009.

M. Nazarov. *Adaptive Algorithms and High Order Stabilization for Finite Element Computation of Turbulent Compressible Flow*. PhD thesis, KTH, Numerical Analysis, NA, 2011. QC 20110627.

J.-C. Nédélec. Mixed finite elements in $\mathbf{R}^3$. *Numer. Math.*, 35(3):315–341, 1980. URL `http://dx.doi.org/10.1007/BF01396415`.

J.-C. Nédélec. A new family of mixed finite elements in $\mathbf{R}^3$. *Numer. Math.*, 50(1):57–81, 1986.

M. Nikbakht and G. N. Wells. Automated modelling of evolving discontinuities. *Algorithms*, 2(3): 1008–1030, 2009. URL `http://dx.doi.org/10.3390/a2031008`.

O. Nwogu. Alternative form of Boussinesq equations for nearshore wave propagation. *Journal of Waterway, Port, Coastal, and Ocean Engineering*, 119(6):618–638, 1993.

J. T. Oden and L. Demkowicz. *Applied Functional Analysis*. CRC press, 1996.

R. W. Ogden. *Nonlinear Elastic Deformations*. Dover Publications, 1997.

E. H. Oldfield, K. Muraszko, T. H. Shawker, and N. J. Patronas. Pathophyiology of syringomyelia associated with chiari i malformation of the cerebellar tonsils. *Neurosurg.*, 80:3–15, 1994.

K. B. Ølgaard and G. N. Wells. Optimisations for quadrature representations of finite element tensors through automated code generation. *ACM Transactions on Mathematical Software*, 37(1):8:1–8:23, 2010. URL http://dx.doi.org/10.1145/1644001.1644009.

K. B. Ølgaard, A. Logg, and G. N. Wells. Automated code generation for discontinuous Galerkin methods. *SIAM J. Sci. Comput.*, 31(2):849–864, 2008. URL http://dx.doi.org//10.1137/070710032.

S. A. Orzag. Accurate solution of the Orr–Sommerfeld stability equation. *J. Fluid Mech.*, 50:689–703, 1971. URL http://dx.doi.org/10.1017/S0022112071002842.

S. K. Pandit, J. C. Kalita, and D. C. Dalal. A transient higher order compact scheme for incompressible viscous flows on geometries beyond rectangular. *Journal of Computational Physics*, 225(1):1100–1124, 2007.

R. L. Panton. *Incompressible Flow*. John Wiley & Sons Inc, 1984.

ParaView. Software package. URL http://www.paraview.org.

ParMETIS. Software package. URL http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview.

PaStiX. Software package. URL http://pastix.gforge.inria.fr/.

B. A. Pearlmutter and J. M. Siskind. Lazy multivariate higher-order forward-mode AD. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 155–160, Nice, France, 2007. URL http://dx.doi.org/10.1145/1190216.1190242.

G. Pedersen and F. Løvholt. Documentation of a global Boussinesq solver. Technical report, 2008. URL http://www.math.uio.no/eprint/appl_math/2008/01-08.html.

F. Pellegrini. Scotch. URL http://www.labri.fr/perso/pelegrin/scotch.

G. Pelosi, R. Coccioli, and S. Selleri. *Quick Finite Elements for Electromagnetic Waves*. Artech House, Norwood, 1998. ISBN 0890068488.

D. H. Peregrine. Long waves on a beach. *Journal of Fluid Mechanics Digital Archive*, (27):815–827, 1967. URL http://dx.doi.org/10.1017/S0022112067002605.

A. Peskoff, J. A. Post, and G. A. Langer. Sarcolemmal calcium binding sites in heart: II. mathematical model for diffusion of calcium released from the sarcoplasmic reticulum into the diadic region. *J. Membr. Biol.*, 129(1):59–69, Jul 1992. URL http://dx.doi.org/10.1007/BF00232055.

PETSc. Software package. URL http://www.anl.gov/petsc.

G. Pinna, F. Alessandrini, A. Alfieri, M. Rossi, and A. Bricolo. Cerebrospinal fluid flow dynamics study in Chiari I malformation: implications for syrinx formation. *Neurosurg Focus 8*, 3(3), 2000.

U. Piomelli and E. Balaras. Wall-layer models for large-eddy simulations. *Annu. Rev. Fluid Mech.*, 34: 349–374, 2002.

A. C. Polycarpou, M. Lyons, and C. A. Balanis. Finite element analysis of MMIC waveguide structures with anisotropic substrates. *IEEE Trans. Microwave Theory Tech.*, 44(10):1650–1663, October 1996.

D. M. Pozar. *Microwave Engineering*. John Wiley & Sons, Inc., third edition, 2005.

R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36 (6):1389–1401, 1957. URL http://bstj.bell-labs.com/BSTJ/images/Vol36/bstj36-6-1389.pdf.

C. Prud'homme. Life: Overview of a unified c++ implementation of the finite and spectral element methods in 1d, 2d and 3d. In *Workshop On State-Of-The-Art In Scientific And Parallel Computing*, Lecture Notes in Computer Science, page 10. Springer-Verlag, dec 2006a.

