Hans Petter Langtangen[*], Anders Logg[†]

Solving PDEs in Minutes – The FEniCS Tutorial Volume I

Mar 21, 2017

Springer

[*]Center for Biomedical Computing, Simula Research Laboratory and Department of Informatics, University of Oslo.

[†]Email: logg@chalmers.se. Department of Mathematical Sciences, Chalmers University of Technology; Center for Biomedical Computing, Simula Research Laboratory; and Computational Engineering and Design, Fraunhofer-Chalmers Centre.

Contents

Preface

This book gives a concise and gentle introduction to finite element programming in Python based on the popular FEniCS software library. FEniCS can be programmed in both C++ and Python, but this tutorial focuses exclusively on Python programming, since this is the simplest and most effective approach for beginners. After having digested the examples in this tutorial, the reader should be able to learn more from the FEniCS documentation, the numerous demo programs that come with the software, and the comprehensive FEniCS book *Automated Solution of Differential Equations by the Finite Element Method* [26]. This tutorial is a further development of the opening chapter in [26].

We thank Johan Hake, Kent-Andre Mardal, and Kristian Valen-Sendstad for many helpful discussions during the preparation of the first version of this tutorial for the FEniCS book [26]. We are particularly thankful to Professor Douglas Arnold for very valuable feedback on early versions of the text. Øystein Sørensen pointed out numerous typos and contributed with many helpful comments. Many errors and typos were also reported by Mauricio Angeles, Ida Drøsdal, Miroslav Kuchta, Hans Ekkehard Plesser, Marie Rognes, Hans Joachim Scroll, Glenn Terje Lines, Simon Funke, Matthew Moelter, and Magne Nordaas. Ekkehard Ellmann as well as two anonymous reviewers provided a series of suggestions and improvements. Special thanks go to Benjamin Kehlet for all his work with the `mshr` tool and for quickly implementing our requests for this tutorial.

Comments and corrections can be reported as *issues* for the Git repository of this book[1], or via email to `logg@chalmers.se`.

Oslo and Smögen, November 2016 *Hans Petter Langtangen, Anders Logg*

[1] `https://github.com/hplgit/fenics-tutorial/`

Chapter 1
Preliminaries

1.1 The FEniCS Project

The FEniCS Project is a research and software project aimed at creating mathematical methods and software for automated computational mathematical modeling. This means creating easy, intuitive, efficient, and flexible software for solving partial differential equations (PDEs) using finite element methods. FEniCS was initially created in 2003 and is developed in collaboration between researchers from a number of universities and research institutes around the world. For more information about FEniCS and the latest updates of the FEniCS software and this tutorial, visit the FEniCS web page at `https://fenicsproject.org`.

FEniCS consists of a number of building blocks (software components) that together form the FEniCS software: DOLFIN [27], FFC [17], FIAT [16], UFL [1], mshr, and a few others. For an overview, see [26]. FEniCS users rarely need to think about this internal organization of FEniCS, but since even casual users may sometimes encounter the names of various FEniCS components, we briefly list the components and their main roles in FEniCS. DOLFIN is the computational high-performance C++ backend of FEniCS. DOLFIN implements data structures such as meshes, function spaces and functions, compute-intensive algorithms such as finite element assembly and mesh refinement, and interfaces to linear algebra solvers and data structures such as PETSc. DOLFIN also implements the FEniCS problem-solving environment in both C++ and Python. FFC is the code generation engine of FEniCS (the form compiler), responsible for generating efficient C++ code from high-level mathematical abstractions. FIAT is the finite element backend of FEniCS, responsible for generating finite element basis functions, UFL implements the abstract mathematical language by which users may express variational problems, and mshr provides FEniCS with mesh generation capabilities.

1.2 What you will learn

The goal of this tutorial is to demonstrate how to apply the finite element to solve PDEs in FEniCS. Through a series of examples, we demonstrate how to:

- solve linear PDEs (such as the Poisson equation),
- solve time-dependent PDEs (such as the heat equation),
- solve nonlinear PDEs,
- solve systems of time-dependent nonlinear PDEs.

Important topics involve how to set boundary conditions of various types (Dirichlet, Neumann, Robin), how to create meshes, how to define variable coefficients, how to interact with linear and nonlinear solvers, and how to postprocess and visualize solutions.

We will also discuss how to best structure the Python code for a PDE solver, how to debug programs, and how to take advantage of testing frameworks.

1.3 Working with this tutorial

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 the step from solving a simple model problem to a challenging real-world problem is often quite short and easy with FEniCS.

Using FEniCS to solve PDEs may seem to require a thorough understanding of the abstract mathematical framework of the finite element method as well as expertise in Python programming. Nevertheless, it turns out that many users 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 at how easy it is to solve PDEs with FEniCS!

1.4 Obtaining the software

Working with this tutorial obviously requires access to the FEniCS software. FEniCS is a complex software library, both in itself and due to its many dependencies to state-of-the-art open-source scientific software libraries. Manually building FEniCS and all its dependencies from source can thus be a daunting task. Even for an expert who knows exactly how to configure and build each component, a full build can literally take hours! In addition to the complexity of the software itself, there is an additional layer of complexity in how many different kinds of operating systems (Linux, Mac, Windows) may be running on a user's laptop or compute server, with different requirements for how to configure and build software.

For this reason, the FEniCS Project provides prebuilt packages to make the installation easy, fast, and foolproof.

FEniCS download and installation

In this tutorial, we highlight two main options for installing the FEniCS software: Docker containers and Ubuntu packages. While the Docker containers work on all operating systems, the Ubuntu packages only work on Ubuntu-based systems. Note that the built-in FEniCS plotting does currently not work from Docker, although rudimentary plotting is supported via the Docker Jupyter notebook option.

FEniCS may also be installed using other methods, including Conda packages and building from source. For more installation options and the latest information on the simplest and best options for installing FEniCS, check out the official FEniCS installation instructions. These can be found at `https://fenicsproject.org/download`.

FEniCS version: 2016.2

FEniCS versions are labeled 2016.1, 2016.2, 2017.1 and so on, where the major number indicates the year of release and the minor number is a counter starting at 1. The number of releases per year varies but typically one can expect 2–3 releases per year. This tutorial was prepared for and tested with FEniCS version 2016.2.

1.4.1 Installation using Docker containers

A modern solution to the challenge of software installation on diverse software platforms is to use so-called *containers*. The FEniCS Project provides custom-made containers that are controlled, consistent, and high-performance software environments for FEniCS programming. FEniCS containers work equally well[1] on all operating systems, including Linux, Mac, and Windows.

To use FEniCS containers, you must first install the Docker platform. Docker installation is simple and instructions are available on the Docker web page[2]. Once you have installed Docker, just copy the following line into a terminal window:

Terminal

```
Terminal> curl -s https://get.fenicsproject.org | bash
```

The command above will install the program `fenicsproject` on your system. This program lets you easily create FEniCS sessions (containers) on your system:

Terminal

```
Terminal> fenicsproject run
```

This command has several useful options, such as easily switching between the latest release of FEniCS, the latest development version and many more. To learn more, type `fenicsproject help`. FEniCS can also be used directly with Docker, but this typically requires typing a relatively complex Docker command, for example:

Terminal

```
docker run --rm -ti -v `pwd`:/home/fenics/shared -w
/home/fenics/shared quay.io/fenicsproject/stable:current '/bin/bash -l
-c "export TERM=xterm; bash -i"'
```

Sharing files with FEniCS containers

When you run a FEniCS session using `fenicsproject run`, it will automatically share your current working directory (the directory from

[1] Running Docker containers on Mac and Windows involves a small performance overhead compared to running Docker containers on Linux. However, this performance penalty is typically small and is often compensated for by using the highly tuned and optimized version of FEniCS that comes with the official FEniCS containers, compared to building FEniCS and its dependencies from source on Mac or Windows.

[2] `https://www.docker.com`

which you run the `fenicsproject` command) with the FEniCS session. When the FEniCS session starts, it will automatically enter into a directory named `shared` which will be identical with your current working directory on your host system. This means that you can easily edit files and write data inside the FEniCS session, and the files will be directly accessible on your host system. It is recommended that you edit your programs using your favorite editor (such as Emacs or Vim) on your host system and use the FEniCS session only to run your program(s).

1.4.2 Installation using Ubuntu packages

For users of Ubuntu GNU/Linux, FEniCS can also be installed easily via the standard Ubuntu package manager `apt-get`. Just copy the following lines into a terminal window:

Terminal

```
Terminal> sudo add-apt-repository ppa:fenics-packages/fenics
Terminal> sudo apt-get update
Terminal> sudo apt-get install fenics
Terminal> sudo apt-get dist-upgrade
```

This will add the FEniCS package archive (PPA) to your Ubuntu computer's list of software sources and then install FEniCS. It will will also automatically install packages for dependencies of FEniCS.

Watch out for old packages!

In addition to being available from the FEniCS PPA, the FEniCS software is also part of the official Ubuntu repositories. However, depending on which release of Ubuntu you are running, and when this release was created in relation to the latest FEniCS release, the official Ubuntu repositories might contain an outdated version of FEniCS. For this reason, it is better to install from the FEniCS PPA.

1.4.3 Testing your installation

Once you have installed FEniCS, you should make a quick test to see that your installation works properly. To do this, type the following command in a FEniCS-enabled[3] terminal:

Terminal

```
Terminal> python -c 'import fenics'
```

If all goes well, you should be able to run this command without any error message (or any other output).

1.5 Obtaining the tutorial examples

In this tutorial, you will learn finite element and FEniCS programming through a number of example programs that demonstrate both how to solve particular PDEs using the finite element method, how to program solvers in FEniCS, and how to create well-designed Python code that can later be extended to solve more complex problems. All example programs are available from the web page of this book at `https://fenicsproject.org/tutorial`. The programs as well as the source code for this text can also be accessed directly from the Git repository[4] for this book.

1.6 Background knowledge

1.6.1 Programming in Python

While you can likely pick up basic Python programming by working through the examples in this tutorial, you may want to study additional material on the Python language. A natural starting point for beginners is the classic *Python Tutorial* [11], or a tutorial geared towards scientific computing [22]. In the latter, you will also find pointers to other tutorials for scientific computing in Python. Among ordinary books we recommend the general introduction *Dive into Python* [28] as well as texts that focus on scientific computing with Python [15, 18–21].

[3] For users of FEniCS containers, this means first running the command `fenicsproject run`.

[4] `https://github.com/hplgit/fenics-tutorial/`

Python versions

Python comes in two versions, 2 and 3, and these are not compatible. FEniCS works with both versions of Python. All the programs in this tutorial are also developed such that they can be run under both Python 2 and 3. Python programs that need to print must then start with

```
from __future__ import print_function
```

to enable the `print` function from Python 3 in Python 2. All use of `print` in the programs in this tutorial consists of function calls, like `print('a:', a)`. Almost all other constructions are of a form that looks the same in Python 2 and 3.

1.6.2 The finite element method

Many good books have been written on the finite element method. The books typically fall in either of two categories: the abstract mathematical version of the method or the engineering "structural analysis" formulation. FEniCS builds heavily on concepts from the abstract mathematical exposition. The first author has a book[5] [24] in development that explains all details of the finite element method in an intuitive way, using the abstract mathematical formulations that FEniCS employs.

The finite element text by Larson and Bengzon [25] is our recommended introduction to the finite element method, with a mathematical notation that goes well with FEniCS. An easy-to-read book, which also provides a good general background for using FEniCS, is Gockenbach [12]. The book by Donea and Huerta [8] has a similar style, but aims at readers with an interest in fluid flow problems. Hughes [14] is also recommended, especially for readers interested in solid mechanics and heat transfer applications.

Readers with a background in the engineering "structural analysis" version of the finite element method may find Bickford [3] 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* [9] is a very good choice. On the other hand, FEniCS users with a strong background in mathematics will appreciate the texts by Brenner and Scott [5], Braess [4], Ern and Guermond [10], Quarteroni and Valli [29], or Ciarlet [7].

[5] `http://hplgit.github.io/fem-book/doc/web/index.html`

Chapter 2
Fundamentals: Solving the Poisson equation

The goal of this chapter is to show how the Poisson equation, the most basic of all PDEs, can be quickly solved with a few lines of FEniCS code. We introduce the most fundamental FEniCS objects such as `Mesh`, `Function`, `FunctionSpace`, `TrialFunction`, and `TestFunction`, and learn how to write a basic PDE solver, including how to formulate the mathematical variational problem, apply boundary conditions, call the FEniCS solver, and plot the solution.

2.1 Mathematical problem formulation

Many books on programming languages start with a "Hello, World!" program. Readers are curious to know how fundamental tasks are expressed in the language, and printing a text to the screen can be such a task. In the world of *finite element methods for PDEs*, the most fundamental task must be to solve the Poisson equation. Our counterpart to the classical "Hello, World!" program therefore solves the following boundary-value problem:

$$-\nabla^2 u(\boldsymbol{x}) = f(\boldsymbol{x}), \quad \boldsymbol{x} \text{ in } \Omega, \tag{2.1}$$

$$u(\boldsymbol{x}) = u_{\mathrm{D}}(\boldsymbol{x}), \quad \boldsymbol{x} \text{ on } \partial\Omega\,. \tag{2.2}$$

Here, $u = u(\boldsymbol{x})$ is the unknown function, $f = f(\boldsymbol{x})$ is a prescribed function, ∇^2 is the Laplace operator (often written as Δ), Ω is the spatial domain, and $\partial\Omega$ is the boundary of Ω. The Poisson problem, including both the PDE $-\nabla^2 u = f$ and the boundary condition $u = u_{\mathrm{D}}$ on $\partial\Omega$, is an example of 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 as

$$-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} = f(x,y)\,. \tag{2.3}$$

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

The Poisson equation 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 for more complicated systems of PDEs, in particular the Navier–Stokes equations.

Solving a boundary-value problem such as the Poisson equation in FEniCS consists of the following steps:

1. Identify the computational domain (Ω), the PDE, its boundary conditions, and source terms (f).
2. Reformulate the PDE as a finite element variational problem.
3. Write a Python program which defines the computational domain, the variational problem, the boundary conditions, and source terms, using the corresponding FEniCS abstractions.
4. Call FEniCS to solve the boundary-value problem and, optionally, extend the program to compute derived quantities such as fluxes and averages, and visualize 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 a similar program in most other software frameworks for PDEs require much more code and technically difficult programming.

What makes FEniCS attractive?

Although many software frameworks have a really elegant "Hello, World!" example for the Poisson equation, FEniCS is to our knowledge the only framework where the code stays compact and nice, very close to the mathematical formulation, even when the mathematical and algorithmic complexity increases and when moving from a laptop to a high-performance compute server (cluster).

2.1.1 Finite element variational formulation

FEniCS is based on the finite element method, which is a general and efficient mathematical machinery for the numerical solution of PDEs. The starting point for the finite element methods is a PDE expressed in *variational form*. Readers who are not familiar with variational problems will get a very brief introduction to the topic in this tutorial, but reading a proper book on the

finite element method in addition is encouraged. Section 1.6.2 contains a list of recommended books. Experience shows that you can work with FEniCS as a tool to solve PDEs even without thorough knowledge of the finite element method, as long as you get somebody to help you with formulating the PDE as a variational problem.

The basic recipe for turning a PDE into a variational problem is to multiply the PDE by a function v, integrate the resulting equation over the domain Ω, and perform integration by parts of terms with second-order derivatives. The function v which multiplies the PDE is called a *test function*. The unknown function u to be approximated is referred to as a *trial function*. The terms trial and test functions are used in FEniCS programs too. The trial and test functions belong to certain so-called *function spaces* that specify the properties of the functions.

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

$$-\int_\Omega (\nabla^2 u)v \,\mathrm{d}x = \int_\Omega fv \,\mathrm{d}x\,. \tag{2.4}$$

We here let $\mathrm{d}x$ denote the differential element for integration over the domain Ω. We will later let $\mathrm{d}s$ denote the differential element for integration over the boundary of Ω.

A common rule when we derive variational formulations is that we try to keep the order of the derivatives of u and v as small as possible. Here, we have a second-order spatial derivative of u, which can be transformed to a first-derivative of u and v by applying the technique of integration by parts[1]. The formula reads

$$-\int_\Omega (\nabla^2 u)v \,\mathrm{d}x = \int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x - \int_{\partial\Omega} \frac{\partial u}{\partial n} v \,\mathrm{d}s, \tag{2.5}$$

where $\frac{\partial u}{\partial n} = \nabla u \cdot n$ is the derivative of u in the outward normal direction n on the boundary.

Another feature of variational formulations is that the test function v is required to vanish on the parts of the boundary where the solution u is known (the book [24] explains in detail why this requirement is necessary). In the present problem, this means that $v = 0$ on the whole boundary $\partial\Omega$. The second term on the right-hand side of (2.5) therefore vanishes. From (2.4) and (2.5) it follows that

$$\int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x = \int_\Omega fv \,\mathrm{d}x\,. \tag{2.6}$$

If we require that this equation holds for all test functions v in some suitable space $\hat{V}$, the so-called *test space*, we obtain a well-defined mathematical problem that uniquely determines the solution u which lies in some (possi-

[1] `https://en.wikipedia.org/wiki/Integration_by_parts`

bly different) function space V, the so-called *trial space*. We refer to (2.6) as the *weak form* or *variational form* of the original boundary-value problem (2.1)–(2.2).

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

$$\int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x = \int_\Omega f v \,\mathrm{d}x \quad \forall v \in \hat{V}\,. \tag{2.7}$$

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

$$V = \{v \in H^1(\Omega) : v = u_{\mathrm{D}} \text{ on } \partial\Omega\},$$
$$\hat{V} = \{v \in H^1(\Omega) : v = 0 \text{ on } \partial\Omega\}\,.$$

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 (essentially meaning that the functions are continuous). The solution of the underlying PDE must lie in a function space where the derivatives are also continuous, but the Sobolev space $H^1(\Omega)$ allows functions with discontinuous derivatives. This weaker continuity requirement of u in the variational statement (2.7), as a result of the integration by parts, has great practical consequences when it comes to constructing finite element function spaces. In particular, it allows the use of piecewise polynomial function spaces; i.e., function spaces constructed by stitching together polynomial functions on simple domains such as intervals, triangles, or tetrahedrons.

The variational problem (2.7) is a *continuous problem*: it defines the solution u in the infinite-dimensional function space V. The finite element method for the Poisson equation finds an approximate solution of the variational problem (2.7) by replacing the infinite-dimensional function spaces V and $\hat{V}$ by *discrete* (finite-dimensional) trial and test spaces $V_h \subset V$ and $\hat{V}_h \subset \hat{V}$. The discrete variational problem reads: find $u_h \in V_h \subset V$ such that

$$\int_\Omega \nabla u_h \cdot \nabla v \,\mathrm{d}x = \int_\Omega f v \,\mathrm{d}x \quad \forall v \in \hat{V}_h \subset \hat{V}\,. \tag{2.8}$$

This variational problem, together with a suitable definition of the function spaces V_h and $\hat{V}_h$, uniquely define our approximate numerical solution of Poisson's equation (2.1). Note that the boundary conditions are encoded as part of the trial and test spaces. The mathematical framework may seem complicated at first glance, but the good news is that the finite element variational problem (2.8) looks the same as the continuous variational problem (2.7), and FEniCS can automatically solve variational problems like (2.8)!

What we mean by the notation u and V

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 drop the subscript $_h$ and use u for the solution of the discrete problem. We will use u_{e} for the exact solution of the continuous problem, *if* we need to explicitly distinguish between the two. Similarly, we will let V denote the discrete finite element function space in which we seek our solution.

2.1.2 Abstract finite element variational formulation

It turns out to be convenient to introduce the following canonical notation for variational problems: find $u \in V$ such that

$$a(u,v) = L(v) \quad \forall v \in \hat{V}. \tag{2.9}$$

For the Poisson equation, we have:

$$a(u,v) = \int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x, \tag{2.10}$$

$$L(v) = \int_\Omega f v \,\mathrm{d}x\,. \tag{2.11}$$

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 can then be expressed directly in our FEniCS programs.

To solve a linear PDE in FEniCS, such as the Poisson equation, a user thus needs to perform only two steps:

- Choose the finite element spaces V and $\hat{V}$ by specifying the domain (the mesh) and the type of function space (polynomial degree and type).
- Express the PDE as a (discrete) variational problem: find $u \in V$ such that $a(u,v) = L(v)$ for all $v \in \hat{V}$.

2.1.3 Choosing a test problem

The Poisson problem (2.1)–(2.2) has so far featured a general domain Ω and general functions u_{D} for the boundary conditions and f for the right-hand side. For our first implementation we will need to make specific choices for Ω, u_{D}, and f. It will be wise to construct a problem with a known analytical solution so that we can easily check that the computed solution is correct. Solutions that are lower-order polynomials are primary candidates. Standard finite element function spaces of degree r will exactly reproduce polynomials of degree r. And piecewise linear elements ($r = 1$) are able to exactly reproduce a quadratic polynomial on a uniformly partitioned mesh. This important result can be used to verify our implementation. We just manufacture some quadratic function in 2D as the exact solution, say

$$u_{\text{e}}(x, y) = 1 + x^2 + 2y^2 . \tag{2.12}$$

By inserting (2.12) into the Poisson equation (2.1), we find that $u_{\text{e}}(x, y)$ is a solution if

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

regardless of the shape of the domain as long as u_{e} is prescribed along the boundary. We choose here, for simplicity, the domain to be the unit square,

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

This simple but very powerful method for constructing test problems is called the *method of manufactured solutions*: pick a simple expression for the exact solution, plug it into the equation to obtain the right-hand side (source term f), then solve the equation with this right-hand side and using the exact solution as a boundary condition, and try to reproduce the exact solution.