C. Prud'homme. A domain specific embedded language in C++ for automatic differentiation, projection, integration and variational formulations. *Sci. Program.*, 14(2):81–110, 2006b. URL http://portal.acm.org/citation.cfm?id=1376891.1376895.

Pyrex. Software package. URL http://www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex/.

Python C-API. Python C API Reference Manual. URL http://docs.python.org/c-api.

J. Qin. *On the Convergence of Some Simple Finite Elements for Incompressible Flows*. PhD thesis, Penn State, 1994.

A. Quarteroni and A. Valli. *Numerical Approximation of Partial Differential Equations*, volume 23 of *Springer Series in Computational Mathematics*. Springer Verlag, 2008. URL http://dx.doi.org/10.1007/978-3-540-85268-1.

M. F. Quigley, B. J. Iskandar, M. A. Quigley, M. N. Nicosia, and V. Haughton. Cerebrospinal fluid flow in foramen magnum: Temporal and spatial patterns at MR imaging in volunteers and in patients with Chiari I malformation. *Radiology*, 232:229–236, 2004.

S. Ramo, J. R. Whinnery, and T. Van Duzer. *Fields and Waves in Communication Electronics*. John Wiley & Sons, New York, third edition, 1994.

R. Rannacher and S. Turek. A simple nonconforming quadrilateral Stokes element. *Numer. Meth. Parth. Diff. Equ.*, 8:97–111, 1992.

P.-A. Raviart and J. M. Thomas. A mixed finite element method for 2nd order elliptic problems. In *Mathematical aspects of finite element methods (Proc. Conf., Consiglio Naz. delle Ricerche (C.N.R.), Rome, 1975)*, pages 292–315. Lecture Notes in Math., Vol. 606. Springer, Berlin, 1977.

Rayleigh. On the theory of resonance. *Trans. Roy. Soc.*, A161:77–118, 1870.

Y. Ricard. *Physics of mantle convection*, volume 7 of *Treatise on geophysics*, pages 1–30. 2009.

W. Ritz. Über eine neue Methode zur Lösung gewisser Variationsprobleme der mathematischen Physik. *J. reine angew. Math.*, 135:1–61, 1908.

M.-C. Rivara. Mesh refinement processes based on the generalized bisection of simplices. *SIAM Journal on Numerical Analysis*, 21(3):604–613, 1984. URL http://dx.doi.org/10.1137/0721042.

M.-C. Rivara. Local modification of meshes for adaptive and/or multigrid finite-element methods. *Journal of Computational and Applied Mathematics*, 36(1):78–89, 1992.

J. C. Rivenæs. Application of a dual–lithology, depth–dependent diffusion equation in stratigraphic simulation. *Basin Research*, 4:133–146, 1992.

J. C. Rivenæs. *A Computer Simulation Model for Siliciclastic Basin Stratigraphy*. PhD thesis, NTH–Trondheim, 1993.

M. E. Rognes. Automated stability condition tester (ASCoT). 2009. URL https://launchpad.net/ascot.

M. E. Rognes, R. C. Kirby, and A. Logg. Efficient assembly of $H(\mathrm{div})$ and $H(\mathrm{curl})$ conforming finite elements. *SIAM Journal on Scientific Computing*, 31(6):4130–4151, 2009.

M. E. Rognes and A. Logg. Automated Goal-Oriented Error Control I: Stationary Variational Problems 2011. Submitted.

A. Roldan, , V. Wieben, O. Haughton, T. Osswald, and N. Chesler, N. Characterization of complex CSF hydrodynamics at the cranio-vertebral junction with computational flow analysis: Healthy and Chiari I malformation. *American Journal of Neuroradiology*, 30(5):941–946, 2009.

S. Rüdiger, J. W. Shuai, W. Huisinga, C. Nagaiah, G. Warnecke, I. Parker, and M. Falcke. Hybrid stochastic and deterministic simulations of calcium blips. *Biophys. J.*, 93(6):1847–1857, Sep 2007. URL http://dx.doi.org/10.1529/biophysj.106.099879.

T. Rusten and R. Winther. A preconditioned iterative method for saddlepoint problems. *SIAM Journal on Matrix Analysis and Applications*, 13(3):887–904, 1992.

Y. Saad. ILUT: A dual threshold incomplete LU factorization. *Numerical Linear Algebra with Applications*, 1(4):387–402, 1994.

Y. Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, second edition, 2003.

Y. Saad and M. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 7(3):856–869, 1986.

P. Sagaut. *Large-Eddy Simulation for Incompressible Flows — An Introduction*. Springer-Verlag, 2005.

P. Sagaut, S. Deck, and M. Terracol. *Multiscale and Multiresolution Approaches in Turbulence*. Imperial College Press, 2006.

M. Sala, W. F. Spotz, and M. A. Heroux. PyTrilinos: High-performance distributed-memory solvers for python. *ACM Trans. Math. Softw.*, 34(2):1–33, 2008. ISSN 0098-3500. URL http://doi.acm.org/10.1145/1326548.1326549.

H. J. Schroll. *Dual Problems in Depositional Modeling*. Research Report, Simula Research Laboratory, 2008.

U. Schumann. Subgrid-sclae model for finite difference simulation of turbulent flows in plane channels and annuli. *J. Comput. Phys.*, 18:376–404, 1975.

C. Schwab. *p-and hp-Finite Element Methods: Theory and Applications in Solid and Fluid Mechanics*. Oxford University Press, 1998.

L. R. Scott and M. Vogelius. Norm estimates for a maximal right inverse of the divergence operator in spaces of piecewise polynomials. *RAIRO Modél. Math. Anal. Numér.*, 19(1):111–143, 1985. ISSN 0764-583X.

K. Selim, A. Logg, H. Narayanan, and M. G. Larson. An adaptive finite element method for fluid–structure interaction. Submitted.

H. Si. TetGen. a quality tetrahedral mesh generator and three-dimensional delaunay triangulator., 2007. URL http://tetgen.berlios.de.

SILOON. Software package. URL http://acts.nersc.gov/siloon/index.html.

D. Silvester and A. Wathen. Fast iterative solution of stabilised Stokes systems. Part I: Using simple diagonal preconditioners. *SIAM Journal on Numerical Analysis*, 30(3):630–649, 1993.

D. Silvester and A. Wathen. Fast iterative solution of stabilized Stokes systems. Part II: Using block diagonal preconditioners. *SIAM Journal on Numerical Analysis*, 31(5):1352–1367, 1994.

P. P. Silvester. Finite-element solution of homogeneous waveguide problems. *Alta Frequenza*, 38: 313–317, 1969.

J. C. Simo and F. Armero. Unconditional stability and long-term behavior of transient algorithms for the incompressible Navier–Stokes and Euler equations. *Computer Methods in Applied Mechanics and Engineering*, 111:111–154, 1994.

J. C. Simo and T. J. R. Hughes. *Computational Inelasticity*. Springer Verlag, 1998.

J. C. Simo and R. L. Taylor. Consistent tangent operators for rate-independent elastoplasticity. *Computer Methods in Applied Mechanics and Engineering*, 48(1):101–118, 1985. URL http://dx.doi.org/10.1016/0045-7825(85)90070-2.

SIP. Software package. URL http://www.riverbankcomputing.com/software/sip/intro.

J. M. Siskind and B. A. Pearlmutter. Nesting forward-mode AD in a functional framework. *Higher Order Symbol. Comput.*, 21(4):361–376, 2008. URL http://dx.doi.org/10.1007/s10990-008-9037-1.

O. Skavhaug and O. Čertík. Swiginac Python interface to GiNaC, 2009. URL http://swiginac.berlios.de/.

B. W. Smith, J. G. Chase, G. M. Shaw, and R. I. Nokes. Simulating transient ventricular interaction using a minimal cardiovascular system model. *Physiological Measurement*, pages 165–179, 2006.

G. S. Smith. *An Introduction to Classical Electromagnetic Radiation*. Cambridge University Press, 1997.

C. Soeller and M. B. Cannell. Numerical simulation of local calcium movements during l-type calcium channel gating in the cardiac diad. *Biophys. J.*, 73(1):97–111, Jul 1997. URL http://www.pubmedcentral.gov/articlerender.fcgi?tool=pubmed&pubmedid=9199775.

C. Soeller and M. B. Cannell. Examination of the transverse tubular system in living cardiac rat myocytes by 2-photon microscopy and digital image-processing techniques. *Circ. Res.*, 84(3):266–75, 1999. URL http://circres.ahajournals.org/cgi/content/full/84/3/266.

P. Šolín, K. Segeth, and I. Doležel. *Higher-order Finite Element Methods*. Studies in Advanced Mathematics. Chapman & Hall/CRC, 2004.

P. R. Spalart and K. M. Mejia. Analysis of experimental and numerical studies of the rudimentary landing gear. In *49th AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Exposition*, number AIAA-2011-355, 2011.

W. E. Stehbens. Ultrastructure of aneurysms. *Arch Neurol*, 32(12):798–807, 1975. URL http://dx.doi.org/10.1001/archneur.1975.00490540042005.

R. Stenberg. Analysis of mixed finite elements methods for the Stokes problem: a unified approach. *Mathematics of Computation*, 42(165):9–23, 1984.

M. D. Stern, L. S. Song, H. Cheng, J. S. Sham, H. T. Yang, K. R. Boheler, and E. Ríos. Local control models of cardiac excitation-contraction coupling. a possible role for allosteric interactions between ryanodine receptors. *J. Gen. Physiol.*, 113(3):469–489, 1999.

G. Strang and G. J. Fix. *An Analysis of the Finite Element Method*. Prentice-Hall, Englewood Cliffs, 1973.

B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, July 1997.

SWIG. Software package. URL `http://www.swig.org`.

B. Szabó, B. Szabó, and I. Babuška. *Finite Element Analysis*. Wiley-Interscience, 1991.

E. M. Tadjouddine. Vertex-ordering algorithms for automatic differentiation of computer codes. *The Computer Journal*, 51(6):688–699, Nov. 2008. URL `http://dx.doi.org/10.1093/comjnl/bxm115`.

X.-C. Tai and R. Winther. A discrete de Rham complex with enhanced smoothness. *Calcolo*, 43:287–306, 2006.