Tip: Try to verify your code with exact numerical solutions!

A common approach to testing the implementation of a numerical method is to compare the numerical solution with an exact analytical solution of the test problem and conclude that the program works if the error is "small enough". Unfortunately, it is impossible to tell if an error of size 10^{-5} on a 20×20 mesh of linear elements is the expected (in)accuracy of the numerical approximation or if the error also contains the effect of a bug in the code. All we usually know about the numerical error is its *asymptotic properties*, for instance that it is proportional to h^2 if h is the size of a cell in the mesh. Then we compare the error on meshes with different h-values to see if the asymptotic behavior is correct. This is a very powerful verification technique and is explained

in detail in Section 5.5.4. However, if we have a test problem for which we know that there should be no approximation errors, we know that the analytical solution of the PDE problem should be reproduced to machine precision by the program. That is why we emphasize this kind of test problems throughout this tutorial. Typically, elements of degree r can reproduce polynomials of degree r exactly, so this is the starting point for constructing a solution without numerical approximation errors.

2.2 FEniCS implementation

2.2.1 The complete program

A FEniCS program for solving our test problem for the Poisson equation in 2D with the given choices of Ω, u_{D}, and f may look as follows:

```
from fenics import *

# Create mesh and define function space
mesh = UnitSquareMesh(8, 8)
V = FunctionSpace(mesh, 'P', 1)

# Define boundary condition
u_D = Expression('1 + x[0]*x[0] + 2*x[1]*x[1]', degree=2)

def boundary(x, on_boundary):
    return on_boundary

bc = DirichletBC(V, u_D, boundary)

# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)
f = Constant(-6.0)
a = dot(grad(u), 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)

# Save solution to file in VTK format
vtkfile = File('poisson/solution.pvd')
vtkfile << u
```

```
# Compute error in L2 norm
error_L2 = errornorm(u_D, u, 'L2')

# Compute maximum error at vertices
vertex_values_u_D = u_D.compute_vertex_values(mesh)
vertex_values_u = u.compute_vertex_values(mesh)
import numpy as np
error_max = np.max(np.abs(vertex_values_u_D - vertex_values_u))

# Print errors
print('error_L2  =', error_L2)
print('error_max =', error_max)

# Hold plot
interactive()
```

This example program can be found in the file `ft01_poisson.py`.

2.2.2 Running the program

The FEniCS program must be available in a plain text file, written with a text editor such as Atom, Sublime Text, Emacs, Vim, or similar. There are several ways to run a Python program like `ft01_poisson.py`:

- Use a terminal window.
- Use an integrated development environment (IDE), e.g., Spyder.
- Use a Jupyter notebook.

Terminal window. Open a terminal window, move to the directory containing the program and type the following command:

Terminal

```
Terminal> python ft01_poisson.py
```

Note that this command must be run in a FEniCS-enabled terminal. For users of the FEniCS Docker containers, this means that you must type this command after you have started a FEniCS session using `fenicsproject run` or `fenicsproject start`.

When running the above command, FEniCS will run the program to compute the approximate solution u. The approximate solution u will be compared to the exact solution $u_{\mathrm{e}} = u_{\mathrm{D}}$ and the error in the L^2 and maximum norms will be printed. Since we know that our approximate solution should reproduce the exact solution to within machine precision, this error should be small, something on the order of 10^{-15}. If plotting is enabled in your FEniCS installation, then a window with a simple plot of the solution will appear as in Figure 2.1.

Spyder. Many prefer to work in an integrated development environment that provides an editor for programming, a window for executing code, a window for inspecting objects, etc. Just open the file `ft01_poisson.py` and press the play button to run it. We refer to the Spyder tutorial to learn more about working in the Spyder environment. Spyder is highly recommended if you are used to working in the *graphical* MATLAB environment.

Jupyter notebooks. Notebooks make it possible to mix text and executable code in the same document, but you can also just use it to run programs in a web browser. Run the command `jupyter notebook` from a terminal window, find the **New** pulldown menu in the upper right corner of the GUI, choose a new notebook in Python 2 or 3, write `%load ft01_poisson.py` in the blank cell of this notebook, then press Shift+Enter to execute the cell. The file `ft01_poisson.py` will then be loaded into the notebook. Re-execute the cell (Shift+Enter) to run the program. You may divide the entire program into several cells to examine intermediate results: place the cursor where you want to split the cell and choose **Edit - Split Cell**. For users of the FEniCS Docker images, run the `fenicsproject notebook` command and follow the instructions. To enable plotting, make sure to run the command `%matplotlib inline` inside the notebook.

2.3 Dissection of the program

We shall now dissect our FEniCS program in detail. The listed FEniCS program defines a finite element mesh, a finite element function space V on this mesh, boundary conditions for u (the function u_{D}), and the bilinear and linear forms $a(u,v)$ and $L(v)$. Thereafter, the solution u is computed. At the end of the program, we compare the numerical and the exact solutions. We also plot the solution using the `plot` command and save the solution to a file for external postprocessing.

2.3.1 The important first line

The first line in the program,

```
from fenics import *
```

imports the key classes `UnitSquareMesh`, `FunctionSpace`, `Function`, and so forth, from the FEniCS library. All FEniCS programs for solving PDEs by the finite element method normally start with this line.

2.3.2 Generating simple meshes

The statement

```
mesh = UnitSquareMesh(8, 8)
```

defines a uniform finite element mesh over the unit square $[0,1] \times [0,1]$. The mesh consists of *cells*, which in 2D are triangles with straight sides. The parameters 8 and 8 specify that the square should be divided into 8×8 rectangles, each divided into a pair of triangles. The total number of triangles (cells) thus becomes 128. The total number of vertices in the mesh is $9 \cdot 9 = 81$. In later chapters, you will learn how to generate more complex meshes.

2.3.3 Defining the finite element function space

Once the mesh has been created, we can create a finite element function space `V`:

```
V = FunctionSpace(mesh, 'P', 1)
```

The second argument `'P'` specifies the type of element. The type of element here is P, implying the standard Lagrange family of elements. You may also use `'Lagrange'` to specify this type of element. FEniCS supports all simplex element families and the notation defined in the Periodic Table of the Finite Elements[2] [2].

The third argument `1` specifies the degree of the finite element. In this case, the standard P_1 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 solution u will be continuous across elements and linearly varying in x and y inside each element. Higher-degree polynomial approximations over each cell are trivially obtained by increasing the third parameter to `FunctionSpace`, which will then generate function spaces of type P_2, P_3, and so forth. Changing the second parameter to `'DP'` creates a function space for discontinuous Galerkin methods.

2.3.4 Defining the trial and test functions

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

[2] `https://www.femtable.org`

space, so it is sufficient to work with one common space `V` for both the trial and test functions in the program:

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

2.3.5 Defining the boundary conditions

The next step is to specify the boundary condition: $u = u_{\mathrm{D}}$ on $\partial\Omega$. This is done by

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

where `u_D` is an expression defining the solution values on the boundary, and `boundary` is a function (or object) defining which points belong to the boundary.

Boundary conditions of the type $u = u_{\mathrm{D}}$ are known as *Dirichlet conditions*. For the present finite element method for the Poisson problem, they are also called *essential boundary conditions*, as they need to be imposed explicitly as part of the trial space (in contrast to being defined implicitly as part of the variational formulation). Naturally, the FEniCS class used to define Dirichlet boundary conditions is named `DirichletBC`.

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

```
u_D = Expression(formula, degree=1)
```

where `formula` is a string containing a mathematical expression. The formula must be written with C++ syntax and is automatically turned into an efficient, compiled C++ function.

Expressions and accuracy

When defining an `Expression`, the second argument `degree` is a parameter that specifies how the expression should be treated in computations. On each local element, FEniCS will interpolate the expression into a finite element space of the specified degree. To obtain optimal (order of) accuracy in computations, it is usually a good choice to use the same degree as for the space V that is used for the trial and test functions. However, if an `Expression` is used to represent an exact solution which is used to evaluate the accuracy of a computed solution, a higher degree must be used for the expression (one or two degrees higher).

The expression may depend on the variables `x[0]` and `x[1]` corresponding to the x and y coordinates. In 3D, the expression may also depend on the variable `x[2]` corresponding to the z coordinate. With our choice of $u_{\mathrm{D}}(x,y) = 1 + x^2 + 2y^2$, the formula string can be written as `1 + x[0]*x[0] + 2*x[1]*x[1]`:

```
u_D = Expression('1 + x[0]*x[0] + 2*x[1]*x[1]', degree=2)
```

We set the degree to 2 so that `u_D` may represent the exact quadratic solution to our test problem.

String expressions must have valid C++ syntax!

The string argument to an `Expression` object must obey C++ syntax. Most Python syntax for mathematical expressions is 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 the documentation of `cmath` for more information on the various functions.

If/else tests are possible using the C syntax for inline branching. The function

$$f(x,y) = \begin{cases} x^2, & x, y \geq 0, \\ 2, & \text{otherwise}, \end{cases}$$

is implemented as

```
f = Expression('x[0]>=0 && x[1]>=0 ? pow(x[0], 2) : 2', degree=2)
```

Parameters in expression strings are allowed, but must be initialized via keyword arguments when creating the `Expression` object. For example, the function $f(x) = e^{-\kappa\pi^2 t}\sin(\pi k x)$ can be coded as

```
f = Expression('exp(-kappa*pow(pi, 2)*t)*sin(pi*k*x[0])', degree=2,
               kappa=1.0, t=0, k=4)
```

At any time, parameters can be updated:

```
f.t += dt
f.k = 10
```

The function `boundary` specifies which points that belong to the part of the boundary where the boundary condition should be applied:

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

A function like `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 `True` if `x` is on the physical boundary of the mesh, so 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 `boundary` function will be called for every discrete point in the mesh, which means that we may define boundaries where u is also known inside the domain, if desired.

One way to think about the specification of boundaries in FEniCS is that FEniCS will ask you (or rather the function `boundary` which you have implemented) whether or not a specific point `x` is part of the boundary. FEniCS already knows whether the point belongs to the *actual* boundary (the mathematical boundary of the domain) and kindly shares this information with you in the variable `on_boundary`. You may choose to use this information (as we do here), or ignore it completely.

The argument `on_boundary` may also be omitted, but in that case we need to test on the value of the coordinates in `x`:

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

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, either explicitly

```
tol = 1E-14
def boundary(x):
    return abs(x[0]) < tol or abs(x[1]) < tol \
        or abs(x[0] - 1) < tol or abs(x[1] - 1) < tol
```

or using the `near` command in FEniCS:

```
def boundary(x):
    return near(x[0], 0, tol) or near(x[1], 0, tol) \
        or near(x[0], 1, tol) or near(x[1], 1, tol)
```

Never use == for comparing real numbers!

A comparison like `x[0] == 1` should never be used if `x[0]` is a real number, because rounding errors in `x[0]` may make the test fail even when it is mathematically correct. Consider the following calculations in Python:

```
>>> 0.1 + 0.2 == 0.3
False
>>> 0.1 + 0.2
0.30000000000000004
```

Comparison of real numbers needs to be made with tolerances! The values of the tolerances depend on the size of the numbers involved in arithmetic operations:

```
>>> abs(0.1 + 0.2 - 0.3)
5.551115123125783e-17
>>> abs(1.1 + 1.2 - 2.3)
0.0
>>> abs(10.1 + 10.2 - 20.3)
3.552713678800501e-15
>>> abs(100.1 + 100.2 - 200.3)
0.0
>>> abs(1000.1 + 1000.2 - 2000.3)
2.2737367544323206e-13
>>> abs(10000.1 + 10000.2 - 20000.3)
3.637978807091713e-12
```

For numbers of unit size, tolerances as low as $3 \cdot 10^{-16}$ can be used (in fact, this tolerance is known as the constant `DOLFIN_EPS` in FEniCS). Otherwise, an appropriately scaled tolerance must be used.

2.3.6 Defining the source term

Before defining the bilinear and linear forms $a(u,v)$ and $L(v)$ we have to specify the source term f:

```
f = Expression('-6', degree=0)
```

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

```
f = Constant(-6)
```

2.3.7 Defining the variational problem

We now have all the ingredients we need to define the variational problem:

```
a = dot(grad(u), 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 and solve complicated PDE problems. The language used to express weak forms is called UFL (Unified Form Language) [1, 26] and is an integral part of FEniCS.

Expressing inner products

The inner product $\int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x$ can be expressed in various ways in FEniCS. Above, we have used the notation `dot(grad(u), grad(v))*dx`. The dot product in FEniCS/UFL computes the sum (contraction) over the last index of the first factor and the first index of the second factor. In this case, both factors are tensors of rank one (vectors) and so the sum is just over the one single index of both ∇u and ∇v. To compute an inner product of matrices (with two indices), one must instead of `dot` use the function `inner`. For vectors, `dot` and `inner` are equivalent.

2.3.8 Forming and solving the linear system

Having defined the finite element variational problem and boundary condition, we can now ask FEniCS to compute the solution:

```
u = Function(V)
solve(a == L, 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; i.e., the computed finite element function u. This redefinition of the variable `u` is possible in Python and is often used in FEniCS applications for linear problems. 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.

2.3.9 Plotting the solution using the `plot` command

Once the solution has been computed, it can be visualized by the `plot` command:

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

```
interactive()
```

Note the call to the function `interactive` after the `plot` commands. This call makes it possible to interact with the plots (rotating and zooming). The call to `interactive` is usually placed at the end of a program that creates plots. Figure 2.1 displays the two plots.

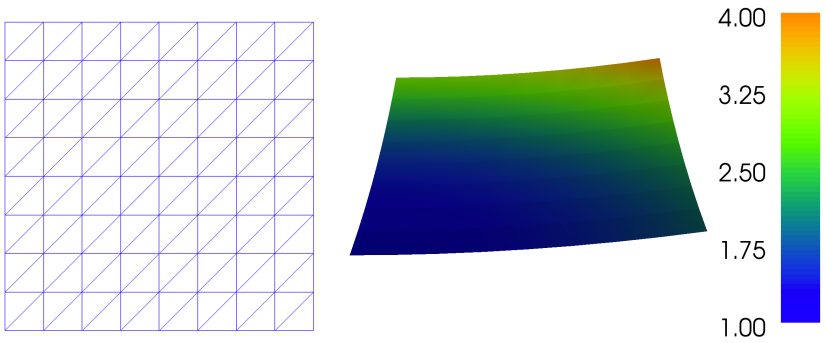


Fig. 2.1 Plot of the mesh and the solution for the Poisson problem created using the built-in FEniCS visualization tool (`plot` command).

The `plot` command is useful for debugging and initial scientific investigations. More advanced visualizations are better created by exporting the solution to a file and using an advanced visualization tool like ParaView, as explained in the next section.

By clicking the left mouse button in the plot window, you may rotate the solution, while the right mouse button is used for zooming. Point the mouse to the `Help` text in the lower left corner to display a list of all available shortcut commands. The help menu may alternatively be activated by typing **h** in the plot window. The `plot` command also accepts a number of additional arguments, such as for example setting the title of the plot window:

```
plot(u, title='Finite element solution')
plot(mesh, title='Finite element mesh')
```

For detailed documentation, either run the command `help(plot)` in Python or `pydoc fenics.plot` from a terminal window.

Built-in plotting on Mac OS X and in Docker

The built-in plotting in FEniCS may not work as expected when either running on Mac OS X or when running inside a FEniCS Docker container. FEniCS supports plotting using the `plot` command on Mac OS X. However, the keyboard shortcuts may fail to work. When running

inside a Docker container, plotting is not supported since Docker does not interact with your windowing system. For Docker users who need plotting, it is recommended to either work within a Jupyter/FEniCS notebook (command `fenicsproject notebook`) or rely on ParaView or other external tools for visualization.

2.3.10 Plotting the solution using ParaView

The simple `plot` command is useful for quick visualizations, but for more advanced visualizations an external tool must be used. In this section we demonstrate how to visualize solutions in ParaView. ParaView[3] is a powerful tool for visualizing scalar and vector fields, including those computed by FEniCS.

The first step is to export the solution in VTK format:

```
vtkfile = File('poisson/solution.pvd')
vtkfile << u
```

The following steps demonstrate how to create a plot of the solution of our Poisson problem in ParaView. The resulting plot is shown in Figure 2.2.

1. Start the ParaView application.
2. Click **File–Open...** in the top menu and navigate to the directory containing the exported solution. This should be inside a subdirectory named `poisson` below the directory where the FEniCS Python program was started. Select the file named `solution.pvd` and then click **OK**.
3. Click **Apply** in the Properties pane on the left. This will bring up a plot of the solution.
4. To make a 3D plot of the solution, we will make use of one of ParaView's many *filters*. Click **Filters–Alphabetical–Warp By Scalar** in the top menu and then **Apply** in the Properties pane on the left. This create an elevated surface with the height determined by the solution value.
5. To show the original plot below the elevated surface, click the little eye icon to the left of `solution.pvd` in the Pipeline Browser pane on the left. Also click the little 2D button at the top of the plot window to change the visualization to 3D. This lets you interact with the plot by rotating (left mouse button) and zooming (Ctrl + left mouse button).
6. To show the finite element mesh, click on `solution.pvd` in the Pipeline Browser, navigate to **Representation** in the Properties pane, and select **Surface With Edges**. This should make the finite element mesh visible.
7. To change the aspect ratio of the plot, click on **WarpByScalar1** in

the Pipeline Browser and navigate to **Scale Factor** in the Properties pane. Change the value to 0.2 and click **Apply**. This will change the scale of the

[3] `http://www.paraview.org`

warped plot. We also unclick **Orientation Axis Visibility** at the bottom of the Properties pane to remove the little 3D axes in the lower left corner of the plot window. You should now see something that resembles the plot in Figure 2.2.

1. Finally, to export the visualization to a file, click **File–Save Screenshot...** and select a suitable file name such as `poisson.png`.

For more information, we refer to The ParaView Guide [30] (free PDF available), the ParaView tutorial[4], and the instruction video Introduction to ParaView[5].

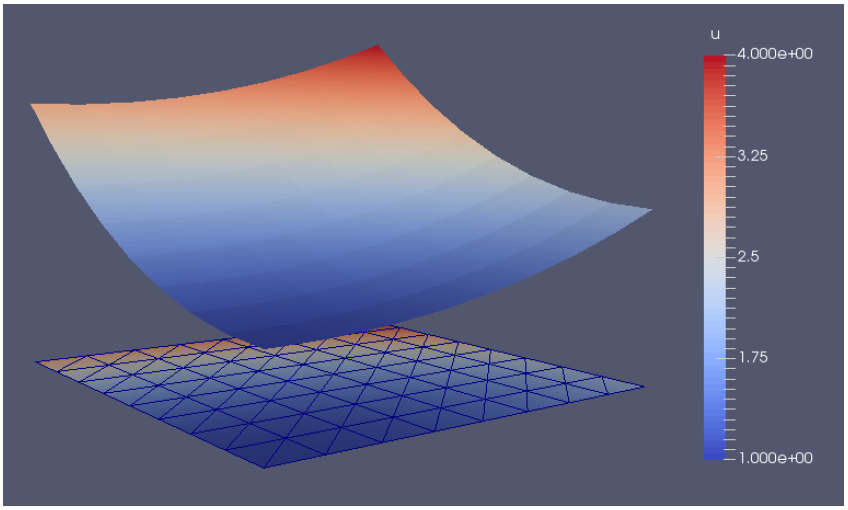


Fig. 2.2 Plot of the mesh and the solution for the Poisson problem created using ParaView.

2.3.11 Computing the error

Finally, we compute the error to check the accuracy of the solution. We do this by comparing the finite element solution `u` with the exact solution, which in this example happens to be the same as the expression `u_D` used to set the boundary conditions. We compute the error in two different ways. First, we compute the L^2 norm of the error, defined by

[4] `http://www.paraview.org/Wiki/The_ParaView_Tutorial`

[5] `https://vimeo.com/34037236`

$$E = \sqrt{\int_\Omega (u_{\mathrm{D}} - u)^2 \,\mathrm{d}x}\,.$$

Since the exact solution is quadratic and the finite element solution is piecewise linear, this error will be nonzero. To compute this error in FEniCS, we simply write

```
error_L2 = errornorm(u_D, u, 'L2')
```

The `errornorm` function can also compute other error norms such as the H^1 norm. Type `pydoc fenics.errornorm` in a terminal window for details.