E. Tan and M. Gurnis. Metastable superplumes and mantle compressibility. *Geophys. Res. Lett.*, 32:1–4, 2005.

A. J. Tanskanen, J. L. Greenstein, A. Chen, S. X. Sun, and R. L. Winslow. Protein geometry and placement in the cardiac dyad influence macroscopic properties of calcium-induced calcium release. *Biophys. J.*, 92(10):3379–3396, May 2007. URL `http://dx.doi.org/10.1529/biophysj.106.089425`.

C. Taylor and P. Hood. A numerical solution of the Navier-Stokes equations using the finite element technique. *Internat. J. Comput. & Fluids*, 1(1):73–100, 1973. URL `http://dx.doi.org/10.1016/0045-7930(73)90027-3`.

R. Temam. Sur l'approximation de la solution des équations de Navier-Stokes par la méthode des pas fractionnaires. *Arc. Ration. Mech. Anal.*, 32:377–385, 1969.

A. R. Terrel, L. R. Scott, M. G. Knepley, and R. C. Kirby. Automated FEM discretizations for the Stokes equation. *BIT Numerical Mathematics*, 48(2):389–404, 2008. URL `http://dx.doi.org/10.1007/s10543-008-0178-8`.

T. E. Tezduyar and Y. J. Park. Discontinuity-capturing finite element formulations for nonlinear convection-diffusion-reaction equations. *Comp. Methods Appl. Mech. Eng.*, 59(3):307–325, 1986.

The Python Tutorial. The python tutorial. `http://docs.python.org/tutorial/`.

L. N. Trefethen. *Spectral Methods in Matlab*. Springer-Verlag, New York, 2006.

Trilinos. Software package. URL `http://trilinos.sandia.gov/`.

U. Trottenberg, C. Oosterlee, and A. Schuller. *Multigrid*. Academic Press, 2001.

C. A. Truesdell and W. Noll. *The Non-linear Field Theories (Handbuch der Physik, band III)*. Springer, Berlin, 1965.

C. A. Truesdell and R. A. Toupin. *The Classical Field Theories (Handbuch der Physik, band III/1)*. Springer-Verlag, Berlin, 1960.

S. Turek. Recent benchmark computations of laminar flow around a cylinder. In *3rd World Conference in Applied Computational Fluid Mechanics*, Freiburg, 1996.

S. Turek. *Efficient Solvers for Incompressible Flow Problems*. Springer, 1999.

uBLAS. Software package. URL `http://www.boost.org/libs/numeric/ublas/doc/`.

Valgrind. Software package. URL `http://valgrind.org`.

A. P. van den Berg, P. E. van Keken, and D. A. Yuen. The effects of a composite non-Newtonian and Newtonian rheology on mantle convection. *Geophys. J. Int.*, 115:62–78, 1993.

D. G. van der Meer, W. Spakman, D. J. J. van Hinsbergen, M. L. Amaru, and T. H. Torsvik. Towards absolute plate motions constrained by lower-mantle slab remnants. *Nature Geoscience*, 3:36–40, 2010.

H. A. van der Vorst. Bi-CGSTAB: A fast and smoothly converging variant of bi-cg for the solution of nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 13(2):631–644, 1992. URL `http://dx.doi.org/10.1137/0913035`.

K. G. van der Zee. *Goal-Adaptive Discretization of Fluid–Structure Interaction*. PhD thesis, Delft University of Technology, 2009.

K. G. van der Zee, E. H. van Brummelen, and R. de Borst. Goal-oriented error estimation for Stokes flow interacting with a flexible channel. *International Journal for Numerical Methods in Fluids*, 56(8): 1551–1557, 2008.

P. E. van Keken, S. D. King, H. Schmeling, et al. A comparison of methods for the modeling of thermochemical convection. *J. Geophys. Res.*, 102(B10):22477–22495, 1997.

G. van Rossum et al. *Python*. URL `http://www.python.org/`.

L. Vardapetyan and L. Demkowicz. hp-vector finite elements method for the full-wave analysis of waveguides with no spurious modes. *Electromagnetics*, 22(5):419–428, 2002.

Vascular Modeling Toolkit. Software package. URL `http://vmtk.org/`.

O. Čertík et al. *SymPy*, 2009. URL `http://docs.sympy.org`.

R. Verfürth. *A posteriori* error estimation and adaptive mesh-refinement techniques. In *Proceedings of the Fifth International Conference on Computational and Applied Mathematics*, pages 67–83. Amsterdam, Elsevier Science Publishers, 1994.

R. Verfürth. A review of a posteriori error estimation techniques for elasticity problems. *Computer Methods in Applied Mechanics and Engineering*, 176(1-4):419–440, 1999.

I. E. Vignon-Clementel, C. A. Figueroa, K. E. Jansen, and C. A. Taylor. Outflow boundary conditions for three-dimensional finite element modeling of blood flow and pressure in arteries. *Computer Methods in Applied Mechanics and Engineering*, 195(29-32):3776 – 3796, 2006. URL `http://dx.doi.org/10.1016/j.cma.2005.04.014`.

Viper. Software package. URL `https://launchpad.net/fenics-viper`.

VMTK. Software package. URL `http://www.vmtk.org`.

VTK. Software package. URL `http://www.kitware.com`.

M. Walkley. *A Numerical Method for Extended Boussinesq Shallow-Water Wave Equations*. PhD thesis, The University of Leeds, School of Computer Studies, 1999.

M. Walkley and M. Berzins. A finite element method for the two-dimensional extended Boussinesq equations. *Internat. J. Numer. Methods Fluids*, 39(10):865–885, 2002.

T. Warburton. An explicit construction for interpolation nodes on the simplex. *Journal of Engineering Mathematics*, 2005.

Weave. Software package. URL `http://scipy.org/Weave`.

J. P. Webb. Edge elements and what they can do for you. *IEEE Trans. Magn.*, 29(2):1460–1465, March 1993.

G. Wei and J. T. Kirby. Time-dependent numerical code for extended Boussinesq equations. *Journal of Waterway, Port, Coastal, and Ocean Engineering*, 121(5):251–261, 1995. URL `http://dx.doi.org/10.1061/(ASCE)0733-950X(1995)121:5(251)`.

G. Wei, J. T. Kirby, and A. Sinha. Generation of waves in Boussinesq models using a source function method. *Coastal Engineering*, 36:271–279, 1999.

B. Weir. Unruptured intracranial aneurysms: a review. *Journal of Neurosurgery*, 96(1):3–42, 2002. URL `http://dx.doi.org/10.3171/jns.2002.96.1.0003`.

G. N. Wells and L. J. Sluys. A new method for modelling cohesive cracks using finite elements. *International Journal for Numerical Methods in Engineering*, 50(12):2667–2682, 2001. URL `http://dx.doi.org/10.1002/nme.143`.

J. R. Welty, C. E. Wicks, and R. E. Wilson. *Fundamentals of Momentum, Heat, and Mass Transfer*. John Wiley & Sons Inc, 2001.

P. Wesseling. *An Introduction to Multigrid Methods*. Wiley & Sons, 1992.

R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001. URL `http://dx.doi.org/10.1016/S0167-8191(00)00087-9`.

F. M. White. *Viscous Fluid Flow*. McGraw-Hill, 1991.

F. M. White. *Fluid Mechanics*. McGraw-Hill, fourth edition, 1999.

G. B. Whitham. *Linear and Nonlinear Waves*. John Wiley & Sons, New York, 1974.

I. M. Wilbers, H. P. Langtangen, and A. Ødegård. Using Cython to speed up numerical Python programs. In B. Skallerud and H. I. Andersson, editors, *Proceedings of MekIT'09*, pages 495–512. Norwegian University of Science and Technology, Tapir Academic Press, 2009. ISBN 978-82-519-2421-4.

M. M. Wolf and M. T. Heath. Combinatorial optimization of matrix-vector multiplicaion in finite element assembly. *SIAM J. Sci. Comput.*, 31:2960, 2009.

S.-B. Woo and P.-F. Liu. Finite element model for modified Boussinesq equations i: Model development. *Journal of Waterway, Port, Coastal and Ocean Engineering*, 130(1):1–16, 2004a.

S.-B. Woo and P.-F. Liu. Finite element model for modified Boussinesq equations II: Applications to nonlinear harbor oscillations. *Journal of Waterway, Port, Coastal and Ocean Engineering*, 130(1):17–28, 2004b.

T. Y. Wu. Long waves in ocean and coastal waters. *Journal of the Engineering Mechanics*, 107(3):501–522, 1981.

M. M. Zdravkovich. *Flow Around Circular Cylinders*. Oxford University Press, 2003.

M. Zhao, B. Teng, and L. Cheng. A new form of generalized Boussinesq equations for varying water depth. *Ocean Engineering*, 31:2047–2072, 11 2004. URL `http://dx.doi.org/10.1016/j.oceaneng.2004.03.010`.

Y. Zhu and A. C. Cangellaris. *Multigrid Finite Element Methods for Electromagnetic Field Modelling*. IEEE Press, New York, 2006.

O. C. Zienkiewicz and R. L. Taylor. *The Finite Element Method: Fluid dynamics*, volume 3. Butterworth-Heinemann, 5 edition, 2000.

O. C. Zienkiewicz and J. Z. Zhu. A simple error estimator and adaptive procedure for practical engineerng analysis. *International Journal for Numerical Methods in Engineering*, 24(2), 1987.

O. C. Zienkiewicz, R. L. Taylor, and J. Z. Zhu. *The Finite Element Method — Its Basis and Fundamentals*. Elsevier, 6th edition, 2005.