We also compute the maximum value of the error at all the vertices of the finite element mesh. As mentioned above, we expect this error to be zero to within machine precision for this particular example. To compute the error at the vertices, we first ask FEniCS to compute the value of both `u_D` and `u` at all vertices, and then subtract the results:

```
vertex_values_u_D = u_D.compute_vertex_values(mesh)
vertex_values_u = u.compute_vertex_values(mesh)
import numpy as np
error_max = np.max(np.abs(vertex_values_u_D - vertex_values_u))
```

We have here used the maximum and absolute value functions from `numpy`, because these are much more efficient for large arrays (a factor of 30) than Python's built-in `max` and `abs` functions.

How to check that the error vanishes

With inexact (floating point) arithmetic, the maximum error at the vertices is not zero, but should be a small number. The machine precision is about 10^{-16}, but in finite element calculations, rounding errors of this size may accumulate, to produce an error larger than 10^{-16}. Experiments show that increasing the number of elements and increasing the degree of the finite element polynomials increases the error. For a mesh with $2 \times (20 \times 20)$ cubic Lagrange elements (degree 3) the error is about $2 \cdot 10^{-12}$, while for 128 linear elements the error is about $2 \cdot 10^{-15}$.

2.3.12 Examining degrees of freedom and vertex values

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

$$u = \sum_{j=1}^{N} U_j \phi_j\,. \tag{2.13}$$

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 values $U_1,\ldots,U_N$. The values $U_1,\ldots,U_N$ are known as the *degrees of freedom* ("dofs") or *nodal values* of u. For Lagrange elements (and many other element types) U_j is simply the value of u at the node with global number j. The locations of the nodes and cell vertices coincide for linear Lagrange elements, while for higher-order elements there are additional nodes associated with the facets, edges and sometimes also the interior of cells.

Having `u` represented as a `Function` object, we can either evaluate `u(x)` at any point `x` in the mesh (expensive operation!), or we can grab all the degrees of freedom in the vector U directly by

```
nodal_values_u = u.vector()
```

The result is a `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:

```
array_u = nodal_values_u.array()
```

With `numpy` arrays we can write MATLAB-like code to analyze the data. Indexing is done with square brackets: `array_u[j]`, where the index `j` always starts at `0`. If the solution is computed with piecewise linear Lagrange elements (P_1), then the size of the array `array_u` is equal to the number of vertices, and each `array_u[j]` is the value at some vertex in the mesh. However, the degrees of freedom are not necessarily numbered in the same way as the vertices of the mesh. (This is discussed in some detail in Section 5.4.1). If we therefore want to know the values at the vertices, we need to call the function `u.compute_vertex_values`. This function returns the values at all the vertices of the mesh as a `numpy` array with the same numbering as for the vertices of the mesh, for example:

```
vertex_values_u = u.compute_vertex_values()
```

Note that for P_1 elements, the arrays `array_u` and `vertex_values_u` have the same lengths and contain the same values, albeit in different order.

2.4 Deflection of a membrane

Our first FEniCS program for the Poisson equation targeted a simple test problem where we could easily verify the implementation. We now turn our attention to a physically more relevant problem with solutions of somewhat more exciting shape.

We want to compute the deflection $D(x,y)$ of a two-dimensional, circular membrane of radius R, subject to a load p over the membrane. The appropriate PDE model is

$$-T\nabla^2 D = p \quad \text{in } \Omega = \{(x,y)\,|\,x^2+y^2 \leq R\}\,. \tag{2.14}$$

Here, T is the tension in the membrane (constant), and p is the external pressure load. The boundary of the membrane has no deflection, implying $D=0$ as a boundary condition. A localized load can be modeled as a Gaussian function:

$$p(x,y) = \frac{A}{2\pi\sigma}\exp\left(-\frac{1}{2}\left(\frac{x-x_0}{\sigma}\right)^2 - \frac{1}{2}\left(\frac{y-y_0}{\sigma}\right)^2\right). \tag{2.15}$$

The parameter A is the amplitude of the pressure, (x_0,y_0) the localization of the maximum point of the load, and σ the "width" of p. We will take the center (x_0,y_0) of the pressure to be $(0,R_0)$ for some $0<R_0<R$.

2.4.1 Scaling the equation

There are many physical parameters in this problem, and we can benefit from grouping them by means of scaling. Let us introduce dimensionless coordinates $\bar{x}=x/R$, $\bar{y}=y/R$, and a dimensionless deflection $w=D/D_c$, where D_c is a characteristic size of the deflection. Introducing $\bar{R}_0=R_0/R$, we obtain

$$-\frac{\partial^2 w}{\partial\bar{x}^2} - \frac{\partial^2 w}{\partial\bar{y}^2} = \alpha\exp\left(-\beta^2(\bar{x}^2+(\bar{y}-\bar{R}_0)^2)\right) \text{ for } \bar{x}^2+\bar{y}^2<1,$$

where

$$\alpha = \frac{R^2A}{2\pi TD_c\sigma}, \quad \beta = \frac{R}{\sqrt{2}\sigma}\,.$$

With an appropriate scaling, w and its derivatives are of size unity, so the left-hand side of the scaled PDE is about unity in size, while the right-hand side has α as its characteristic size. This suggest choosing α to be unity, or around unity. We shall in this particular case choose $\alpha=4$. (One can also find the analytical solution in scaled coordinates and show that the maximum deflection $D(0,0)$ is D_c if we choose $\alpha=4$ to determine D_c.) With $D_c=AR^2/(8\pi\sigma T)$ and dropping the bars we obtain the scaled problem

$$-\nabla^2 w = 4\exp\left(-\beta^2(x^2+(y-R_0)^2)\right), \tag{2.16}$$

to be solved over the unit disc with $w=0$ on the boundary. Now there are only two parameters to vary: the dimensionless extent of the pressure, β, and

the localization of the pressure peak, $R_0 \in [0,1]$. As $\beta \to 0$, the solution will approach the special case $w = 1 - x^2 - y^2$.

Given a computed scaled solution w, the physical deflection can be computed by

$$D = \frac{AR^2}{8\pi\sigma T} w\,.$$

Just a few modifications are necessary to our previous program to solve this new problem.

2.4.2 Defining the mesh

A mesh over the unit disk can be created by the `mshr` tool in FEniCS:

```
from mshr import *
domain = Circle(Point(0, 0), 1)
mesh = generate_mesh(domain, 64)
```

The `Circle` shape from `mshr` takes the center and radius of the circle as arguments. The second argument to the `generate_mesh` function specifies the desired mesh resolution. The cell size will be (approximately) equal to the diameter of the domain divided by the resolution.

2.4.3 Defining the load

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

```
beta = 8
R0 = 0.6
p = Expression('4*exp(-pow(beta, 2)*(pow(x[0], 2) + pow(x[1] - R0, 2)))',
               degree=1, beta=beta, R0=R0)
```

The coordinates in `Expression` objects are always an array `x` with components `x[0]`, `x[1]`, and `x[2]`, corresponding to x, y, and z. 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

```
p.beta = 12
p.R0 = 0.3
```

2.4.4 Defining the variational problem

The variational problem and the boundary conditions are the same as in our first Poisson problem, but we may introduce `w` instead of `u` as primary unknown and `p` instead of `f` as right-hand side function:

```
w = TrialFunction(V)
v = TestFunction(V)
a = dot(grad(w), grad(v))*dx
L = p*v*dx

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

2.4.5 Plotting the solution

It is of interest to visualize the pressure p along with the deflection w so that we may examine the membrane's response to the pressure. 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 are calculated from p. That is, we interpolate p to the function space V:

```
p = interpolate(p, V)
```

Note that the assignment to `p` destroys the previous `Expression` object `p`, 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`. The two functions `w` and `p` may be plotted using the built-in plot command:

```
plot(w, title='Deflection')
plot(p, title='Load')
```

As before, we also export the solutions in VTK format for visualization in ParaView:

```
vtkfile_w = File('poisson_membrane/deflection.pvd')
vtkfile_w << w
vtkfile_p = File('poisson_membrane/load.pvd')
vtkfile_p << p
```

Figure 2.3 shows a visualization of the deflection `w` and the load `p` created with ParaView.

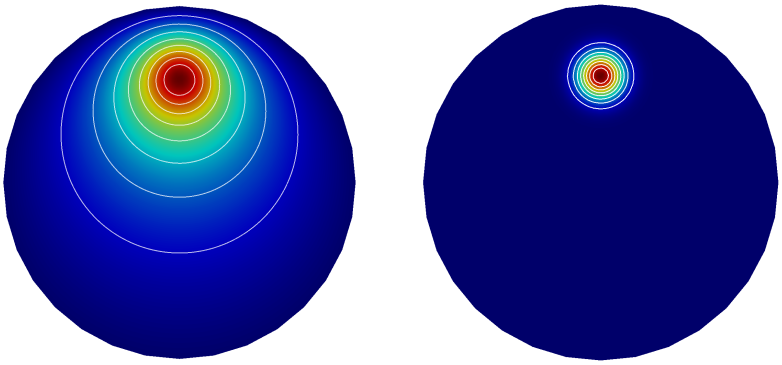


Fig. 2.3 Plot of the deflection (left) and load (right) for the membrane problem created using ParaView. The plot uses 10 equispaced isolines for the solution values and the optional *jet* colormap.

2.4.6 Making curve plots through the domain

Another way to compare the deflection and the load is to make a curve plot along the line $x = 0$. This is just a matter of defining a set of points along the y-axis and evaluating the finite element functions `w` and `p` at these points:

```
# Curve plot along x = 0 comparing p and w
import numpy as np
import matplotlib.pyplot as plt
tol = 0.001  # avoid hitting points outside the domain
y = np.linspace(-1 + tol, 1 - tol, 101)
points = [(0, y_) for y_ in y]  # 2D points
w_line = np.array([w(point) for point in points])
p_line = np.array([p(point) for point in points])
plt.plot(y, 50*w_line, 'k', linewidth=2)  # magnify w
plt.plot(y, p_line, 'b--', linewidth=2)
plt.grid(True)
plt.xlabel('$y$')
plt.legend(['Deflection ($\\times 50$)', 'Load'], loc='upper left')
plt.savefig('poisson_membrane/curves.pdf')
plt.savefig('poisson_membrane/curves.png')
```

This example program can be found in the file `ft02_poisson_membrane.py`.

The resulting curve plot is shown in Figure 2.4. The localized input (p) is heavily damped and smoothed in the output (w). This reflects a typical property of the Poisson equation.

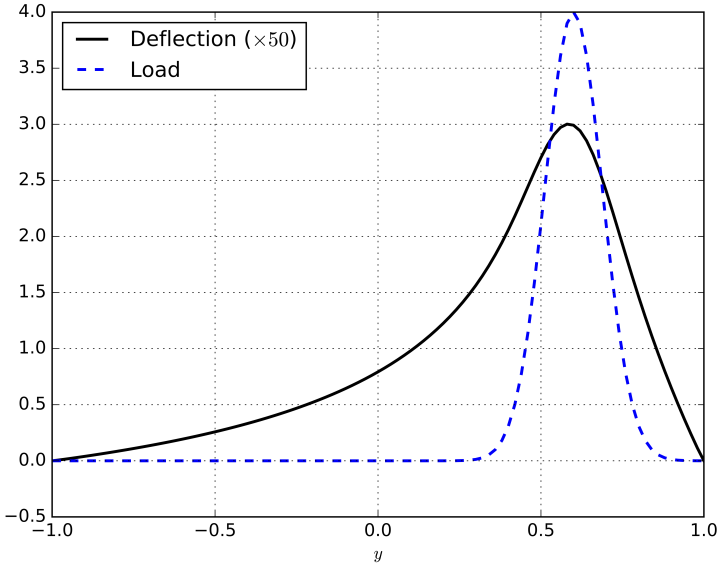


Fig. 2.4 Plot of the deflection and load for the membrane problem created using Matplotlib and sampling of the two functions along the y-axsis.

Chapter 3
A Gallery of finite element solvers

The goal of this chapter is to demonstrate how a range of important PDEs from science and engineering can be quickly solved with a few lines of FEniCS code. We start with the heat equation and continue with a nonlinear Poisson equation, the equations for linear elasticity, the Navier–Stokes equations, and finally look at how to solve systems of nonlinear advection–diffusion–reaction equations. These problems illustrate how to solve time-dependent problems, nonlinear problems, vector-valued problems, and systems of PDEs. For each problem, we derive the variational formulation and express the problem in Python in a way that closely resembles the mathematics.

3.1 The heat equation

As a first extension of the Poisson problem from the previous chapter, we consider the time-dependent heat equation, or the time-dependent diffusion equation. This is the natural extension of the Poisson equation describing the stationary distribution of heat in a body to a time-dependent problem.

We will see that by discretizing time into small time intervals and applying standard time-stepping methods, we can solve the heat equation by solving a sequence of variational problems, much like the one we encountered for the Poisson equation.

3.1.1 PDE problem

Our model problem for time-dependent PDEs reads

$$\frac{\partial u}{\partial t} = \nabla^2 u + f \quad \text{in } \Omega \times (0,T], \tag{3.1}$$

$$u = u_{\text{D}} \quad \text{on } \partial\Omega \times (0,T], \tag{3.2}$$

$$u = u_0 \quad \text{at } t = 0\,. \tag{3.3}$$

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_{D} may also vary with space and time. The initial condition u_0 is a function of space only.

3.1.2 Variational formulation

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 sequence of stationary problems, and then turn each stationary problem into a variational formulation.

Let superscript n denote a quantity at time t_n, where n is an integer counting time levels. For example, u^n means u at time level n. A finite difference discretization in time first consists of sampling the PDE at some time level, say t_{n+1}:

$$\left(\frac{\partial u}{\partial t}\right)^{n+1} = \nabla^2 u^{n+1} + f^{n+1}\,. \tag{3.4}$$

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

$$\left(\frac{\partial u}{\partial t}\right)^{n+1} \approx \frac{u^{n+1} - u^n}{\Delta t}, \tag{3.5}$$

where Δt is the time discretization parameter. Inserting (3.5) in (3.4) yields

$$\frac{u^{n+1} - u^n}{\Delta t} = \nabla^2 u^{n+1} + f^{n+1}\,. \tag{3.6}$$

This is our time-discrete version of the heat equation (3.1), a so-called *backward Euler* or *implicit Euler* discretization.

We may reorder (3.6) so that the left-hand side contains the terms with the unknown u^{n+1} and the right-hand side contains computed terms only. The result is a sequence of spatial (stationary) problems for u^{n+1}, assuming u^n is known from the previous time step:

$$u^0 = u_0, \tag{3.7}$$

$$u^{n+1} - \Delta t\nabla^2 u^{n+1} = u^n + \Delta t f^{n+1}, \quad n = 0, 1, 2, \ldots \tag{3.8}$$

Given u_0, we can solve for u^0, u^1, u^2, and so on.

An alternative to (3.8), which can be convenient in implementations, is to collect all terms on one side of the equality sign:

$$u^{n+1} - \Delta t\nabla^2 u^{n+1} - u^n - \Delta t f^{n+1} = 0, \quad n = 0, 1, 2, \ldots \tag{3.9}$$

We use a finite element method to solve (3.7) and either of the equations (3.8) or (3.9). 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^{n+1} (which is natural in the program), the resulting weak form arising from formulation (3.8) can be conveniently written in the standard notation:

$$a(u, v) = L_{n+1}(v),$$

where

$$a(u, v) = \int_\Omega (uv + \Delta t\nabla u \cdot \nabla v)\,\mathrm{d}x, \tag{3.10}$$

$$L_{n+1}(v) = \int_\Omega \left(u^n + \Delta t f^{n+1}\right) v\,\mathrm{d}x\,. \tag{3.11}$$

The alternative form (3.9) has an abstract formulation

$$F_{n+1}(u; v) = 0,$$

where

$$F_{n+1}(u; v) = \int_\Omega \left(uv + \Delta t\nabla u \cdot \nabla v - (u^n + \Delta t f^{n+1})v\right)\mathrm{d}x\,. \tag{3.12}$$

In addition to the variational problem to be solved in each time step, we also need to approximate the initial condition (3.7). This equation can also be turned into a variational problem:

$$a_0(u, v) = L_0(v),$$

with

$$a_0(u,v) = \int_\Omega uv \,\mathrm{d}x, \tag{3.13}$$

$$L_0(v) = \int_\Omega u_0 v \,\mathrm{d}x\,. \tag{3.14}$$

When solving this variational problem, u^0 becomes the L^2 projection of the given initial value u_0 into the finite element space. The alternative is to construct u^0 by just interpolating the initial value u_0; that is, if $u^0 = \sum_{j=1}^N U_j^0 \phi_j$, we simply set $U_j = u_0(x_j, y_j)$, 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 projection or interpolation. Both operations are easy to compute in FEniCS through a single statement, using either the `project` or `interpolate` function. The most common choice is `project`, which computes an approximation to u_0, but in some applications where we want to verify the code by reproducing exact solutions, one must use `interpolate` (and we use such a test problem here!).

In summary, we thus need to solve the following sequence of variational problems to compute the finite element solution to the heat equation: 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 $u^{n+1} \in V$ such that $a(u^{n+1}, v) = L_{n+1}(v)$ for all $v \in \hat{V}$, or alternatively, $F_{n+1}(u^{n+1}, v) = 0$ for all $v \in \hat{V}$, for $n = 0, 1, 2, \ldots$.

3.1.3 FEniCS implementation

Our program needs to implement the time-stepping manually, but can rely on FEniCS to easily compute a_0, L_0, a, and L (or F_{n+1}), and solve the linear systems for the unknowns.

Test problem 1: A known analytical solution. Just as for the Poisson problem from the previous chapter, we construct a test problem that makes it easy to determine if the calculations are correct. Since we know that our first-order time-stepping scheme is exact for linear functions, we create a test problem which has a linear variation in time. We combine this with a quadratic variation in space. We thus take

$$u = 1 + x^2 + \alpha y^2 + \beta t, \tag{3.15}$$

which yields a function whose computed values at the nodes will be exact, regardless of the size of the elements and Δt, as long as the mesh is uniformly partitioned. By inserting (3.15) into the heat equation (3.1), we find that the right-hand side f must be given by $f(x,y,t) = \beta - 2 - 2\alpha$. The boundary value is $u_{\mathrm{D}}(x,y,t) = 1 + x^2 + \alpha y^2 + \beta t$ and the initial value is $u_0(x,y) = 1 + x^2 + \alpha y^2$.

FEniCS implementation. A new programming issue is how to deal with functions that vary in space *and time*, such as the boundary condi-

tion $u_{\mathrm{D}}(x,y,t) = 1 + x^2 + \alpha y^2 + \beta t$. A natural solution is to use a FEniCS `Expression` with time t as a parameter, in addition to the parameters α and β:

```
alpha = 3; beta = 1.2
u_D = Expression('1 + x[0]*x[0] + alpha*x[1]*x[1] + beta*t',
                 degree=2, alpha=alpha, beta=beta, t=0)
```

This `Expression` uses the components of `x` as independent variables, while `alpha`, `beta`, and `t` are parameters. The time `t` can later be updated by

```
u_D.t = t
```

The essential boundary conditions, along the entire boundary in this case, are implemented in the same way as we have previously implemented the boundary conditions for the Poisson problem:

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

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

We shall use the variable `u` for the unknown u^{n+1} at the new time step and the variable `u_n` for u^n at the previous time step. The initial value of `u_n` can be computed by either projection or interpolation of u_0. Since we set `t = 0` for the boundary value `u_D`, we can use `u_D` to specify the initial condition:

```
u_n = project(u_D, V)
# or
u_n = interpolate(u_D, V)
```

Projecting versus interpolating the initial condition

To actually recover the exact solution (3.15) to machine precision, it is important to compute the discrete initial condition by interpolating u_0. This ensures that the degrees of freedom are exact (to machine precision) at $t = 0$. Projection results in approximate values at the nodes.

We may either define a or L according to the formulas above, or we may just define F and ask FEniCS to figure out which terms should go into the bilinear form a and which should go into the linear form L. The latter is convenient, especially in more complicated problems, so we illustrate that construction of a and L:

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

```
F = u*v*dx + dt*dot(grad(u), grad(v))*dx - (u_n + dt*f)*v*dx
a, L = lhs(F), rhs(F)
```

Finally, we perform the time-stepping in a loop:

```
u = Function(V)
t = 0
for n in range(num_steps):

    # Update current time
    t += dt
    u_D.t = t

    # Solve variational problem
    solve(a == L, u, bc)

    # Update previous solution
    u_n.assign(u)
```

In the last step of the time-stepping loop, we assign the values of the variable `u` (the new computed solution) to the variable `u_n` containing the values at the previous time step. This must be done using the `assign` member function. If we instead try to do `u_n = u`, we will set the `u_n` variable to be the *same* variable as `u` which is not what we want. (We need two variables, one for the values at the previous time step and one for the values at the current time step.)

Remember to update expression objects with the current time!

Inside the time loop, observe that `u_D.t` must be updated before the `solve` statement to enforce computation of Dirichlet conditions at the current time step. A Dirichlet condition defined in terms of an `Exression` looks up and applies the value of a parameter such as `t` when it gets evaluated and applied to the linear system.

The time-stepping loop above does not contain any comparison of the numerical and the exact solutions, which we must include in order to verify the implementation. As for the Poisson equation in Section 2.3, we compute the difference between the array of nodal values for `u` and the array of nodal values for the interpolated exact solution. This may be done as follows:

```
u_e = interpolate(u_D, V)
error = np.abs(u_e.vector().array() - u.vector().array()).max()
print('t = %.2f: error = %.3g' % (t, error))
```

For the Poisson example, we used the function `compute_vertex_values` to extract the function values at the vertices. Here we illustrate an alternative method to extract the vertex values, by calling the function `vector`, which

returns the vector of degrees of freedom. For a P_1 function space, this vector of degrees of freedom will be equal to the array of vertex values obtained by calling `compute_vertex_values`, albeit possibly in a different order.

The complete program for solving the heat equation goes as follows:

```
from fenics import *
import numpy as np

T = 2.0            # final time
num_steps = 10     # number of time steps
dt = T / num_steps # time step size
alpha = 3          # parameter alpha
beta = 1.2         # parameter beta

# Create mesh and define function space
nx = ny = 8
mesh = UnitSquareMesh(nx, ny)
V = FunctionSpace(mesh, 'P', 1)

# Define boundary condition
u_D = Expression('1 + x[0]*x[0] + alpha*x[1]*x[1] + beta*t',
                 degree=2, alpha=alpha, beta=beta, t=0)

def boundary(x, on_boundary):
    return on_boundary

bc = DirichletBC(V, u_D, boundary)

# Define initial value
u_n = interpolate(u_D, V)
#u_n = project(u_D, V)

# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)
f = Constant(beta - 2 - 2*alpha)

F = u*v*dx + dt*dot(grad(u), grad(v))*dx - (u_n + dt*f)*v*dx
a, L = lhs(F), rhs(F)

# Time-stepping
u = Function(V)
t = 0
for n in range(num_steps):

    # Update current time
    t += dt
    u_D.t = t

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

    # Plot solution
```

```
    plot(u)

    # Compute error at vertices
    u_e = interpolate(u_D, V)
    error = np.abs(u_e.vector().array() - u.vector().array()).max()
    print('t = %.2f: error = %.3g' % (t, error))

    # Update previous solution
    u_n.assign(u)

# Hold plot
interactive()
```

This example program can be found in the file `ft03_heat.py`.

Test problem 2: Diffusion of a Gaussian function. Let us now solve a more interesting test problem, namely the diffusion of a Gaussian hill. We take the initial value to be

$$u_0(x,y) = e^{-ax^2-ay^2}$$

for $a = 5$ on the domain $[-2,2] \times [2,2]$. For this problem we will use homogeneous Dirichlet boundary conditions ($u_{\mathrm{D}} = 0$).

FEniCS implementation. Which are the required changes to our previous program? One major change is that the domain is no longer a unit square. The new domain can be created easily in FEniCS using `RectangleMesh`:

```
nx = ny = 30
mesh = RectangleMesh(Point(-2, -2), Point(2, 2), nx, ny)
```

Note that we have used a much higher resolution than before to better resolve the features of the solution. We also need to redefine the initial condition and the boundary condition. Both are easily changed by defining a new `Expression` and by setting $u = 0$ on the boundary.

To be able to visualize the solution in an external program such as ParaView, we will save the solution to a file in VTK format in each time step. We do this by first creating a `File` with the suffix `.pvd`:

```
vtkfile = File('heat_gaussian/solution.pvd')
```

Inside the time loop, we may then append the solution values to this file:

```
vtkfile << (u, t)
```

This line is called in each time step, resulting in the creation of a new file with suffix `.vtu` containing all data for the time step (the mesh and the vertex values). The file `heat_gaussian/solution.pvd` will contain the time values and references to the `.vtu` file, which means that the `.pvd` file will be a single small file that points to a large number of `.vtu` files containing the actual data. Note that we choose to store the solution to a subdirectory named `heat_gaussian`. This is to avoid cluttering our source directory with

all the generated data files. One does not need to create the directory before running the program as it will be created automatically by FEniCS.

The complete program appears below.

```
from fenics import *
import time

T = 2.0            # final time
num_steps = 50     # number of time steps
dt = T / num_steps # time step size

# Create mesh and define function space
nx = ny = 30
mesh = RectangleMesh(Point(-2, -2), Point(2, 2), nx, ny)
V = FunctionSpace(mesh, 'P', 1)

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

bc = DirichletBC(V, Constant(0), boundary)

# Define initial value
u_0 = Expression('exp(-a*pow(x[0], 2) - a*pow(x[1], 2))',
                 degree=2, a=5)
u_n = interpolate(u_0, V)

# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)
f = Constant(0)

F = u*v*dx + dt*dot(grad(u), grad(v))*dx - (u_n + dt*f)*v*dx
a, L = lhs(F), rhs(F)

# Create VTK file for saving solution
vtkfile = File('heat_gaussian/solution.pvd')

# Time-stepping
u = Function(V)
t = 0
for n in range(num_steps):

    # Update current time
    t += dt

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

    # Save to file and plot solution
    vtkfile << (u, t)
    plot(u)

    # Update previous solution
```

```
    u_n.assign(u)

# Hold plot
interactive()
```

This example program can be found in the file `ft04_heat_gaussian.py`.

Visualization in ParaView. To visualize the diffusion of the Gaussian hill, start ParaView, choose **File–Open...**, open `heat_gaussian/solution.pvd`, and click **Apply** in the Properties pane. Click on the play button to display an animation of the solution. To save the animation to a file, click **File–Save Animation...** and save the file to a desired file format, for example AVI or Ogg/Theora. Once the animation has been saved to a file, you can play the animation offline using a player such as mplayer or VLC, or upload your animation to YouTube. Figure 3.1 shows a sequence of snapshots of the solution.

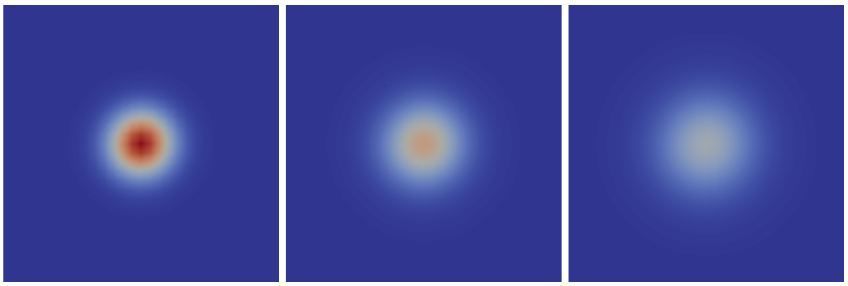


Fig. 3.1 A sequence of snapshots of the solution of the Gaussian hill problem created with ParaView.

3.2 A nonlinear Poisson equation

We shall now address how to solve nonlinear PDEs. We will see that nonlinear problems can be solved just as easily as linear problems in FEniCS, by simply defining a nonlinear variational problem and calling the `solve` function. When doing so, we will encounter a subtle difference in how the variational problem is defined.

3.2.1 PDE problem

As a model problem for the solution of nonlinear PDEs, we take the following nonlinear Poisson equation:

$$-\nabla\cdot(q(u)\nabla u) = f, \tag{3.16}$$

in Ω, with $u = u_{\mathrm{D}}$ on the boundary $\partial\Omega$. The coefficient $q = q(u)$ makes the equation nonlinear (unless $q(u)$ is constant in u).

3.2.2 Variational formulation

As usual, we multiply our PDE by a test function $v \in \hat{V}$, integrate over the domain, and integrate the second-order derivatives by parts. The boundary integral arising from integration by parts vanishes wherever we employ Dirichlet conditions. The resulting variational formulation of our model problem becomes: find $u \in V$ such that

$$F(u;v) = 0 \quad \forall v \in \hat{V}, \tag{3.17}$$

where

$$F(u;v) = \int_\Omega (q(u)\nabla u\cdot\nabla v - fv)\,\mathrm{d}x, \tag{3.18}$$

and

$$V = \{v \in H^1(\Omega) : v = u_{\mathrm{D}} \text{ on } \partial\Omega\},$$
$$\hat{V} = \{v \in H^1(\Omega) : v = 0 \text{ on } \partial\Omega\}.$$

The discrete problem arises as usual by restricting V and $\hat{V}$ to a pair of discrete spaces. As before, we omit any subscript on the discrete spaces and discrete solution. The discrete nonlinear problem is then written as: find $u \in V$ such that

$$F(u;v) = 0 \quad \forall v \in \hat{V}, \tag{3.19}$$

with $u = \sum_{j=1}^{N} U_j\phi_j$. Since F is nonlinear in u, the variational statement gives rise to a system of nonlinear algebraic equations in the unknowns $U_1,\ldots,U_N$.

3.2.3 FEniCS implementation

Test problem. To solve a test problem, we need to choose the right-hand side f, the coefficient $q(u)$ and the boundary value u_{D}. Previously, we have worked with manufactured solutions that can be reproduced without approximation errors. This is more difficult in nonlinear problems, and the algebra

is more tedious. However, we may utilize SymPy for symbolic computing and integrate such computations in the FEniCS solver. This allows us to easily experiment with different manufactured solutions. The forthcoming code with SymPy requires some basic familiarity with this package. In particular, we will use the SymPy functions `diff` for symbolic differentiation and `ccode` for C/C++ code generation.

We take $q(u) = 1 + u^2$ and define a two-dimensional manufactured solution that is linear in x and y:

```
# Warning: from fenics import * will import both ‘sym‘ and
# ‘q‘ from FEniCS. We therefore import FEniCS first and then
# overwrite these objects.
from fenics import *

def q(u):
    "Return nonlinear coefficient"
    return 1 + u**2

# Use SymPy to compute f from the manufactured solution u
import sympy as sym
x, y = sym.symbols('x[0], x[1]')
u = 1 + x + 2*y
f = - sym.diff(q(u)*sym.diff(u, x), x) - sym.diff(q(u)*sym.diff(u, y), y)
f = sym.simplify(f)
u_code = sym.printing.ccode(u)
f_code = sym.printing.ccode(f)
print('u =', u_code)
print('f =', f_code)
```

Define symbolic coordinates as required in `Expression` objects

Note that we would normally write `x, y = sym.symbols('x, y')`, but if we want the resulting expressions to have valid syntax for FEniCS `Expression` objects, we must use `x[0]` and `x[1]`. This is easily accomplished with `sympy` by defining the names of `x` and `y` as `x[0]` and `x[1]`: `x, y = sym.symbols('x[0], x[1]')`.

Turning the expressions for `u` and `f` into C or C++ syntax for FEniCS `Expression` objects needs two steps. First, we ask for the C code of the expressions:

```
u_code = sym.printing.ccode(u)
f_code = sym.printing.ccode(f)
```

In some cases, one will need to edit the result to match the required syntax of `Expression` objects, but not in this case. (The primary example is that `M_PI` for π in C/C++ must be replaced by `pi` for `Expression` objects.) In the present case, the output of `u_code` and `f_code` is

```
x[0] + 2*x[1] + 1
-10*x[0] - 20*x[1] - 10
```

After having defined the mesh, the function space, and the boundary, we define the boundary value `u_D` as

```
u_D = Expression(u_code, degree=1)
```

Similarly, we define the right-hand side function as

```
f = Expression(f_code, degree=1)
```

Name clash between FEniCS and program variables

In a program like the one above, strange errors may occur due to name clashes. If you define `sym` and `q` prior to doing `from fenics import *`, the latter statement will also import variables with the names `sym` and `q`, overwriting the objects you have previously defined! This may lead to strange errors. The safest solution is to do `import fenics` instead of `from fenics import *` and then prefix all FEniCS object names by `fenics`. The next best solution is to do `from fenics import *` first and then define your own variables that overwrite those imported from `fenics`. This is acceptable if we do not need `sym` and `q` from `fenics`.

FEniCS implementation. A solver for the nonlinear Poisson equation is as easy to implement as a solver for the linear Poisson equation. All we need to do is to state the formula for F and call `solve(F == 0, u, bc)` instead of `solve(a == L, u, bc)` as we did in the linear case. Here is a minimalistic code:

```
from fenics import *

def q(u):
    return 1 + u**2

mesh = UnitSquareMesh(8, 8)
V = FunctionSpace(mesh, 'P', 1)
u_D = Expression(u_code, degree=1)

def boundary(x, on_boundary):
    return on_boundary

bc = DirichletBC(V, u_D, boundary)

u = Function(V)
v = TestFunction(V)
f = Expression(f_code, degree=1)
F = q(u)*dot(grad(u), grad(v))*dx - f*v*dx
```

```
solve(F == 0, u, bc)
```

A complete version of this example program can be found in the file `ft05_poisson_nonlinear.py`.

The major difference from a linear problem is that the unknown function `u` in the variational form in the nonlinear case must be defined as a `Function`, not as a `TrialFunction`. In some sense this is a simplification from the linear case where we must define `u` first as a `TrialFunction` and then as a `Function`.

The `solve` function takes the nonlinear equations, derives symbolically the Jacobian matrix, and runs a Newton method to compute the solution.

When we run the code, FEniCS reports on the progress of the Newton iterations. With $2 \cdot (8 \times 8)$ cells, we reach convergence in eight iterations with a tolerance of 10^{-9}, and the error in the numerical solution is about 10^{-16}. These results bring evidence for a correct implementation. Thinking in terms of finite differences on a uniform mesh, P_1 elements mimic standard second-order differences, which compute the derivative of a linear or quadratic function exactly. Here, ∇u is a constant vector, but then multiplied by $(1+u^2)$, which is a second-order polynomial in x and y, which the divergence "difference operator" should compute exactly. We can therefore, even with $\mathcal{P}_1$ elements, expect the manufactured u to be reproduced by the numerical method. With a nonlinearity like $1+u^4$, this will not be the case, and we would need to verify convergence rates instead.

The current example shows how easy it is to solve a nonlinear problem in FEniCS. However, experts on the numerical solution of nonlinear PDEs know very well that automated procedures may fail for nonlinear problems, and that it is often necessary to have much better manual control of the solution process than what we have in the current case. We return to this problem in [23] and show how we can implement taylored solution algorithms for nonlinear equations and also how we can steer the parameters in the automated Newton method used above.

3.3 The equations of linear elasticity

Analysis of structures is one of the major activities of modern engineering, which likely makes the PDE modeling the deformation of elastic bodies the most popular PDE in the world. It takes just one page of code to solve the equations of 2D or 3D elasticity in FEniCS, and the details follow below.

3.3.1 PDE problem

The equations governing small elastic deformations of a body Ω can be written as

$$-\nabla\cdot\sigma = f \text{ in } \Omega, \tag{3.20}$$

$$\sigma = \lambda \operatorname{tr}(\varepsilon) I + 2\mu\varepsilon, \tag{3.21}$$

$$\varepsilon = \frac{1}{2}\left(\nabla u + (\nabla u)^{\top}\right), \tag{3.22}$$

where σ is the stress tensor, f is the body force per unit volume, λ and μ are Lamé's elasticity parameters for the material in Ω, I is the identity tensor, tr is the trace operator on a tensor, ε is the symmetric strain-rate tensor (symmetric gradient), and u is the displacement vector field. We have here assumed isotropic elastic conditions.

We combine (3.21) and (3.22) to obtain

$$\sigma = \lambda(\nabla\cdot u)I + \mu(\nabla u + (\nabla u)^{\top}). \tag{3.23}$$

Note that (3.20)–(3.22) can easily be transformed to a single vector PDE for u, which is the governing PDE for the unknown u (Navier's equation). In the derivation of the variational formulation, however, it is convenient to keep the equations split as above.

3.3.2 Variational formulation

The variational formulation of (3.20)–(3.22) consists of forming the inner product of (3.20) and a *vector* test function $v \in \hat{V}$, where $\hat{V}$ is a vector-valued test function space, and integrating over the domain Ω:

$$-\int_{\Omega}(\nabla\cdot\sigma)\cdot v\,\mathrm{d}x = \int_{\Omega} f\cdot v\,\mathrm{d}x\,.$$

Since $\nabla\cdot\sigma$ contains second-order derivatives of the primary unknown u, we integrate this term by parts:

$$-\int_{\Omega}(\nabla\cdot\sigma)\cdot v\,\mathrm{d}x = \int_{\Omega}\sigma : \nabla v\,\mathrm{d}x - \int_{\partial\Omega}(\sigma\cdot n)\cdot v\,\mathrm{d}s,$$

where the colon operator is the inner product between tensors (summed pairwise product of all elements), and n is the outward unit normal at the boundary. The quantity $\sigma\cdot n$ is known as the *traction* or stress vector at the boundary, and is often prescribed as a boundary condition. We here assume that it is prescribed on a part $\partial\Omega_T$ of the boundary as $\sigma\cdot n = T$. On the remaining

part of the boundary, we assume that the value of the displacement is given as a Dirichlet condition. We thus obtain

$$\int_\Omega \sigma : \nabla v \,\mathrm{d}x = \int_\Omega f \cdot v \,\mathrm{d}x + \int_{\partial\Omega_T} T \cdot v \,\mathrm{d}s\,.$$

Inserting the expression (3.23) for σ gives the variational form with u as unknown. Note that the boundary integral on the remaining part $\partial\Omega \setminus \partial\Omega_T$ vanishes due to the Dirichlet condition.

We can now summarize the variational formulation as: find $u \in V$ such that

$$a(u,v) = L(v) \quad \forall v \in \hat{V}, \tag{3.24}$$

where

$$a(u,v) = \int_\Omega \sigma(u) : \nabla v \,\mathrm{d}x, \tag{3.25}$$

$$\sigma(u) = \lambda(\nabla \cdot u)I + \mu(\nabla u + (\nabla u)^\top), \tag{3.26}$$

$$L(v) = \int_\Omega f \cdot v \,\mathrm{d}x + \int_{\partial\Omega_T} T \cdot v \,\mathrm{d}s\,. \tag{3.27}$$

One can show that the inner product of a symmetric tensor A and an anti-symmetric tensor B vanishes. If we express ∇v as a sum of its symmetric and anti-symmetric parts, only the symmetric part will survive in the product $\sigma : \nabla v$ since σ is a symmetric tensor. Thus replacing ∇u by the symmetric gradient $\epsilon(u)$ gives rise to the slightly different variational form

$$a(u,v) = \int_\Omega \sigma(u) : \varepsilon(v) \,\mathrm{d}x, \tag{3.28}$$

where $\varepsilon(v)$ is the symmetric part of ∇v:

$$\varepsilon(v) = \frac{1}{2}\left(\nabla v + (\nabla v)^\top\right).$$

The formulation (3.28) is what naturally arises from minimization of elastic potential energy and is a more popular formulation than (3.25).

3.3.3 FEniCS implementation

Test problem. As a test example, we will model a clamped beam deformed under its own weight in 3D. This can be modeled by setting the right-hand side body force per unit volume to $f = (0, 0, -\varrho g)$ with ϱ the density of the beam and g the acceleration of gravity. The beam is box-shaped with length

L and has a square cross section of width W. We set $u = u_{\mathrm{D}} = (0,0,0)$ at the clamped end, $x = 0$. The rest of the boundary is traction free; that is, we set $T = 0$.

FEniCS implementation. We first list the code and then comment upon the new constructions compared to the previous examples we have seen.

```
from fenics import *

# Scaled variables
L = 1; W = 0.2
mu = 1
rho = 1
delta = W/L
gamma = 0.4*delta**2
beta = 1.25
lambda_ = beta
g = gamma

# Create mesh and define function space
mesh = BoxMesh(Point(0, 0, 0), Point(L, W, W), 10, 3, 3)
V = VectorFunctionSpace(mesh, 'P', 1)

# Define boundary condition
tol = 1E-14

def clamped_boundary(x, on_boundary):
    return on_boundary and x[0] < tol

bc = DirichletBC(V, Constant((0, 0, 0)), clamped_boundary)

# Define strain and stress

def epsilon(u):
    return 0.5*(nabla_grad(u) + nabla_grad(u).T)
    #return sym(nabla_grad(u))

def sigma(u):
    return lambda_*nabla_div(u)*Identity(d) + 2*mu*epsilon(u)

# Define variational problem
u = TrialFunction(V)
d = u.geometric_dimension()  # space dimension
v = TestFunction(V)
f = Constant((0, 0, -rho*g))
T = Constant((0, 0, 0))
a = inner(sigma(u), epsilon(v))*dx
L = dot(f, v)*dx + dot(T, v)*ds

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

# Plot solution
```

```
plot(u, title='Displacement', mode='displacement')

# Plot stress
s = sigma(u) - (1./3)*tr(sigma(u))*Identity(d)  # deviatoric stress
von_Mises = sqrt(3./2*inner(s, s))
V = FunctionSpace(mesh, 'P', 1)
von_Mises = project(von_Mises, V)
plot(von_Mises, title='Stress intensity')

# Compute magnitude of displacement
u_magnitude = sqrt(dot(u, u))
u_magnitude = project(u_magnitude, V)
plot(u_magnitude, 'Displacement magnitude')
print('min/max u:',
      u_magnitude.vector().array().min(),
      u_magnitude.vector().array().max())
```

This example program can be found in the file `ft06_elasticity.py`.

Vector function spaces. The primary unknown is now a vector field u and not a scalar field, so we need to work with a vector function space:

```
V = VectorFunctionSpace(mesh, 'P', 1)
```

With `u = Function(V)` we get `u` as a vector-valued finite element function with three components for this 3D problem.

Constant vectors. For the boundary condition $u = (0,0,0)$, we must set a vector value to zero, not just a scalar. Such a vector constant is specified as `Constant((0, 0, 0))` in FEniCS. The corresponding 2D code would use `Constant((0, 0))`. Later in the code, we also need `f` as a vector and specify it as `Constant((0, 0, rho*g))`.

`nabla_grad`. The gradient and divergence operators now have a prefix `nabla_`. This is strictly not necessary in the present problem, but recommended in general for vector PDEs arising from continuum mechanics, if you interpret ∇ as a vector in the PDE notation; see the box about `nabla_grad` in Section 3.4.2.

Stress computation. As soon as the displacement `u` is computed, we can compute various stress measures. We will compute the von Mises stress defined as $\sigma_M = \sqrt{\frac{3}{2}s : s}$ where s is the deviatoric stress tensor

$$s = \sigma - \frac{1}{3}\mathrm{tr}\,(\sigma)\,I\,.$$

There is a one-to-one mapping between these formulas and the FEniCS code:

```
s = sigma(u) - (1./3)*tr(sigma(u))*Identity(d)
von_Mises = sqrt(3./2*inner(s, s))
```

The `von_Mises` variable is now an expression that must be projected to a finite element space before we can visualize it:

```
V = FunctionSpace(mesh, 'P', 1)
von_Mises = project(von_Mises, V)
plot(von_Mises, title='Stress intensity')
```

Scaling. It is often advantageous to scale a problem as it reduces the need for setting physical parameters, and one obtains dimensionsless numbers that reflect the competition of parameters and physical effects. We develop the code for the original model with dimensions, and run the scaled problem by tweaking parameters appropriately. Scaling reduces the number of active parameters from 6 to 2 for the present application.

In Navier's equation for u, arising from inserting (3.21) and (3.22) into (3.20),

$$-(\lambda+\mu)\nabla(\nabla\cdot u)-\mu\nabla^2 u=f,$$

we insert coordinates made dimensionless by L, and $\bar{u}=u/U$, which results in the dimensionless governing equation

$$-\beta\bar{\nabla}(\bar{\nabla}\cdot\bar{u})-\bar{\nabla}^2\bar{u}=\bar{f},\quad \bar{f}=(0,0,\gamma),$$

where $\beta=1+\lambda/\mu$ is a dimensionless elasticity parameter and where

$$\gamma=\frac{\varrho g L^2}{\mu U}$$

is a dimensionless variable reflecting the ratio of the load ϱg and the shear stress term $\mu\nabla^2 u\sim\mu U/L^2$ in the PDE.

One option for the scaling is to chose U such that γ is of unit size ($U=\varrho g L^2/\mu$). However, in elasticity, this leads to displacements of the size of the geometry, which makes plots look very strange. We therefore want the characteristic displacement to be a small fraction of the characteristic length of the geometry. This can be achieved by choosing U equal to the maximum deflection of a clamped beam, for which there actually exists a formula: $U=\frac{3}{2}\varrho g L^2\delta^2/E$, where $\delta=L/W$ is a parameter reflecting how slender the beam is, and E is the modulus of elasticity. Thus, the dimensionless parameter δ is very important in the problem (as expected, since $\delta\gg 1$ is what gives beam theory!). Taking E to be of the same order as μ, which is the case for many materials, we realize that $\gamma\sim\delta^{-2}$ is an appropriate choice. Experimenting with the code to find a displacement that "looks right" in plots of the deformed geometry, points to $\gamma=0.4\delta^{-2}$ as our final choice of γ.

The simulation code implements the problem with dimensions and physical parameters λ, μ, ϱ, g, L, and W. However, we can easily reuse this code for a scaled problem: just set $\mu=\varrho=L=1$, W as W/L (δ^{-1}), $g=\gamma$, and $\lambda=\beta$.

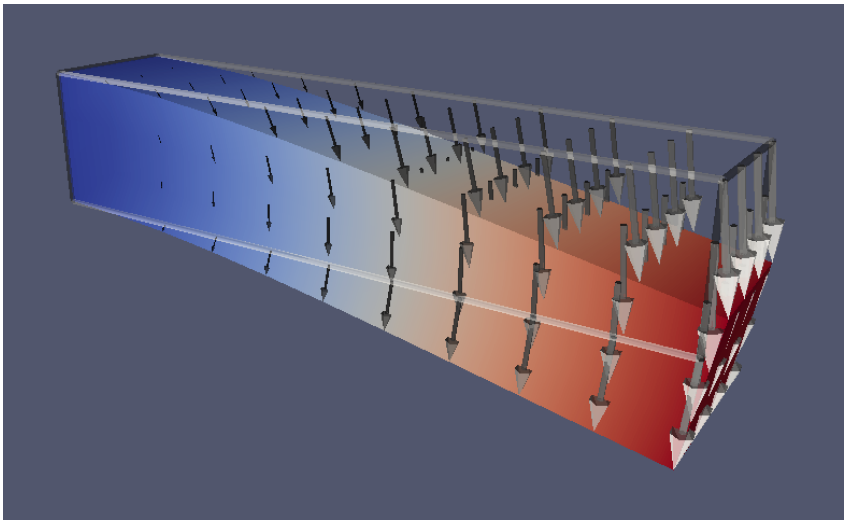


Fig. 3.2 Plot of gravity-induced deflection in a clamped beam for the elasticity problem.

3.4 The Navier–Stokes equations

For the next example, we will solve the incompressible Navier–Stokes equations. This problem combines many of the challenges from our previously studied problems: time-dependence, nonlinearity, and vector-valued variables. We shall touch on a number of FEniCS topics, many of them quite advanced. But you will see that even a relatively complex algorithm such as a second-order splitting method for the incompressible Navier–Stokes equations, can be implemented with relative ease in FEniCS.

3.4.1 PDE problem

The incompressible Navier–Stokes equations form a system of equations for the velocity u and pressure p in an incompressible fluid:

$$\varrho\left(\frac{\partial u}{\partial t} + u\cdot\nabla u\right) = \nabla\cdot\sigma(u,p) + f, \tag{3.29}$$

$$\nabla\cdot u = 0. \tag{3.30}$$

The right-hand side f is a given force per unit volume and just as for the equations of linear elasticity, $\sigma(u,p)$ denotes the stress tensor, which for a Newtonian fluid is given by

$$\sigma(u,p) = 2\mu\epsilon(u) - pI, \tag{3.31}$$

where $\epsilon(u)$ is the strain-rate tensor

$$\epsilon(u) = \frac{1}{2}\left(\nabla u + (\nabla u)^T\right).$$

The parameter μ is the dynamic viscosity. Note that the momentum equation (3.29) is very similar to the elasticity equation (3.20). The difference is in the two additional terms $\varrho(\partial u/\partial t + u\cdot\nabla u)$ and the different expression for the stress tensor. The two extra terms express the acceleration balanced by the force $F = \nabla\cdot\sigma + f$ per unit volume in Newton's second law of motion.

3.4.2 Variational formulation

The Navier–Stokes equations are different from the time-dependent heat equation in that we need to solve a system of equations and this system is of a special type. If we apply the same technique as for the heat equation; that is, replacing the time derivative with a simple difference quotient, we obtain a nonlinear system of equations. This in itself is not a problem for FEniCS as we saw in Section 3.2, but the system has a so-called *saddle point structure* and requires special techniques (special preconditioners and iterative methods) to be solved efficiently.

Instead, we will apply a simpler and often very efficient approach, known as a *splitting method*. The idea is to consider the two equations (3.29) and (3.30) separately. There exist many splitting strategies for the incompressible Navier–Stokes equations. One of the oldest is the method proposed by Chorin [6] and Temam [31], often referred to as *Chorin's method*. We will use a modified version of Chorin's method, the so-called incremental pressure correction scheme (IPCS) due to [13] which gives improved accuracy compared to the original scheme at little extra cost.

The IPCS scheme involves three steps. First, we compute a *tentative velocity* $u^\star$ by advancing the momentum equation (3.29) by a midpoint finite difference scheme in time, but using the pressure p^n from the previous time interval. We will also linearize the nonlinear convective term by using the known velocity u^n from the previous time step: $u^n\cdot\nabla u^n$. The variational problem for this first step is

$$\begin{aligned}\langle\varrho(u^\star - u^n)/\Delta t, v\rangle + \langle\varrho u^n\cdot\nabla u^n, v\rangle + \langle\sigma(u^{n+\frac{1}{2}}, p^n), \epsilon(v)\rangle \\ + \langle p^n n, v\rangle_{\partial\Omega} - \langle\mu\nabla u^{n+\frac{1}{2}}\cdot n, v\rangle_{\partial\Omega} = \langle f^{n+1}, v\rangle.\end{aligned} \tag{3.32}$$

This notation, suitable for problems with many terms in the variational formulations, requires some explanation. First, we use the short-hand notation

$$\langle v, w\rangle = \int_\Omega vw \,\mathrm{d}x, \quad \langle v, w\rangle_{\partial\Omega} = \int_{\partial\Omega} vw \,\mathrm{d}s.$$

This allows us to express the variational problem in a more compact way. Second, we use the notation $u^{n+\frac{1}{2}}$. This notation refers to the value of u at the midpoint of the interval, usually approximated by an arithmetic mean:

$$u^{n+\frac{1}{2}} \approx (u^n + u^{n+1})/2.$$

Third, we notice that the variational problem (3.32) arises from the integration by parts of the term $\langle -\nabla\cdot\sigma, v\rangle$. Just as for the elasticity problem in Section 3.3, we obtain

$$\langle -\nabla\cdot\sigma, v\rangle = \langle \sigma, \epsilon(v)\rangle - \langle T, v\rangle_{\partial\Omega},$$

where $T = \sigma\cdot n$ is the boundary traction. If we solve a problem with a free boundary, we can take $T = 0$ on the boundary. However, if we compute the flow through a channel or a pipe and want to model flow that continues into an "imaginary channel" at the outflow, we need to treat this term with some care. The assumption we then make is that the derivative of the velocity in the direction of the channel is zero at the outflow, corresponding to a flow that is "fully developed" or doesn't change significantly downstream of the outflow. Doing so, the remaining boundary term at the outflow becomes $pn - \mu\nabla u\cdot n$, which is the term appearing in the variational problem (3.32). Note that this argument and the implementation depends on the exact definition of ∇u, as either the matrix with components $\partial u_i/\partial x_j$ or $\partial u_j/\partial x_i$. We here choose the latter, $\partial u_j/\partial x_i$, which means that we must use the FEniCS operator `nabla_grad` for the implementation. If we use the `grad` operator and the definition $\partial u_i/\partial x_j$, we must instead keep the terms $pn - \mu(\nabla u)^\top\cdot n$!

`grad(u)` vs. `nabla_grad(u)`

For scalar functions, ∇u has a clear meaning as the vector

$$\nabla u = \left(\frac{\partial u}{\partial x}, \frac{\partial u}{\partial y}, \frac{\partial u}{\partial z}\right).$$

However, if u is vector-valued, the meaning is less clear. Some sources define ∇u as the matrix with elements $\partial u_j/\partial x_i$, while other sources prefer $\partial u_i/\partial x_j$. In FEniCS, `grad(u)` is defined as the matrix with elements $\partial u_i/\partial x_j$, which is the natural definition of ∇u if we think of this as the *gradient* or *derivative* of u. This way, the matrix ∇u can be applied to a differential $\mathrm{d}x$ to give an increment $\mathrm{d}u = \nabla u\,\mathrm{d}x$. Since the alterna-

tive interpretation of ∇u as the matrix with elements $\partial u_j/\partial x_i$ is very common, in particular in continuum mechanics, FEniCS provides the operator `nabla_grad` for this purpose. For the Navier–Stokes equations, it is important to consider the term $u \cdot \nabla u$ which should be interpreted as the vector w with elements $w_i = \sum_j \left(u_j \frac{\partial}{\partial x_j} \right) u_i = \sum_j u_j \frac{\partial u_i}{\partial x_j}$. This term can be implemented in FEniCS either as `grad(u)*u`, since this is expression becomes $\sum_j \partial u_i/\partial x_j u_j$, or as `dot(u, nabla_grad(u))` since this expression becomes $\sum_i u_i \partial u_j/\partial x_i$. We will use the notation `dot(u, nabla_grad(u))` below since it corresponds more closely to the standard notation $u \cdot \nabla u$.

To be more precise, there are three different notations used for PDEs involving gradient, divergence, and curl operators. One employs $\operatorname{grad} u$, $\operatorname{div} u$, and $\operatorname{curl} u$ operators. Another employs ∇u as a synonym for $\operatorname{grad} u$, $\nabla \cdot u$ means $\operatorname{div} u$, and $\nabla \times u$ is the name for $\operatorname{curl} u$. The third operates with ∇u, $\nabla \cdot u$, and $\nabla \times u$ in which ∇ is a *vector* and, e.g., ∇u is a dyadic expression: $(\nabla u)_{i,j} = \partial u_j/\partial x_i = (\operatorname{grad} u)^\top$. The latter notation, with ∇ as a vector operator, is often handy when deriving equations in continuum mechanics, and if this interpretation of ∇ is the foundation of your PDE, you must use `nabla_grad`, `nabla_div`, and `nabla_curl` in FEniCS code as these operators are compatible with dyadic computations. From the Navier–Stokes equations we can easily see what ∇ means: if the convective term has the form $u \cdot \nabla u$, actually meaning $(u \cdot \nabla)u$, then ∇ is a vector and the implementation becomes `dot(u, nabla_grad(u))` in FEniCS, but if we see $\nabla u \cdot u$ or $(\operatorname{grad} u) \cdot u$, the corresponding FEniCS expression is `dot(grad(u), u)`.

Similarly, the divergence of a tensor field like the stress tensor σ can also be expressed in two different ways, as either `div(sigma)` or `nabla_div(sigma)`. The first case corresponds to the components $\partial \sigma_{ij}/\partial x_j$ and the second to $\partial \sigma_{ij}/\partial x_i$. In general, these expressions will be different but when the stress measure is symmetric, the expressions have the same value.

We now move on to the second step in our splitting scheme for the incompressible Navier–Stokes equations. In the first step, we computed the tentative velocity $u^\star$ based on the pressure from the previous time step. We may now use the computed tentative velocity to compute the new pressure p^n:

$$\langle \nabla p^{n+1}, \nabla q \rangle = \langle \nabla p^n, \nabla q \rangle - \Delta t^{-1} \langle \nabla \cdot u^\star, q \rangle. \tag{3.33}$$

Note here that q is a scalar-valued test function from the pressure space, whereas the test function v in (3.32) is a vector-valued test function from the velocity space.

One way to think about this step is to subtract the Navier–Stokes momentum equation (3.29) expressed in terms of the tentative velocity $u^\star$ and the

pressure p^n from the momentum equation expressed in terms of the velocity u^{n+1} and pressure p^{n+1}. This results in the equation

$$(u^{n+1} - u^\star)/\Delta t + \nabla p^{n+1} - \nabla p^n = 0. \tag{3.34}$$

Taking the divergence and requiring that $\nabla \cdot u^{n+1} = 0$ by the Navier–Stokes continuity equation (3.30), we obtain the equation $-\nabla \cdot u^\star/\Delta t + \nabla^2 p^{n+1} - \nabla^2 p^n = 0$, which is a Poisson problem for the pressure p^{n+1} resulting in the variational problem (3.33).

Finally, we compute the corrected velocity u^{n+1} from the equation (3.34). Multiplying this equation by a test function v, we obtain

$$\langle u^{n+1}, v\rangle = \langle u^\star, v\rangle - \Delta t \langle \nabla(p^{n+1} - p^n), v\rangle. \tag{3.35}$$

In summary, we may thus solve the incompressible Navier–Stokes equations efficiently by solving a sequence of three linear variational problems in each time step.

3.4.3 FEniCS implementation

Test problem 1: Channel flow. As a first test problem, we compute the flow between two infinite plates, so-called channel or Poiseuille flow. As we shall see, this problem has a known analytical solution. Let H be the distance between the plates and L the length of the channel. There are no body forces.

We may scale the problem first to get rid of seemingly independent physical parameters. The physics of this problem is governed by viscous effects only, in the direction perpendicular to the flow, so a time scale should be based on diffusion accross the channel: $t_c = H^2/\nu$. We let U, some characteristic inflow velocity, be the velocity scale and H the spatial scale. The pressure scale is taken as the characteristic shear stress, $\mu U/H$, since this is a primary example of shear flow. Inserting $\bar{x} = x/H$, $\bar{y} = y/H$, $\bar{z} = z/H$, $\bar{u} = u/U$, $\bar{p} = Hp/(\mu U)$, and $\bar{t} = H^2/\nu$ in the equations results in the scaled Navier–Stokes equations (dropping bars after the scaling):

$$\frac{\partial u}{\partial t} + \mathrm{Re}\, u \cdot \nabla u = -\nabla p + \nabla^2 u,$$
$$\nabla \cdot u = 0\,.$$

Here, $\mathrm{Re} = \varrho U H/\mu$ is the Reynolds number. Because of the time and pressure scales, which are different from convection-dominated fluid flow, the Reynolds number is associated with the convective term and not the viscosity term.

The exact solution is derived by assuming $u = (u_x(x,y,z), 0, 0)$, with the x axis pointing along the channel. Since $\nabla \cdot u = 0$, u cannot depend on x.

The physics of channel flow is also two-dimensional so we can omit the z coordinate (more precisely: $\partial/\partial z = 0$). Inserting $u = (u_x, 0, 0)$ in the (scaled) governing equations gives $u_x''(y) = \partial p/\partial x$. Differentiating this equation with respect to x shows that $\partial^2 p/\partial^2 x = 0$ so $\partial p/\partial x$ is a constant, here called $-\beta$. This is the driving force of the flow and can be specified as a known parameter in the problem. Integrating $u_x''(y) = -\beta$ over the width of the channel, $[0,1]$, and requiring $u = (0,0,0)$ at the channel walls, results in $u_x = \frac{1}{2}\beta y(1-y)$. The characteristic inlet velocity U can be taken as the maximum inflow at $y = 1/2$, implying $\beta = 8$. The length of the channel, L/H in the scaled model, has no impact on the result, so for simplicity we just compute on the unit square. Mathematically, the pressure must be prescribed at a point, but since p does not depend on y, we can set p to a known value, e.g. zero, along the outlet boundary $x = 1$. The result is $p(x) = 8(1-x)$ and $u_x = 4y(1-y)$.

The boundary conditions can be set as $p = 8$ at $x = 0$, $p = 0$ at $x = 1$ and $u = (0,0,0)$ on the walls $y = 0, 1$. This defines the pressure drop and should result in unit maximum velocity at the inlet and outlet and a parabolic velocity profile without further specifications. Note that it is only meaningful to solve the Navier–Stokes equations in 2D or 3D geometries, although the underlying mathematical problem collapses to two 1D problems, one for $u_x(y)$ and one for $p(x)$.

The scaled model is not so easy to simulate using a standard Navier–Stokes solver with dimensions. However, one can argue that the convection term is zero, so the Re coefficient in front of this term in the scaled PDEs is not important and can be set to unity. In that case, setting $\varrho = \mu = 1$ in the original Navier–Stokes equations resembles the scaled model.

For a specific engineering problem one wants to simulate a specific fluid and set corresponding parameters. A general solver is therefore most naturally implemented with dimensions and the original physical parameters. However, scaling may greatly simplify numerical simulations. First of all, it shows that all fluids behave in the same way: it does not matter whether we have oil, gas, or water flowing between two plates, and it does not matter how fast the flow is (up to some criticial value of the Reynolds number where the flow becomes unstable and transitions to a complicated turbulent flow of totally different nature). This means that one simulation is enough to cover all types of channel flow! In other applications, scaling shows that it might be necessary to set just the fraction of some parameters (dimensionless numbers) rather than the parameters themselves. This simplifies exploring the input parameter space which is often the purpose of simulation. Frequently, the scaled problem is run by setting some of the input parameters with dimension to fixed values (often unity).

FEniCS implementation. Our previous examples have all started out with the creation of a mesh and then the definition of a `FunctionSpace` on the mesh. For the Navier–Stokes splitting scheme we will need to define two function spaces, one for the velocity and one for the pressure:

```
V = VectorFunctionSpace(mesh, 'P', 2)
Q = FunctionSpace(mesh, 'P', 1)
```

The first space `V` is a vector-valued function space for the velocity and the second space `Q` is a scalar-valued function space for the pressure. We use piecewise quadratic elements for the velocity and piecewise linear elements for the pressure. When creating a `VectorFunctionSpace` in FEniCS, the value-dimension (the length of the vectors) will be set equal to the geometric dimension of the finite element mesh. One can easily create vector-valued function spaces with other dimensions in FEniCS by adding the keyword parameter `dim`:

```
V = VectorFunctionSpace(mesh, 'P', 2, dim=10)
```

Stable finite element spaces for the Navier–Stokes equations

It is well-known that certain finite element spaces are not *stable* for the Navier–Stokes equations, or even for the simpler Stokes equations. The prime example of an unstable pair of finite element spaces is to use first degree continuous piecewise polynomials for both the velocity and the pressure. Using an unstable pair of spaces typically results in a solution with *spurious* (unwanted, non-physical) oscillations in the pressure solution. The simple remedy is to use continuous piecewise quadratic elements for the velocity and continuous piecewise linear elements for the pressure. Together, these elements form the so-called *Taylor-Hood* element. Spurious oscillations may occur also for splitting methods if an unstable element pair is used.

Since we have two different function spaces, we need to create two sets of trial and test functions:

```
u = TrialFunction(V)
v = TestFunction(V)
p = TrialFunction(Q)
q = TestFunction(Q)
```

As we have seen in previous examples, boundaries may be defined in FEniCS by defining Python functions that return `True` or `False` depending on whether a point should be considered part of the boundary, for example

```
def boundary(x, on_boundary):
    return near(x[0], 0)
```

This function defines the boundary to be all points with x-coordinate equal to (near) zero. The `near` function comes from FEniCS and performs a test with tolerance: `abs(x[0] - 0) < 3E-16` so we do not run into rounding troubles. Alternatively, we may give the boundary definition as a

string of C++ code, much like we have previously defined expressions such as `u_D = Expression('1 + x[0]*x[0] + 2*x[1]*x[1]', degree=2)`. The above definition of the boundary in terms of a Python function may thus be replaced by a simple C++ string:

```
boundary = 'near(x[0], 0)'
```

This has the advantage of moving the computation of which nodes belong to the boundary from Python to C++, which improves the efficiency of the program.

For the current example, we will set three different boundary conditions. First, we will set $u = 0$ at the walls of the channel; that is, at $y = 0$ and $y = 1$. Second, we will set $p = 8$ at the inflow ($x = 0$) and, finally, $p = 0$ at the outflow ($x = 1$). This will result in a pressure gradient that will accelerate the flow from the initial state with zero velocity. These boundary conditions may be defined as follows:

```
# Define boundaries
inflow   = 'near(x[0], 0)'
outflow  = 'near(x[0], 1)'
walls    = 'near(x[1], 0) || near(x[1], 1)'

# Define boundary conditions
bcu_noslip  = DirichletBC(V, Constant((0, 0)), walls)
bcp_inflow  = DirichletBC(Q, Constant(8), inflow)
bcp_outflow = DirichletBC(Q, Constant(0), outflow)
bcu = [bcu_noslip]
bcp = [bcp_inflow, bcp_outflow]
```

At the end, we collect the boundary conditions for the velocity and pressure in Python lists so we can easily access them in the following computation.

We now move on to the definition of the variational forms. There are three variational problems to be defined, one for each step in the IPCS scheme. Let us look at the definition of the first variational problem. We start with some constants:

```
U   = 0.5*(u_n + u)
n   = FacetNormal(mesh)
f   = Constant((0, 0))
k   = Constant(dt)
mu  = Constant(mu)
rho = Constant(rho)
```

The next step is to set up the variational form for the first step (3.32) in the solution process. Since the variational problem contains a mix of known and unknown quantities we will use the following naming convention: `u` is the unknown (mathematically u^{n+1}) as a trial function in the variational form, `u_` is the most recently computed approximation (u^{n+1} available as a `Function` object), `u_n` is u^n, and the same convention goes for `p`, `p_` (p^{n+1}), and `p_n` (p^n).

```
# Define strain-rate tensor
def epsilon(u):
    return sym(nabla_grad(u))

# Define stress tensor
def sigma(u, p):
    return 2*mu*epsilon(u) - p*Identity(len(u))

# Define variational problem for step 1
F1 = rho*dot((u - u_n) / k, v)*dx + \
     rho*dot(dot(u_n, nabla_grad(u_n)), v)*dx \
   + inner(sigma(U, p_n), epsilon(v))*dx \
   + dot(p_n*n, v)*ds - dot(mu*nabla_grad(U)*n, v)*ds \
   - dot(f, v)*dx
a1 = lhs(F1)
L1 = rhs(F1)
```

Note that we take advantage of the Python programming language to define our own operators `sigma` and `epsilon`. Using Python this way makes it easy to extend the mathematical language of FEniCS with special operators and constitutive laws.

Also note that FEniCS can sort out the bilinear form $a(u, v)$ and linear form $L(v)$ forms by the `lhs` and `rhs` functions. This is particularly convenient in longer and more complicated variational forms.

The splitting scheme requires the solution of a sequence of three variational problems in each time step. We have previously used the built-in FEniCS function `solve` to solve variational problems. Under the hood, when a user calls `solve(a == L, u, bc)`, FEniCS will perform the following steps:

```
A = assemble(A)
b = assemble(L)
bc.apply(A, b)
solve(A, u.vector(), b)
```

In the last step, FEniCS uses the overloaded `solve` function to solve the linear system `AU = b` where `U` is the vector of degrees of freedom for the function $u(x) = \sum_{j=1} U_j \phi_j(x)$.

In our implementation of the splitting scheme, we will make use of these low-level commands to first assemble and then call solve. This has the advantage that we may control when we assemble and when we solve the linear system. In particular, since the matrices for the three variational problems are all time-independent, it makes sense to assemble them once and for all outside of the time-stepping loop:

```
A1 = assemble(a1)
A2 = assemble(a2)
A3 = assemble(a3)
```

Within the time-stepping loop, we may then assemble only the right-hand side vectors, apply boundary conditions, and call the solve function as here for the first of the three steps:

```
# Time-stepping
t = 0
for n in range(num_steps):

    # Update current time
    t += dt

    # Step 1: Tentative velocity step
    b1 = assemble(L1)
    [bc.apply(b1) for bc in bcu]
    solve(A1, u_.vector(), b1)
```

Notice the Python *list comprehension* `[bc.apply(b1) for bc in bcu]` which iterates over all `bc` in the list `bcu`. This is a convenient and compact way to construct a loop that applies all boundary conditions in a single line. Also, the code works if we add more Dirichlet boundary conditions in the future. Note that the boundary conditions only need to be applied to the right-hand side vectors as they have already been applied to the matrices (not shown).

Finally, let us look at an important detail in how we use parameters such as the time step `dt` in the definition of our variational problems. Since we might want to change these later, for example if we want to experiment with smaller or larger time steps, we wrap these using a FEniCS `Constant`:

```
k = Constant(dt)
```

The assembly of matrices and vectors in FEniCS is based on code generation. This means that whenever we change a variational problem, FEniCS will have to generate new code, which may take a little time. New code will also be generated and compiled when a float value for the time step is changed. By wrapping this parameter using `Constant`, FEniCS will treat the parameter as a generic constant and not as a specific numerical value, which prevents repeated code generation. In the case of the time step, we choose a new name `k` instead of `dt` for the `Constant` since we also want to use the variable `dt` as a Python float as part of the time-stepping.

The complete code for simulating 2D channel flow with FEniCS can be found in the file `ft07_navier_stokes_channel.py`.

Verification. We compute the error at the nodes as we have done before to verify that our implementation is correct. Our Navier–Stokes solver computes the solution to the time-dependent incompressible Navier–Stokes equations, starting from the initial condition $u = (0, 0)$. We have not specified the initial condition explicitly in our solver which means that FEniCS will initialize all variables, in particular the previous and current velocities `u_n` and `u_`, to zero. Since the exact solution is quadratic, we expect the solution to be exact to within machine precision at the nodes at infinite time. For our implementation, the error quickly approaches zero and is approximately 10^{-6} at time $T = 10$.

Test problem 2: Flow past a cylinder. We now turn our attention to a more challenging problem: flow past a circular cylinder. The geometry and pa-

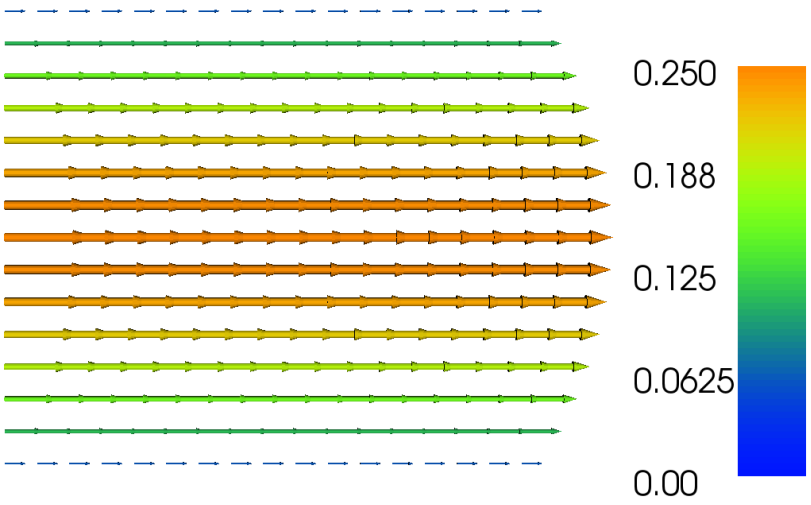


Fig. 3.3 Plot of the velocity profile at the final time for the Navier–Stokes channel flow example.

rameters are taken from problem DFG 2D-2 in the FEATFLOW/1995-DFG benchmark suite[1] and is illustrated in Figure 3.4. The kinematic viscosity is given by $\nu = 0.001 = \mu/\varrho$ and the inflow velocity profile is specified as

$$u(x,y,t) = \left(1.5 \cdot \frac{4y(0.41-y)}{0.41^2}, 0\right),$$

which has a maximum magnitude of 1.5 at $y = 0.41/2$. We do not use any scaling for this problem since all exact parameters are known.

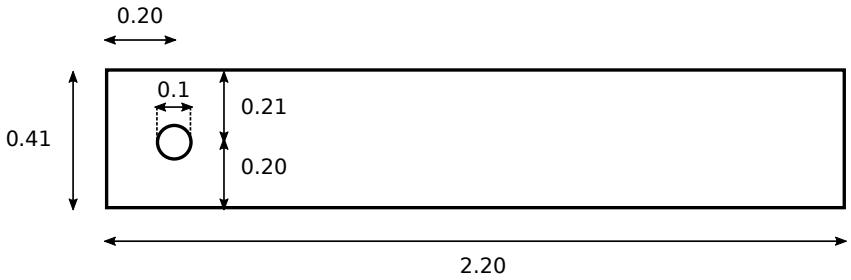


Fig. 3.4 Geometry for the flow past a cylinder test problem. Notice the slightly perturbed and unsymmetric geometry.

[1] `http://www.featflow.de/en/benchmarks/cfdbenchmarking/flow/dfg_benchmark2_re100.html`

FEniCS implementation. So far all our domains have been simple shapes such as a unit square or a rectangular box. A number of such simple meshes may be created using the built-in mesh classes in FEniCS (`UnitIntervalMesh`, `UnitSquareMesh`, `UnitCubeMesh`, `IntervalMesh`, `RectangleMesh`, `BoxMesh`). FEniCS supports the creation of more complex meshes via a technique called *constructive solid geometry* (CSG), which lets us define geometries in terms of simple shapes (primitives) and set operations: union, intersection, and set difference. The set operations are encoded in FEniCS using the operators `+` (union), `*` (intersection), and `-` (set difference). To access the CSG functionality in FEniCS, one must import the FEniCS module `mshr` which provides the extended meshing functionality of FEniCS.

The geometry for the cylinder flow test problem can be defined easily by first defining the rectangular channel and then subtracting the circle:

```
channel = Rectangle(Point(0, 0), Point(2.2, 0.41))
cylinder = Circle(Point(0.2, 0.2), 0.05)
domain = channel - cylinder
```

We may then create the mesh by calling the function `generate_mesh`:

```
mesh = generate_mesh(domain, 64)
```

Here the argument `64` indicates that we want to resolve the geometry with 64 cells across its diameter (the channel length).

To solve the cylinder test problem, we only need to make a few minor changes to the code we wrote for the channel flow test case. Besides defining the new mesh, the only change we need to make is to modify the boundary conditions and the time step size. The boundaries are specified as follows:

```
inflow   = 'near(x[0], 0)'
outflow  = 'near(x[0], 2.2)'
walls    = 'near(x[1], 0) || near(x[1], 0.41)'
cylinder = 'on_boundary && x[0]>0.1 && x[0]<0.3 && x[1]>0.1 && x[1]<0.3'
```

The last line may seem cryptic before you catch the idea: we want to pick out all boundary points (`on_boundary`) that also lie within the 2D domain $[0.1, 0.3] \times [0.1, 0.3]$, see Figure 3.4. The only possible points are then the points on the circular boundary!

In addition to these essential changes, we will make a number of small changes to improve our solver. First, since we need to choose a relatively small time step to compute the solution (a time step that is too large will make the solution blow up) we add a progress bar so that we can follow the progress of our computation. This can be done as follows:

```
progress = Progress('Time-stepping')
set_log_level(PROGRESS)

# Time-stepping
t = 0.0
for n in range(num_steps):
```

```
    # Update current time
    t += dt

    # Place computation here

    # Update progress bar
    progress.update(t / T)
```

Log levels and printing in FEniCS

Notice the call to `set_log_level(PROGRESS)` which is essential to make FEniCS actually display the progress bar. FEniCS is actually quite informative about what is going on during a computation but the amount of information printed to screen depends on the current log level. Only messages with a priority higher than or equal to the current log level will be displayed. The predefined log levels in FEniCS are `DBG`, `TRACE`, `PROGRESS`, `INFO`, `WARNING`, `ERROR`, and `CRITICAL`. By default, the log level is set to `INFO` which means that messages at level `DBG`, `TRACE`, and `PROGRESS` will not be printed. Users may print messages using the FEniCS functions `info`, `warning`, and `error` which will print messages at the obvious log level (and in the case of `error` also throw an exception and exit). One may also use the call `log(level, message)` to print a message at a specific log level.

Since the system(s) of linear equations are significantly larger than for the simple channel flow test problem, we choose to use an iterative method instead of the default direct (sparse) solver used by FEniCS when calling `solve`. Efficient solution of linear systems arising from the discretization of PDEs requires the choice of both a good iterative (Krylov subspace) method and a good preconditioner. For this problem, we will simply use the biconjugate gradient stabilized method (BiCGSTAB) and the conjugate gradient method. This can be done by adding the keywords `bicgstab` or `cg` in the call to `solve`. We also specify suitable preconditioners to speed up the computations:

```
solve(A1, u1.vector(), b1, 'bicgstab', 'hypre_amg')
solve(A2, p1.vector(), b2, 'bicgstab', 'hypre_amg')
solve(A3, u1.vector(), b3, 'cg', 'sor')
```

Finally, to be able to postprocess the computed solution in ParaView, we store the solution to a file in each time step. We have previously created files with the suffix `.pvd` for this purpose. In the example program `ft04_heat_gaussian.py`, we first created a file named `heat_gaussian/solution.pvd` and then saved the solution in each time step using

```
vtkfile << (u, t)
```

For the present example, we will instead choose to save the solution to XDMF format. This file format works similarly to the `.pvd` files we have seen earlier but has several advantages. First, the storage is much more efficient, both in terms of speed and file sizes. Second, `.xdmf` files work in parallell, both for writing and reading (postprocessing). Much like `.pvd` files, the actual data will not be stored in the `.xdmf` file itself, but will instead be stored in a (single) separate data file with the suffix `.hdf5` which is an advanced file format designed for high-performance computing. We create the XDMF files as follows:

```
xdmffile_u = XDMFFile('navier_stokes_cylinder/velocity.xdmf')
xdmffile_p = XDMFFile('navier_stokes_cylinder/pressure.xdmf')
```

In each time step, we may then store the velocity and pressure by

```
xdmffile_u.write(u, t)
xdmffile_p.write(p, t)
```

We also store the solution using a FEniCS `TimeSeries`. This allows us to store the solution not for visualization, but for later reuse in a computation as we will see in the next section. Using a `TimeSeries` it is easy and efficient to read in solutions from certain points in time during a simulation. The `TimeSeries` class also uses the HDF5 file format for efficient storage and access to data.

Figures 3.5 and 3.6 show the velocity and pressure at final time visualized in ParaView. For the visualization of the velocity, we have used the **Glyph** filter to visualize the vector velocity field. For the visualization of the pressure, we have used the **Warp By Scalar** filter.

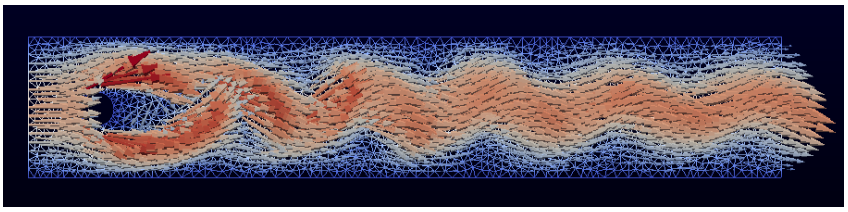


Fig. 3.5 Plot of the velocity for the cylinder test problem at final time.

The complete code for the cylinder test problem looks as follows:

```
from fenics import *
from mshr import *
import numpy as np

T = 5.0            # final time
num_steps = 5000   # number of time steps
dt = T / num_steps # time step size
mu = 0.001         # dynamic viscosity
```

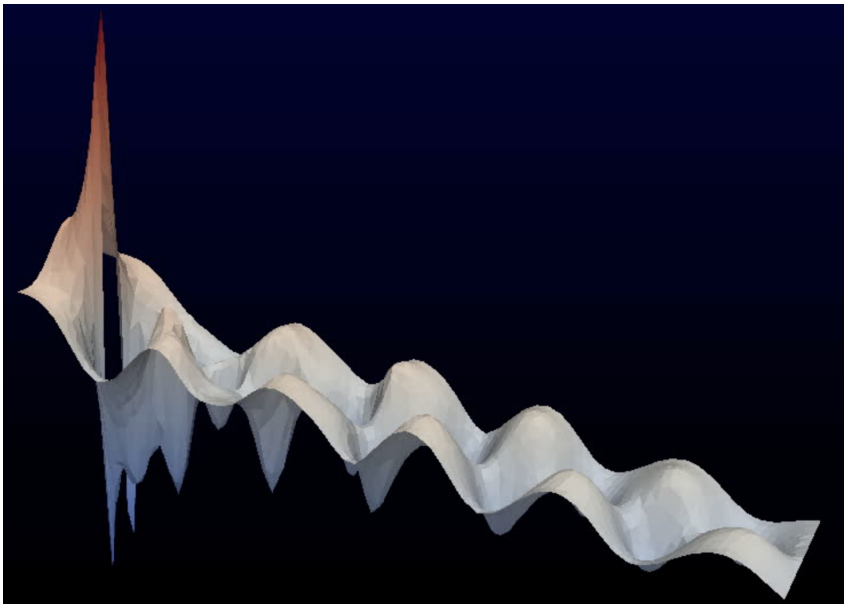


Fig. 3.6 Plot of the pressure for the cylinder test problem at final time.