Index

∇, 308
Argument, 305
Box, 57
CellFunction, 186
CellIterator, 185
Cell, 183
Coefficients, 305
Coefficient, 305
ComponentTensor, 306
Constant, 305
DirichletBC, 7, 200
DofMap, 189
Dx, 308
EdgeFunction, 186
EdgeIterator, 185
Edge, 183
Expression, 7, 193
Expr, 315
FaceFunction, 186
FaceIterator, 185
FacetFunction, 186
FacetIterator, 185
Facet, 183
Face, 183
FiniteElement, 188, 302
Form, 303
FunctionSpace, 6, 189
Function, 191
Identity, 305
IndexSum, 306
Indexed, 306
Index, 306
Integral, 303
Interval, 57
KrylovSolver, 35, 176
LUSolver, 176
LinearVariationalProblem, 11, 202
LinearVariationalSolver, 11, 202
ListTensor, 306
Matrix, 174
Measure, 303
MeshConnectivity, 184
MeshData, 187
MeshEditor, 182
MeshEntityIterator, 185
MeshEntity, 183
MeshFunction, 186
MeshGeometry, 184
MeshTopology, 184
Mesh, 6, 181
MixedElement, 302
NewtonSolver, 180
NonlinearVariationalProblem, 46, 202
NonlinearVariationalSolver, 46, 202
Operator, 315
Parameters, 211
Point, 186
Progress, 210
Rectangle, 57
TensorConstant, 305
TensorElement, 302
Terminal, 305, 315
TestFunctions, 305
TestFunction, 6, 305
TimeSeries, 207
Timer, 211
TrialFunctions, 305
TrialFunction, 6, 305
UnitCircle, 57
UnitCube, 57, 181
UnitInterval, 57
UnitSphere, 57
UnitSquare, 57, 181
VectorConstant, 305
VectorElement, 302
Vector, 174
VertexFunction, 186
VertexIterator, 185

`Vertex`, 183
`acos`, 308
`action`, 311
`adjoint`, 311
`as_matrix`, 306
`as_tensor`, 306
`as_vector`, 306
`asin`, 308
`assemble_system`, 34
`assemble`, 34, 50, 197
`atan`, 308
`avg`, 310
`begin`, 210
`compile_element`, 224
`compile_form`, 224
`cos`, 308
`cout`, 208
`cross`, 308
`curl`, 308
`derivative`, 311
`det`, 308
`diff`, 308
`div`, 308
`dolfin-convert`, 203
`dot`, 308
`dx`, 308
`elem_op`, 308
`endl`, 208
`end`, 210
`energy_norm`, 311
`error`, 208
`exp`, 308
`grad`, 308
`info`, 11, 208
`inner`, 308
`instant-clean`, 261
`instant-showcache`, 261
`inv`, 308
`jump`, 310
`lhs`, 311
`list_krylov_solver_methods`, 176
`list_krylov_solver_preconditioners`, 176
`list_lu_solver_methods`, 176
`ln`, 308
`outer`, 308
`plot`, 18, 204
`pow`, 308
`pydoc`, 13, 67
`replace`, 311
`rhs`, 311
`rot`, 308
`self`, 67
`sensitivity_rhs`, 311
`set_log_active`, 208
`set_log_level`, 208
`sin`, 308
`solve`, 175, 202
`split`, 305
`sqrt`, 308
`system`, 311
`tan`, 308
`transpose`, 308
`tr`, 308
`warning`, 208
`diffsim`, 625, 626

a posteriori error estimate, *see* error estimate
a priori error estimate, *see* error estimate
action, 133
AD, *see* automatic differentiation
adaptive refinement, 187
adaptivity, 88, 565
adjoint, 135
advection–diffusion equation, 589, 614
    discontinuous Galerkin discretization, 591
affine equivalence, 84, 92
affine mapping, 83, 127
ALE, 557
algebraic operator, 308
Argyris element, *see* finite element
Arnold–Winther element, *see* finite element
assembly, 137, 280
    algorithm, 137
    implementation, 139, 197, 286
    increasing efficiency, 50
    parallel implementation, 140
atomic value, 305
automatic calibration, 603
automatic differentiation, 45, 321
    forward mode, 321
    reverse mode, 321
automation, 89

Babuška condition, 659
basis function, 305
Beltrami flow, 412
bilinear form, 76

block preconditioner, 649
boundary condition, 62, 73, 198, 482
    Dirichlet, 7, 62, 73, 75, 200
    essential, 75, 200
    natural, 75, 198
    Neumann, 30, 62, 73, 75, 198
    Robin, 62
    symmetric application of, 140
boundary integral, 198
boundary markers, 186
boundary measure, 303
Boussinesq model, 471
Brezzi coercivity condition, 659
Brezzi inf-sup condition, 659
Brezzi–Douglas–Marini element, *see* finite element, 83
Bubble element, *see* finite element
Burgers' equation, 429

cache, 262
CBC.Block, 643
CBC.Flow, 559
CBC.Solve, 527, 559, 569
CBC.Swing, 569
CBC.Twist, 527, 536, 559
Cea's lemma, 86
cell, 183
cell integral, 303
cell membrane, 613, 614
cerebrospinal fluid flow, 455
CG element, *see* finite element
channel transition, 625
Chiari I malformation, 455
Chorin's method, 397
Ciarlet finite element definition, *see* finite element
code generation utilities, 284
coefficient, 305
collapsed-coordinate polynomial, 243
compiler, 228
computational graph, 318
conjugate gradient method, 645
consistent splitting scheme, 399
constitutive model, 508
contour plot, 28
contravariant Piola mapping, 83
convection, 421
coordinate stretching, 58
coordinate transformation, 58
covariant Piola mapping, 83
cross product, 308
Crouzeix–Raviart element, *see* finite element, 83
CSS, *see* consistent splitting scheme
cylinder flow, 409
Cython, 268
cytosol, 613, 615

Dörfler marking, 89
debugging, 377
degrees of freedom, 12, 92
deposition, 603
derivative, 45
determinant, 308
DG element, *see* finite element
DG operator, 310
differential operator, 308
differentiation, 321
diffusion constant, 614
dimension-independent code, 36
Dirichlet boundary condition, *see* boundary condition
discontinuity, 573
discontinuous Galerkin, 310, 591
discontinuous Lagrange element, *see* finite element
discrete operator, 132
discrete stability, 655
discrete state, 616
discretization, 74
dispersion curve, 634, 636, 639, 641, 642
dispersion relation, 478
DOLFIN, 171
domain specific language, 299
dot product, 308
driven cavity, 406
dual lithology model, 603
dual problem, 87, 562, 604
duality argument, 604
Dubiner polynomial, 244
dyadic cleft, 614

edge, 183
efficiency index, 89
eigenvalue problem, 178, 633
elasticity, 527
elastodynamics, 522
electro-diffusion, 614
electromagnetics, 629
element tensor, 223

energy conservation, 421
energy functional, 24
Epetra, 179
error estimate
    a posteriori, 87
    a priori, 85
    goal oriented, 87
error estimation, 85, 562
error functional, 24
Eulerian framework, 556
event-driven simulator, 626
expression, 193, 315
    with parameters, 16
expression representation, 328
expression transformation, 327, 328
expression tree, 315, 325
extended finite element method, 573
exterior facet integral, 303

F2PY, 268
face, 183
facet, 183
facet normal, 305
FEniCS Apps, 625
FErari, 235
FFC, 223
FIAT, 243
Fick's second law, 614
file formats, 203
    DOLFIN XML, 182, 206
    PVD, 204
    VTU, 204
filtering, 593
finite difference time discretization, 47
finite element
    Argyris, 105
    Arnold–Winther, 100
    Brezzi–Douglas–Marini, 99
    Bubble element, 110
    Crouzeix–Raviart, 95
    definition, 79, 91, 118
    Discontinuous Lagrange, 105, 302
    Hermite, 107
    implementation, 188
    Lagrange, 6, 94
    list of supported, 189
    Mardal–Tai–Winther, 100
    Morley, 108
    Nédélec, 102, 103
    Raviart–Thomas, 98
finite element assembly, *see* assembly
finite element exterior calculus, 110
flops, 150
fluid–structure interaction, 545, 555
flux functional, 27
foramen magnum, 455
form, 303
    action of, 133
    adjoint of, 135
    algorithms, 325
    argument of, 305
    canonical, 131
    derivative of, 134
    language, 299
    multilinear, 130
    operator, 311
    representation of, 230
form compiler, 223, 269
forward mode AD, *see* automatic differentiation
Fourier's law, 74
function, 191, 305
    evaluation, 192
    subfunction, 193
function space, 78, 81, 189
    mixed, 190
    subspace, 191
functional, 23, 299

G2, 401
Gateaux derivative, 44
generalized finite element method, 573
Gillespie method, 625
GiNaC, 269
GMRES, 178
Gmsh, 462
goal oriented error estimate, *see* error estimate
Gouy-Chapman, 614
GRPC, 402

hemodynamics, 439
Hermite element, *see* finite element, 84
heterogeneous medium, 54, 59
hexahedron, 291, 296
hyperelasticity, 520, 527

identity matrix, 305
ILU, 178
implicit summation, 306

incompressible Navier–Stokes equations, 395, 419, 439, 455, 545, 555
incremental pressure correction, 397
index notation, 306
inner product, 308
input/output, 203
Instant, 253
integral, 303
interior facet integral, 303
interior measure, 303
interpolation, 14, 16
interval, 290, 295
inverse, 308
inverse approach, 604
IPCS, *see* incremental pressure correction

Jacobian, 45
JIT, *see* just-in-time compilation, 253
jump, 310
just-in-time compilation, 218, 232

kinetic energy, 422

Ladyzhenskaya–Babuška–Brezzi conditions, 76, 649, 658
Lagrange finite element, *see* finite element
Lagrangian framework, 556
Landweber algorithm, 604
language operator, 308
LBB conditions, *see* Ladyzhenskaya–Babuška–Brezzi conditions
license, ii
linear algebra, 174
linear algebra backend, 10, 179
linear elasticity, 508
linear form, 76
linear solver, 84
linear system, 34, 175
linearization, 77, 511
local-to-global mapping, 81, 128, 223
log level, 208
logging, 208
loop hoisting, 144

mantle convection, 587
mapping from reference element, 83, 127
Mardal–Tai–Winther element, *see* finite element
Markov chain model, 616, 625
material model, 529
matrix, 174
matrix-free method, 141
Maxwell's equations, 629
MayaVi, 204
mesh, 6, 79, 181
  coloring, 214
  connectivity, 184
  creating, 182
  data, 187
  distributed, 188
  geometry, 184
  iterators, 185
  partitioning, 215
  reading, 182
  refinement, 187
  topology, 184
  transformation, 58
  XML format, 182
mesh entity, 183, 292
metaclass, 218
microstrip, *see* shielded microstrip
mixed finite element method, 655
mixed function space, 190
mixed problem, 75
Morley element, *see* finite element
MPI, 215
MTL4, 10, 179
multi-material domain, 54, 59
multicore, 214
multifunction, 326
multilinear form, *see* form
multithreading, 214

Nédélec element, *see* finite element, 83
Navier–Stokes, *see* incompressible Navier–Stokes equations
Nernst–Planck equation, 614
Neumann boundary condition, *see* boundary condition
Newton's method, 77, 180
Newtonian fluid, 456
nodal basis, 80, 91
nonlinear diffusion, 603
nonlinear elasticity, 527
nonlinear PDE, *see* partial differential equation
nonlinear problem, 46, 77
nonlinear system, 180
numbering, 292
  hexahedron, 296
  interval, 295

quadrilateral, 296
tetrahedron, 296
triangle, 295
numerical quadrature, 143
NumPy, 221

operator, 315
optimization, 231, 235
BLAS, 166
geometric, 164
topological, 162
Orr–Sommerfeld equation, 431
outer product, 308

parallel computing, 214
distributed memory, 215
shared memory, 214
parameters, 10, 211, 228
ParaView, 204
ParMETIS, 215
partial differential equation
nonlinear, 38
time-dependent, 47
partition of unity method, 573
PDE, *see* partial differential equation
PDE constrained minimization, 604
PETSc, 10, 179
Picard iteration, 38
Piola mapping, 83
plasticity, 509
Poisson's equation, 2, 73
mixed finite element discretization, 665
nonlinear, 78
variable coefficient, 21
polynomial space
constrained, 248
supplemented, 248
postprocessing, 204
potential, 475
preconditioner, 85, 643
predictor–corrector scheme, 480, 592
preprocessing, 203
pressure-driven channel, 407
prime basis, 243
progress bar, 210
project, 21
projection, 19, 21
propagation constant, 630
propensity function, 616, 625
PyTrilinos, 643, 645

quadrature element, 514
quadrature representation, 143, 231
optimization of, 144
quadrilateral, 291, 296

random start vector (linear systems), 35
Raviart–Thomas element, *see* finite element
Rayleigh number, 588
reduced discrete stability, 660
reference cell, 290
hexahedron, 291
interval, 290
quadrilateral, 291
tetrahedron, 291
triangle, 290
reference counting, 367
referential transparency, 315
reflective boundary, 483
residual, 85
restriction, 310
reverse mode AD, *see* automatic differentiation
Robin boundary condition, *see* boundary condition
root-mean-square velocity, 599
Runge–Kutta, 480
ryanodine receptor, 613, 615, 616

saddle point problem, 402, 658
sarcoplasmic reticulum, 613, 615
SAS, 455
SciPy, 221
SCOTCH, 215
Scott–Vogelius method, 385
screening, 614
sedimentation model, 603
SFC, 223, 269
shared pointer, 367
shielded microstrip, 640
signature, 262, 333
SLEPc, 35, 178
solid mechanics, 505, 527
source function, 483
spatial coordinates, 305
spinal canal, 455
spinal cord, 455
sponge-layer, 483
spurious mode, 633

stabilization, 385, 619
    element Péclet number, 619
    parameter, 619
    streamline upwind Petrov–Galerkin, 619
steepest descent, 604
stochastic channel, 615, 616
Stokes equations, 381
    iterative linear solvers, 594
    mixed finite element discretization, 590, 667
structured mesh, 27
subarachnoid space, 455
SUPG, *see* stabilization
SWIG, 361
Swiginac, 269
SyFi, 269
symbolic differentiation, 321

t-tubule, 614, 615
Taylor–Green vortex, 409, 435
tensor algebra operator, 308
tensor representation, 157, 159, 231
    optimization of, 161, 235
terminal value, 305, 315
test function, 3
TetGen, 619
tetrahedron, 291, 296
time integration, 511, 533
time series, 207
time stepping, 625
time-dependent PDE, *see* partial differential equation
timing, 211
topological dimension, 292
trace, 308
transition to turbulence, 419
transport, 603
transpose, 308
tree traversal, 325
trial function, 3
triangle, 290, 295
Trilinos, 10, 179
turbulent flow, 545
typemap, 253, 368

uBLAS, 10, 179
UFC, 279
UFL, 299, 334
UMFPACK, 10
under-relaxation, 40
Unicorn, 335, 545
unisolvence, 91
user interfaces, 171
    C++, 172
    Python, 173

variational form, 195, 299
variational problem, 2, 129, 201
vector, 174
vector Helmholtz equation, 630
vertex, 183
vertex numbering, 292
Viper, 18
visualization, 18
    of structured mesh, 27
VTK, 18

water waves, 471
waveguide, 629
    cutoff analysis, 632, 638, 639
    dispersion analysis, 633, 638, 640, 642
    half-loaded rectangular, 639–642
    hollow rectangular, 637–639
wavenumber
    cutoff, 632, 636, 638, 640
    operating, 630, 632
weak form, 299
Weave, 268
Windkessel model, 254

XFEM, *see* extended finite element method
XML format, *see* file formats