```
rho = 1            # density

# Create mesh
channel = Rectangle(Point(0, 0), Point(2.2, 0.41))
cylinder = Circle(Point(0.2, 0.2), 0.05)
domain = channel - cylinder
mesh = generate_mesh(domain, 64)

# Define function spaces
V = VectorFunctionSpace(mesh, 'P', 2)
Q = FunctionSpace(mesh, 'P', 1)

# Define boundaries
inflow   = 'near(x[0], 0)'
outflow  = 'near(x[0], 2.2)'
walls    = 'near(x[1], 0) || near(x[1], 0.41)'
cylinder = 'on_boundary && x[0]>0.1 && x[0]<0.3 && x[1]>0.1 && x[1]<0.3'

# Define inflow profile
inflow_profile = ('4.0*1.5*x[1]*(0.41 - x[1]) / pow(0.41, 2)', '0')

# Define boundary conditions
bcu_inflow = DirichletBC(V, Expression(inflow_profile, degree=2), inflow)
bcu_walls = DirichletBC(V, Constant((0, 0)), walls)
bcu_cylinder = DirichletBC(V, Constant((0, 0)), cylinder)
bcp_outflow = DirichletBC(Q, Constant(0), outflow)
bcu = [bcu_inflow, bcu_walls, bcu_cylinder]
```

```
bcp = [bcp_outflow]

# Define trial and test functions
u = TrialFunction(V)
v = TestFunction(V)
p = TrialFunction(Q)
q = TestFunction(Q)

# Define functions for solutions at previous and current time steps
u_n = Function(V)
u_  = Function(V)
p_n = Function(Q)
p_  = Function(Q)

# Define expressions used in variational forms
U  = 0.5*(u_n + u)
n  = FacetNormal(mesh)
f  = Constant((0, 0))
k  = Constant(dt)
mu = Constant(mu)
rho = Constant(rho)

# Define symmetric gradient
def epsilon(u):
    return sym(nabla_grad(u))

# Define stress tensor
def sigma(u, p):
    return 2*mu*epsilon(u) - p*Identity(len(u))

# Define variational problem for step 1
F1 = rho*dot((u - u_n) / k, v)*dx \
   + rho*dot(dot(u_n, nabla_grad(u_n)), v)*dx \
   + inner(sigma(U, p_n), epsilon(v))*dx \
   + dot(p_n*n, v)*ds - dot(mu*nabla_grad(U)*n, v)*ds \
   - dot(f, v)*dx
a1 = lhs(F1)
L1 = rhs(F1)

# Define variational problem for step 2
a2 = dot(nabla_grad(p), nabla_grad(q))*dx
L2 = dot(nabla_grad(p_n), nabla_grad(q))*dx - (1/k)*div(u_)*q*dx

# Define variational problem for step 3
a3 = dot(u, v)*dx
L3 = dot(u_, v)*dx - k*dot(nabla_grad(p_ - p_n), v)*dx

# Assemble matrices
A1 = assemble(a1)
A2 = assemble(a2)
A3 = assemble(a3)

# Apply boundary conditions to matrices
[bc.apply(A1) for bc in bcu]
```

```
[bc.apply(A2) for bc in bcp]

# Create XDMF files for visualization output
xdmffile_u = XDMFFile('navier_stokes_cylinder/velocity.xdmf')
xdmffile_p = XDMFFile('navier_stokes_cylinder/pressure.xdmf')

# Create time series (for use in reaction_system.py)
timeseries_u = TimeSeries('navier_stokes_cylinder/velocity_series')
timeseries_p = TimeSeries('navier_stokes_cylinder/pressure_series')

# Save mesh to file (for use in reaction_system.py)
File('navier_stokes_cylinder/cylinder.xml.gz') << mesh

# Create progress bar
progress = Progress('Time-stepping')
set_log_level(PROGRESS)

# Time-stepping
t = 0
for n in range(num_steps):

    # Update current time
    t += dt

    # Step 1: Tentative velocity step
    b1 = assemble(L1)
    [bc.apply(b1) for bc in bcu]
    solve(A1, u_.vector(), b1, 'bicgstab', 'hypre_amg')

    # Step 2: Pressure correction step
    b2 = assemble(L2)
    [bc.apply(b2) for bc in bcp]
    solve(A2, p_.vector(), b2, 'bicgstab', 'hypre_amg')

    # Step 3: Velocity correction step
    b3 = assemble(L3)
    solve(A3, u_.vector(), b3, 'cg', 'sor')

    # Plot solution
    plot(u_, title='Velocity')
    plot(p_, title='Pressure')

    # Save solution to file (XDMF/HDF5)
    xdmffile_u.write(u_, t)
    xdmffile_p.write(p_, t)

    # Save nodal values to file
    timeseries_u.store(u_.vector(), t)
    timeseries_p.store(p_.vector(), t)

    # Update previous solution
    u_n.assign(u_)
    p_n.assign(p_)
```

```
    # Update progress bar
    progress.update(t / T)
    print('u max:', u_.vector().array().max())

# Hold plot
interactive()
```

This program can be found in the file `ft08_navier_stokes_cylinder.py`. The reader should be advised that this example program is considerably more demanding than our previous examples in terms of CPU time and memory, but it should be possible to run the program on a reasonably modern laptop.

3.5 A system of advection–diffusion–reaction equations

The problems we have encountered so far—with the notable exception of the Navier–Stokes equations—all share a common feature: they all involve models expressed by a *single* scalar or vector PDE. In many situations the model is instead expressed as a system of PDEs, describing different quantities possibly governed by (very) different physics. As we saw for the Navier–Stokes equations, one way to solve a system of PDEs in FEniCS is to use a splitting method where we solve one equation at a time and feed the solution from one equation into the next. However, one of the strengths with FEniCS is the ease by which one can instead define variational problems that couple several PDEs into one compound system. In this section, we will look at how to use FEniCS to write solvers for such systems of coupled PDEs. The goal is to demonstrate how easy it is to implement fully implicit, also known as monolithic, solvers in FEniCS.

3.5.1 PDE problem

Our model problem is the following system of advection–diffusion–reaction equations:

$$\frac{\partial u_1}{\partial t} + w \cdot \nabla u_1 - \nabla \cdot (\epsilon \nabla u_1) = f_1 - K u_1 u_2, \tag{3.36}$$

$$\frac{\partial u_2}{\partial t} + w \cdot \nabla u_2 - \nabla \cdot (\epsilon \nabla u_2) = f_2 - K u_1 u_2, \tag{3.37}$$

$$\frac{\partial u_3}{\partial t} + w \cdot \nabla u_3 - \nabla \cdot (\epsilon \nabla u_3) = f_3 + K u_1 u_2 - K u_3. \tag{3.38}$$

This system models the chemical reaction between two species A and B in some domain Ω:

$$A + B \to C.$$

We assume that the reaction is *first-order*, meaning that the reaction rate is proportional to the concentrations $[A]$ and $[B]$ of the two species A and B:

$$\frac{\mathrm{d}}{\mathrm{d}t}[C] = K[A][B].$$

We also assume that the formed species C spontaneously decays with a rate proportional to the concentration $[C]$. In the PDE system (3.36)–(3.38), we use the variables u_1, u_2, and u_3 to denote the concentrations of the three species:

$$u_1 = [A], \quad u_2 = [B], \quad u_3 = [C].$$

We see that the chemical reactions are accounted for in the right-hand sides of the PDE system (3.36)–(3.38).

The chemical reactions take part at each point in the domain Ω. In addition, we assume that the species A, B, and C diffuse throughout the domain with diffusivity ϵ (the terms $-\nabla\cdot(\epsilon\nabla u_i)$) and are advected with velocity w (the terms $w\cdot\nabla u_i$).

To make things visually and physically interesting, we shall let the chemical reaction take place in the velocity field computed from the solution of the incompressible Navier–Stokes equations around a cylinder from the previous section. In summary, we will thus be solving the following coupled system of nonlinear PDEs:

$$\varrho\left(\frac{\partial w}{\partial t} + w\cdot\nabla w\right) = \nabla\cdot\sigma(w,p) + f, \tag{3.39}$$

$$\nabla\cdot w = 0, \tag{3.40}$$

$$\frac{\partial u_1}{\partial t} + w\cdot\nabla u_1 - \nabla\cdot(\epsilon\nabla u_1) = f_1 - Ku_1u_2, \tag{3.41}$$

$$\frac{\partial u_2}{\partial t} + w\cdot\nabla u_2 - \nabla\cdot(\epsilon\nabla u_2) = f_2 - Ku_1u_2, \tag{3.42}$$

$$\frac{\partial u_3}{\partial t} + w\cdot\nabla u_3 - \nabla\cdot(\epsilon\nabla u_3) = f_3 + Ku_1u_2 - Ku_3. \tag{3.43}$$

We assume that $u_1 = u_2 = u_3 = 0$ at $t = 0$ and inject the species A and B into the system by specifying nonzero source terms f_1 and f_2 close to the corners at the inflow, and take $f_3 = 0$. The result will be that A and B are convected by advection and diffusion throughout the channel, and when they mix the species C will be formed.

Since the system is one-way coupled from the Navier–Stokes subsystem to the advection–diffusion–reaction subsystem, we do not need to recompute the solution to the Navier–Stokes equations, but can just read back the previously computed velocity field w and feed it into our equations. But we *do* need to learn how to read and write solutions for time-dependent PDE problems.

3.5.2 Variational formulation

We obtain the variational formulation of our system by multiplying each equation by a test function, integrating the second-order terms $-\nabla\cdot(\epsilon\nabla u_i)$ by parts, and summing up the equations. When working with FEniCS it is convenient to think of the PDE system as a vector of equations. The test functions are collected in a vector too, and the variational formulation is the inner product of the vector PDE and the vector test function.

We also need introduce some discretization in time. We will use the backward Euler method as before when we solved the heat equation and approximate the time derivatives by $(u_i^{n+1}-u_i^n)/\Delta t$. Let v_1, v_2, and v_3 be the test functions, or the components of the test vector function. The inner product results in

$$
\begin{aligned}
&\int_\Omega (\Delta t^{-1}(u_1^{n+1}-u_1^n)v_1 + w\cdot\nabla u_1^{n+1}\, v_1 + \epsilon\nabla u_1^{n+1}\cdot\nabla v_1)\,\mathrm{d}x \\
&+\int_\Omega (\Delta t^{-1}(u_2^{n+1}-u_2^n)v_2 + w\cdot\nabla u_2^{n+1}\, v_2 + \epsilon\nabla u_2^{n+1}\cdot\nabla v_2)\,\mathrm{d}x \\
&+\int_\Omega (\Delta t^{-1}(u_3^{n+1}-u_3^n)v_3 + w\cdot\nabla u_3^{n+1}\, v_3 + \epsilon\nabla u_3^{n+1}\cdot\nabla v_3)\,\mathrm{d}x \\
&-\int_\Omega (f_1v_1 + f_2v_2 + f_3v_3)\,\mathrm{d}x \\
&-\int_\Omega (-Ku_1^{n+1}u_2^{n+1}v_1 - Ku_1^{n+1}u_2^{n+1}v_2 + Ku_1^{n+1}u_2^{n+1}v_3 - Ku_3^{n+1}v_3)\,\mathrm{d}x = 0.
\end{aligned}
\tag{3.44}
$$

For this problem it is natural to assume homogeneous Neumann boundary conditions on the entire boundary for u_1, u_2, and u_3; that is, $\partial u_i/\partial n = 0$ for $i = 1,2,3$. This means that the boundary terms vanish when we integrate by parts.

3.5.3 FEniCS implementation

The first step is to read the mesh from file. Luckily, we made sure to save the mesh to file in the Navier–Stokes example and can now easily read it back from file:

```
mesh = Mesh('navier_stokes_cylinder/cylinder.xml.gz')
```

The mesh is stored in the native FEniCS XML format (with additional gzipping to decrease the file size).

Next, we need to define the finite element function space. For this problem, we need to define several spaces. The first space we create is the space for the velocity field w from the Navier–Stokes simulation. We call this space W and define the space by

```
W = VectorFunctionSpace(mesh, 'P', 2)
```

It is important that this space is exactly the same as the space we used for the velocity field in the Navier–Stokes solver. To read the values for the velocity field, we use a `TimeSeries`:

```
timeseries_w = TimeSeries('navier_stokes_cylinder/velocity_series')
```

This will initialize the object `timeseries_w` which we will call later in the time-stepping loop to retrieve values from the file `velocity_series.h5` (in binary HDF5 format).

For the three concentrations u_1, u_2, and u_3, we want to create a *mixed space* with functions that represent the full system (u_1, u_2, u_3) as a single entity. To do this, we need to define a `MixedElement` as the product space of three simple finite elements and then used the mixed element to define the function space:

```
P1 = FiniteElement('P', triangle, 1)
element = MixedElement([P1, P1, P1])
V = FunctionSpace(mesh, element)
```

Mixed elements as products of elements

FEniCS also allows finite elements to be defined as products of simple elements (or mixed elements). For example, the well-known Taylor–Hood element, with quadratic velocity components and linear pressure functions, may be defined as follows:

```
P2 = VectorElement('P', triangle, 2)
P1 = FiniteElement('P', triangle, 1)
TH = P2 * P1
```

This syntax works great for two elements, but for three or more elements we meet a subtle issue in how the Python interpreter handles the `*` operator. For the reaction system, we create the mixed element by `element = MixedElement([P1, P1, P1])` and one would be tempted to write

```
element = P1 * P1 * P1
```

However, this is equivalent to writing `element = (P1 * P1) * P1` so the result will be a mixed element consisting of two subsystems, the first of which in turn consists of two scalar subsystems.

Finally, we remark that for the simple case of a mixed system consisting of three scalar elements as for the reaction system, the definition is in fact equivalent to using a standard vector-valued element:

```
element = VectorElement('P', triangle, 1, dim=3)
V = FunctionSpace(mesh, element)
```

Once the space has been created, we need to define our test functions and finite element functions. Test functions for a mixed function space can be created by replacing `TestFunction` by `TestFunctions`:

```
v_1, v_2, v_3 = TestFunctions(V)
```

Since the problem is nonlinear, we need to work with functions rather than trial functions for the unknowns. This can be done by using the corresponding `Functions` construction in FEniCS. However, as we will need to access the `Function` for the entire system itself, we first need to create that function and then access its components:

```
u = Function(V)
u_1, u_2, u_3 = split(u)
```

These functions will be used to represent the unknowns u_1, u_2, and u_3 at the new time level $n+1$. The corresponding values at the previous time level n are denoted by `u_n1`, `u_n2`, and `u_n3` in our program.

When now all functions and test functions have been defined, we can express the nonlinear variational problem (3.44):

```
F = ((u_1 - u_n1) / k)*v_1*dx + dot(w, grad(u_1))*v_1*dx \
  + eps*dot(grad(u_1), grad(v_1))*dx + K*u_1*u_2*v_1*dx  \
  + ((u_2 - u_n2) / k)*v_2*dx + dot(w, grad(u_2))*v_2*dx \
  + eps*dot(grad(u_2), grad(v_2))*dx + K*u_1*u_2*v_2*dx  \
  + ((u_3 - u_n3) / k)*v_3*dx + dot(w, grad(u_3))*v_3*dx \
  + eps*dot(grad(u_3), grad(v_3))*dx - K*u_1*u_2*v_3*dx + K*u_3*v_3*dx \
  - f_1*v_1*dx - f_2*v_2*dx - f_3*v_3*dx
```

The time-stepping simply consists of solving this variational problem in each time step by a call to the `solve` function:

```
t = 0
for n in range(num_steps):
    t += dt
    timeseries_w.retrieve(w.vector(), t)
    solve(F == 0, u)
    u_n.assign(u)
```

In each time step, we first read the current value for the velocity field from the time series we have previously stored. We then solve the nonlinear system, and assign the computed values to the left-hand side values for the next time interval. When retrieving values from a `TimeSeries`, the values will by default be interpolated (linearly) to the given time `t` if the time does not exactly match a sample in the series.

The solution at the final time is shown in Figure 3.7. We clearly see the advection of the species A and B and the formation of C along the center of the channel where A and B meet.

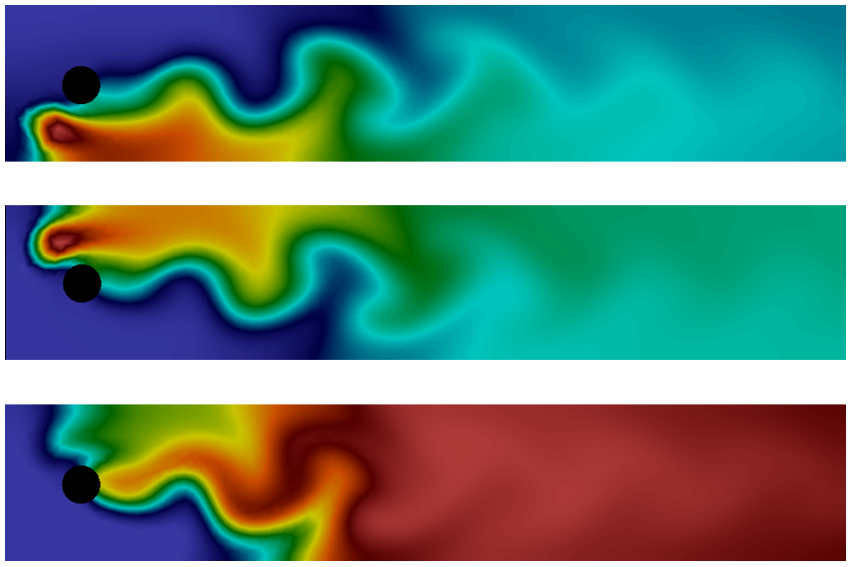


Fig. 3.7 Plot of the concentrations of the three species A, B, and C (from top to bottom) at final time.

The complete code is presented below.

```
from fenics import *

T = 5.0            # final time
num_steps = 500    # number of time steps
dt = T / num_steps # time step size
eps = 0.01         # diffusion coefficient
K = 10.0           # reaction rate
```

```
# Read mesh from file
mesh = Mesh('navier_stokes_cylinder/cylinder.xml.gz')

# Define function space for velocity
W = VectorFunctionSpace(mesh, 'P', 2)

# Define function space for system of concentrations
P1 = FiniteElement('P', triangle, 1)
element = MixedElement([P1, P1, P1])
V = FunctionSpace(mesh, element)

# Define test functions
v_1, v_2, v_3 = TestFunctions(V)

# Define functions for velocity and concentrations
w = Function(W)
u = Function(V)
u_n = Function(V)

# Split system functions to access components
u_1, u_2, u_3 = split(u)
u_n1, u_n2, u_n3 = split(u_n)

# Define source terms
f_1 = Expression('pow(x[0]-0.1,2)+pow(x[1]-0.1,2)<0.05*0.05 ? 0.1 : 0',
                 degree=1)
f_2 = Expression('pow(x[0]-0.1,2)+pow(x[1]-0.3,2)<0.05*0.05 ? 0.1 : 0',
                 degree=1)
f_3 = Constant(0)

# Define expressions used in variational forms
k = Constant(dt)
K = Constant(K)
eps = Constant(eps)

# Define variational problem
F = ((u_1 - u_n1) / k)*v_1*dx + dot(w, grad(u_1))*v_1*dx \
  + eps*dot(grad(u_1), grad(v_1))*dx + K*u_1*u_2*v_1*dx  \
  + ((u_2 - u_n2) / k)*v_2*dx + dot(w, grad(u_2))*v_2*dx \
  + eps*dot(grad(u_2), grad(v_2))*dx + K*u_1*u_2*v_2*dx  \
  + ((u_3 - u_n3) / k)*v_3*dx + dot(w, grad(u_3))*v_3*dx \
  + eps*dot(grad(u_3), grad(v_3))*dx - K*u_1*u_2*v_3*dx + K*u_3*v_3*dx \
  - f_1*v_1*dx - f_2*v_2*dx - f_3*v_3*dx

# Create time series for reading velocity data
timeseries_w = TimeSeries('navier_stokes_cylinder/velocity_series')

# Create VTK files for visualization output
vtkfile_u_1 = File('reaction_system/u_1.pvd')
vtkfile_u_2 = File('reaction_system/u_2.pvd')
vtkfile_u_3 = File('reaction_system/u_3.pvd')

# Create progress bar
```

```
progress = Progress('Time-stepping')
set_log_level(PROGRESS)

# Time-stepping
t = 0
for n in range(num_steps):

    # Update current time
    t += dt

    # Read velocity from file
    timeseries_w.retrieve(w.vector(), t)

    # Solve variational problem for time step
    solve(F == 0, u)

    # Save solution to file (VTK)
    _u_1, _u_2, _u_3 = u.split()
    vtkfile_u_1 << (_u_1, t)
    vtkfile_u_2 << (_u_2, t)
    vtkfile_u_3 << (_u_3, t)

    # Update previous solution
    u_n.assign(u)

    # Update progress bar
    progress.update(t / T)

# Hold plot
interactive()
```

This example program can be found in the file `ft09_reaction_system.py`.

Finally, we comment on three important techniques that are very useful when working with systems of PDEs: setting initial conditions, setting boundary conditions, and extracting components of the system for plotting or postprocessing.

Setting initial conditions for mixed systems. In our example, we did not need to worry about setting an initial condition, since we start with $u_1 = u_2 = u_3 = 0$. This happens automatically in the code when we set `u_n = Function(V)`. This creates a `Function` for the whole system and all degrees of freedom are set to zero.

If we want to set initial conditions for the components of the system separately, the easiest solution is to define the initial conditions as a vector-valued `Expression` and then project (or interpolate) this to the `Function` representing the whole system. For example,

```
u_0 = Expression(('sin(x[0])', 'cos(x[0]*x[1])', 'exp(x[1])'), degree=1)
u_n = project(u_0, V)
```

This defines u_1, u_2, and u_2 to be the projections of $\sin x$, $\cos(xy)$, and $\exp(y)$, respectively.

Setting boundary conditions for mixed systems. In our example, we also did not need to worry about setting boundary conditions since we used a natural Neumann condition. If we want to set Dirichlet conditions for individual components of the system, this can be done as usual by the class `DirichletBC`, but we must specify for which subsystem we set the boundary condition. For example, to specify that u_2 should be equal to xy on the boundary defined by `boundary`, we do

```
u_D = Expression('x[0]*x[1]', degree=1)
bc = DirichletBC(V.sub(1), u_D, boundary)
```

The object `bc` or a list of such objects containing different boundary conditions, can then be passed to the `solve` function as usual. Note that numbering starts at 0 in FEniCS so the subspace corresponding to u_2 is `V.sub(1)`.

Accessing components of mixed systems. If `u` is a `Function` defined on a mixed function space in FEniCS, there are several ways in which `u` can be *split* into components. Above we already saw an example of the first of these:

```
u_1, u_2, u_3 = split(u)
```

This extracts the components of `u` as *symbols* that can be used in a variational problem. The above statement is in fact equivalent to

```
u_1 = u[0]
u_2 = u[1]
u_3 = u[2]
```

Note that `u[0]` is not really a `Function` object, but merely a symbolic expression, just like `grad(u)` in FEniCS is a symbolic expression and not a `Function` representing the gradient. This means that `u_1`, `u_2`, `u_3` can be used in a variational problem, but cannot be used for plotting or postprocessing.

To access the components of `u` for plotting and saving the solution to file, we need to use a different variant of the `split` function:

```
u_1_, u_2_, u_3_ = u.split()
```

This returns three subfunctions as actual objects with access to the common underlying data stored in `u`, which makes plotting and saving to file possible. Alternatively, we can do

```
u_1_, u_2_, u_3_ = u.split(deepcopy=True)
```

which will create `u_1_`, `u_2_`, and `u_3_` as stand-alone `Function` objects, each holding a copy of the subfunction data extracted from `u`. This is useful in many situations but is not necessary for plotting and saving solutions to file.

Chapter 4
Subdomains and boundary conditions

So far, we have only looked briefly at how to specify boundary conditions. In this chapter, we look more closely at how to specify boundary conditions on specific parts (subdomains) of the boundary and how to combine multiple boundary conditions. We will also look at how to generate meshes with subdomains and how to define coefficients with different values in different subdomains.

4.1 Combining Dirichlet and Neumann conditions

Let's return to the Poisson problem from Chapter 2 and see how to extend the mathematics and the implementation to handle a Dirichlet condition in combination with a Neumann condition. The domain is still the unit square, but now we set the Dirichlet condition $u = u_{\mathrm{D}}$ at the left and right sides, $x = 0$ and $x = 1$, while the Neumann condition

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

is applied to the remaining sides $y = 0$ and $y = 1$.

4.1.1 PDE problem

Let Γ_{D} and Γ_{N} denote the parts of the boundary $\partial\Omega$ where the Dirichlet and Neumann conditions apply, respectively. The complete boundary-value problem can be written as

$$-\nabla^2 u = f \quad \text{in } \Omega, \tag{4.1}$$

$$u = u_{\mathrm{D}} \text{ on } \Gamma_{\mathrm{D}}, \tag{4.2}$$

$$-\frac{\partial u}{\partial n} = g \quad \text{on } \Gamma_{\mathrm{N}}. \tag{4.3}$$

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

$$\begin{aligned} f(x,y) &= -6, \\ g(x,y) &= \begin{cases} 0, & y = 0, \\ 4, & y = 1, \end{cases} \\ u_{\mathrm{D}}(x,y) &= 1 + x^2 + 2y^2. \end{aligned}$$

For ease of programming, we define g as a function over the whole domain Ω such that g takes on the correct values at $y = 0$ and $y = 1$. One possible extension is

$$g(x,y) = 4y.$$

4.1.2 Variational formulation

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

$$-\int_\Omega (\nabla^2 u) v \,\mathrm{d}x = \int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x - \int_{\partial\Omega} \frac{\partial u}{\partial n} v \,\mathrm{d}s,$$

and since $v = 0$ on Γ_{D},

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

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

$$\int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x = \int_\Omega f v \,\mathrm{d}x - \int_{\Gamma_{\mathrm{N}}} g v \,\mathrm{d}s. \tag{4.4}$$

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

$$a(u,v) = \int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x, \tag{4.5}$$

$$L(v) = \int_\Omega f v \,\mathrm{d}x - \int_{\Gamma_\mathrm{N}} g v \,\mathrm{d}s\,. \tag{4.6}$$

4.1.3 FEniCS implementation

How does the Neumann condition impact the implementation? Let us revisit our previous implementation `ft01_poisson.py` from Section 2.2 and examine which changes we need to make to incorporate the Neumann condition. It turns out that only two changes are necessary:

- The function `boundary` defining the Dirichlet boundary must be modified.
- The new boundary term must be added to the expression for `L`.

The first adjustment can be coded as

```
tol = 1E-14

def boundary_D(x, on_boundary):
    if on_boundary:
        if near(x[0], 0, tol) or near(x[0], 1, tol):
            return True
        else:
            return False
    else:
        return False
```

A more compact implementation reads

```
def boundary_D(x, on_boundary):
    return on_boundary and (near(x[0], 0, tol) or near(x[0], 1, tol))
```

The second adjustment of our program concerns the definition of `L`, which needs to include the Neumann condition:

```
g = Expression('4*x[1]', degree=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 other modifications are necessary.

Note that the integration `*ds` is carried out over the entire boundary, including the Dirichlet boundary. However, since the test function `v` vanishes on the Dirichlet boundary (as a result specifying a `DirichletBC`), the integral will only include the contribution from the Neumann boundary.

4.2 Setting multiple Dirichlet conditions

In the previous section, we used a single function $u_{\mathrm{D}}(x,y)$ for setting Dirichlet conditions on two parts of the boundary. Often it is more practical to use multiple functions, one for each subdomain of the boundary. Let us return to the case from Section 4.1 and redefine the problem in terms of two Dirichlet conditions:

$$\begin{aligned} -\nabla^2 u &= f \quad \text{in } \Omega, \\ u &= u_{\mathrm{L}} \quad \text{on } \Gamma_{\mathrm{D}}^{\mathrm{L}}, \\ u &= u_{\mathrm{R}} \quad \text{on } \Gamma_{\mathrm{D}}^{\mathrm{R}}, \\ -\frac{\partial u}{\partial n} &= g \quad \text{on } \Gamma_{\mathrm{N}}. \end{aligned}$$

Here, $\Gamma_{\mathrm{D}}^{\mathrm{L}}$ is the left boundary $x=0$, while $\Gamma_{\mathrm{D}}^{\mathrm{R}}$ is the right boundary $x=1$. We note that $u_{\mathrm{L}}(x,y) = 1+2y^2$, $u_{\mathrm{R}}(x,y) = 2+2y^2$, and $g(x,y) = 4y$.

For the boundary condition on $\Gamma_{\mathrm{D}}^{\mathrm{L}}$, we define the usual triple of an expression for the boundary value, a function defining the location of the boundary, and a `DirichletBC` object:

```
u_L = Expression('1 + 2*x[1]*x[1]', degree=2)

def boundary_L(x, on_boundary):
    tol = 1E-14
    return on_boundary and near(x[0], 0, tol)

bc_L = DirichletBC(V, u_L, boundary_L)
```

For the boundary condition on $\Gamma_{\mathrm{D}}^{\mathrm{R}}$, we write a similar code snippet:

```
u_R = Expression('2 + 2*x[1]*x[1]', degree=2)

def boundary_R(x, on_boundary):
    tol = 1E-14
    return on_boundary and near(x[0], 1, tol)

bc_R = DirichletBC(V, u_R, boundary_R)
```

We collect the two boundary conditions in a list which we can pass to the `solve` function to compute the solution:

```
bcs = [bc_L, bc_R]
...
solve(a == L, u, bcs)
```

Note that for boundary values that do not depend on x or y, we might replace the `Expression` objects by `Constant` objects.

4.3 Defining subdomains for different materials

Solving PDEs in domains made up of different materials is a frequently encountered task. In FEniCS, these kinds of problems are handled by defining subdomains inside the domain. A simple example with two materials (subdomains) in 2D will demonstrate the idea. We consider the following variable-coefficient extension of the Poisson equation from Chapter 2:

$$-\nabla\cdot[\kappa(x,y)\nabla u(x,y)] = f(x,y), \tag{4.7}$$

in some domain Ω. Physically, this problem may be viewed as a model of heat conduction, with variable heat conductivity $\kappa(x,y) \geq \underline{\kappa} > 0$.

For illustration purposes, we consider the domain $\Omega = [0,1]\times[0,1]$ and divide it into two equal subdomains, as depicted in Figure 4.1:

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

We define $\kappa(x,y) = \kappa_0$ in Ω_0 and $\kappa(x,y) = \kappa_1$ in Ω_1, where $\kappa_0, \kappa_1 > 0$ are given constants.

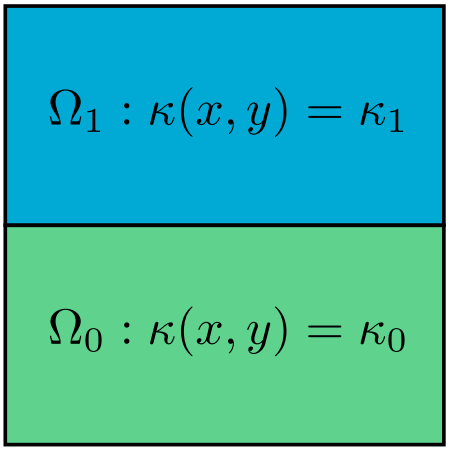


Fig. 4.1 Two subdomains with different material parameters.

The variational formulation may be easily expressed in FEniCS as follows:

```
a = kappa*dot(grad(u), grad(v))*dx
L = f*v*dx
```

In the remainder of this section, we will discuss different strategies for defining the coefficient `kappa` as an `Expression` that takes on different values in the two subdomains.

4.3.1 Using expressions to define subdomains

The simplest way to implement a variable coefficient $\kappa = \kappa(x, y)$ is to define an `Expression` which depends on the coordinates x and y. We have previously used the `Expression` class to define expressions based on simple formulas. Aternatively, an `Expression` can be defined as a Python class which allows for more complex logic. The following code snippet illustrates this construction:

```
class K(Expression):
    def set_k_values(self, k_0, k_1):
        self.k_0, self.k_1 = k_0, k_1

    def eval(self, value, x):
        "Set value[0] to value at point x"
        tol = 1E-14
        if x[1] <= 0.5 + tol:
            value[0] = self.k_0
        else:
            value[0] = self.k_1

# Initialize kappa
kappa = K()
kappa.set_k_values(1, 0.01)
```

The `eval` method gives great flexibility in defining functions, but a downside is that FEniCS will call `eval` in Python for each node `x`, which is a slow process.

An alternative method is to use a C++ string expression as we have seen before, which is much more efficient in FEniCS. This can be done using an inline if test:

```
tol = 1E-14
k_0 = 1.0
k_1 = 0.01
kappa = Expression('x[1] <= 0.5 + tol ? k_0 : k_1', degree=0,
                   tol=tol, k_0=k_0, k_1=k_1)
```

This method of defining variable coefficients works if the subdomains are simple shapes that can be expressed in terms of geometric inequalities. However, for more complex subdomains, we will need to use a more general technique, as we will see next.

4.3.2 Using mesh functions to define subdomains

We now address how to specify the subdomains Ω_0 and Ω_1 using a more general technique. This technique involves the use of two classes that are essential in FEniCS when working with subdomains: `SubDomain` and `MeshFunction`. Consider the following definition of the boundary $x = 0$:

```
def boundary(x, on_boundary):
    tol = 1E-14
    return on_boundary and near(x[0], 0, tol)
```

This boundary definition is actually a shortcut to the more general FEniCS concept `SubDomain`. A `SubDomain` is a class which defines a region in space (a subdomain) in terms of a member function `inside` which returns `True` for points that belong to the subdomain and `False` for points that don't belong to the subdomain. Here is how to specify the boundary $x = 0$ as a `SubDomain`:

```
class Boundary(SubDomain):
    def inside(self, x, on_boundary):
        tol = 1E-14
        return on_boundary and near(x[0], 0, tol)

boundary = Boundary()
bc = DirichletBC(V, Constant(0), boundary)
```

We notice that the `inside` function of the class `Boundary` is (almost) identical to the previous boundary definition in terms of the `boundary` function. Technically, our class `Boundary` is a *subclass* of the FEniCS class `SubDomain`.

We will use two `SubDomain` subclasses to define the two subdomains Ω_0 and Ω_1:

```
tol = 1E-14

class Omega_0(SubDomain):
    def inside(self, x, on_boundary):
        return x[1] <= 0.5 + tol

class Omega_1(SubDomain):
    def inside(self, x, on_boundary):
        return x[1] >= 0.5 - tol
```

Notice the use of `<=` and `>=` in both tests. FEniCS will call the `inside` function for each vertex in a cell to determine whether or not the cell belongs to a particular subdomain. For this reason, it is important that the test holds for all vertices in cells aligned with the boundary. In addition, we use a tolerance to make sure that vertices on the internal boundary at $y = 0.5$ will belong to *both* subdomains. This is a little counter-intuitive, but is necessary to make the cells both above and below the internal boundary belong to either Ω_0 or Ω_1.

To define the variable coefficient κ, we will use a powerful tool in FEniCS called a `MeshFunction`. A `MeshFunction` is a discrete function that can be evaluated at a set of so-called *mesh entities*. A mesh entity in FEniCS is either a vertex, an edge, a face, or a cell (triangle or tetrahedron). A `MeshFunction` over cells is suitable to represent subdomains (materials), while a `MeshFunction` over facets (edges or faces) is used to represent pieces of external or internal boundaries. A `MeshFunction` over cells can also be used to represent boundary markers for mesh refinement. A FEniCS `MeshFunction`

is parameterized both over its data type (like integers or booleans) and its dimension (0 = vertex, 1 = edge etc.). Special subclasses `VertexFunction`, `EdgeFunction` etc. are provided for easy definition of a `MeshFunction` of a particular dimension.

Since we need to define subdomains of Ω in the present example, we make use of a `CellFunction`. The constructor is given two arguments: (1) the type of value: `'int'` for integers, `'size_t'` for non-negative (unsigned) integers, `'double'` for real numbers, and `'bool'` for logical values; (2) a `Mesh` object. Alternatively, the constructor can take just a filename and initialize the `CellFunction` from data in a file.

We first create a `CellFunction` with non-negative integer values (`'size_t'`):

```
materials = CellFunction('size_t', mesh)
```

Next, we use the two subdomains to *mark* the cells belonging to each subdomain:

```
subdomain_0 = Omega_0()
subdomain_1 = Omega_1()
subdomain_0.mark(materials, 0)
subdomain_1.mark(materials, 1)
```

This will set the values of the mesh function `materials` to 0 on each cell belonging to Ω_0 and 1 on all cells belonging to Ω_1. Alternatively, we can use the following equivalent code to mark the cells:

```
materials.set_all(0)
subdomain_1.mark(materials, 1)
```

To examine the values of the mesh function and see that we have indeed defined our subdomains correctly, we can simply plot the mesh function:

```
plot(materials, interactive=True)
```

We may also wish to store the values of the mesh function for later use:

```
File('materials.xml.gz') << materials
```

which can later be read back from file as follows:

```
File('materials.xml.gz') >> materials
```

Now, to use the values of the mesh function `materials` to define the variable coefficient κ, we create a FEniCS `Expression`:

```
class K(Expression):
    def __init__(self, materials, k_0, k_1, **kwargs):
        self.materials = materials
        self.k_0 = k_0
        self.k_1 = k_1

def eval_cell(self, values, x, cell):
        if self.materials[cell.index] == 0:
```

```
            values[0] = self.k_0
        else:
            values[0] = self.k_1

kappa = K(materials, k_0, k_1, degree=0)
```

This is similar to the `Expression` subclass we defined above, but we make use of the member function `eval_cell` in place of the regular `eval` function. This version of the evaluation function has an additional `cell` argument which we can use to check on which cell we are currently evaluating the function. We also defined the special function `__init__` (the constructor) so that we can pass all data to the `Expression` when it is created.

Since we make use of geometric tests to define the two `SubDomains` for Ω_0 and Ω_1, the `MeshFunction` method may seem like an unnecessary complication of the simple method using an `Expression` with an if-test. However, in general the definition of subdomains may be available as a `MeshFunction` (from a data file), perhaps generated as part of the mesh generation process, and not as a simple geometric test. In such cases the method demonstrated here is the recommended way to work with subdomains.

4.3.3 Using C++ code snippets to define subdomains

The `SubDomain` and `Expression` Python classes are very convenient, but their use leads to function calls from C++ to Python for each node in the mesh. Since this involves a significant cost, we need to make use of C++ code if performance is an issue.

Instead of writing the `SubDomain` subclass in Python, we may instead use the `CompiledSubDomain` tool in FEniCS to specify the subdomain in C++ code and thereby speed up our code. Consider the definition of the classes `Omega_0` and `Omega_1` above in Python. The key strings that define these subdomains can be expressed in C++ syntax and given as arguments to `CompiledSubDomain` as follows:

```
tol = 1E-14
subdomain_0 = CompiledSubDomain('x[1] <= 0.5 + tol', tol=tol)
subdomain_1 = CompiledSubDomain('x[1] >= 0.5 - tol', tol=tol)
```

As seen, parameters can be specified using keyword arguments. The resulting objects, `subdomain_0` and `subdomain_1`, can be used as ordinary `SubDomain` objects.

Compiled subdomain strings can be applied for specifying boundaries as well:

```
boundary_R = CompiledSubDomain('on_boundary && near(x[0], 1, tol)',
                               tol=1E-14)
```

It is also possible to feed the C++ string (without parameters) directly as the third argument to `DirichletBC` without explicitly constructing a `CompiledSubDomain` object:

```
bc1 = DirichletBC(V, value, 'on_boundary && near(x[0], 1, tol)')
```

Python `Expression` classes may also be redefined using C++ for more efficient code. Consider again the definition of the class `K` above for the variable coefficient $\kappa = \kappa(x)$. This may be redefined using a C++ code snippet and the keyword `cppcode` to the regular FEniCS `Expression` class:

```
cppcode = """
class K : public Expression
{
public:

  void eval(Array<double>& values,
            const Array<double>& x,
            const ufc::cell& cell) const
  {
    if ((*materials)[cell.index] == 0)
      values[0] = k_0;
    else
      values[0] = k_1;
  }

  std::shared_ptr<MeshFunction<std::size_t>> materials;
  double k_0;
  double k_1;

};
"""

kappa = Expression(cppcode=cppcode, degree=0)
kappa.materials = materials
kappa.k_0 = k_0
kappa.k_1 = k_1
```

4.4 Setting multiple Dirichlet, Neumann, and Robin conditions

Consider again the variable-coefficient Poisson problem from Section 4.3. We will now discuss how to implement general combinations of boundary conditions of Dirichlet, Neumann, and Robin type for this model problem.

4.4.1 Three types of boundary conditions

We extend our repertoire of boundary conditions to three types: Dirichlet, Neumann, and Robin. Dirichlet conditions apply to some parts $\Gamma_{\mathrm{D}}^0, \Gamma_{\mathrm{D}}^1, \ldots$, of the boundary:

$$u = u_{\mathrm{D}}^0 \text{ on } \Gamma_{\mathrm{D}}^0, \quad u = u_{\mathrm{D}}^1 \text{ on } \Gamma_{\mathrm{D}}^1, \quad \ldots,$$

where u_{D}^i are prescribed functions, $i = 0, 1, \ldots$. On other parts, $\Gamma_{\mathrm{N}}^0, \Gamma_{\mathrm{N}}^1, \ldots$, we have Neumann conditions:

$$-\kappa \frac{\partial u}{\partial n} = g_0 \text{ on } \Gamma_{\mathrm{N}}^0, \quad -\kappa \frac{\partial u}{\partial n} = g_1 \text{ on } \Gamma_{\mathrm{N}}^1, \quad \ldots,$$

Finally, we have *Robin conditions*:

$$-\kappa \frac{\partial u}{\partial n} = r(u - s),$$

where r and s are specified functions. The Robin condition is most often used to model heat transfer to the surroundings and arise naturally from Newton's cooling law. In that case, r is a heat transfer coefficient, and s is the temperature of the surroundings. Both can be space and time-dependent. The Robin conditions apply at some parts $\Gamma_{\mathrm{R}}^0, \Gamma_{\mathrm{R}}^1, \ldots$, of the boundary:

$$-\kappa \frac{\partial u}{\partial n} = r_0(u - s_0) \text{ on } \Gamma_{\mathrm{R}}^0, \quad -\kappa \frac{\partial u}{\partial n} = r_1(u - s_1) \text{ on } \Gamma_{\mathrm{R}}^1, \quad \ldots$$

4.4.2 PDE problem

With the notation above, the model problem to be solved with multiple Dirichlet, Neumann, and Robin conditions can be formulated as follows:

$$-\nabla \cdot (\kappa \nabla u) = -f \qquad \text{in } \Omega, \tag{4.8}$$

$$u = u_{\mathrm{D}}^i \qquad \text{on } \Gamma_{\mathrm{D}}^i, \quad i = 0, 1, \ldots \tag{4.9}$$

$$-\kappa \frac{\partial u}{\partial n} = g_i \qquad \text{on } \Gamma_{\mathrm{N}}^i, \quad i = 0, 1, \ldots \tag{4.10}$$

$$-\kappa \frac{\partial u}{\partial n} = r_i(u - s_i) \quad \text{on } \Gamma_{\mathrm{R}}^i, \quad i = 0, 1, \ldots \tag{4.11}$$

4.4.3 Variational formulation

As usual, we multiply by a test function v and integrate by parts:

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

On the Dirichlet part of the boundary (Γ_{D}^i), the boundary integral vanishes since $v=0$. On the remaining part of the boundary, we split the boundary integral into contributions from the Neumann parts (Γ_{N}^i) and Robin parts (Γ_{R}^i). Inserting the boundary conditions, we obtain

$$\begin{aligned}-\int_{\partial\Omega}\kappa\frac{\partial u}{\partial n}v\,\mathrm{d}s &= -\sum_i\int_{\Gamma_{\mathrm{N}}^i}\kappa\frac{\partial u}{\partial n}\,\mathrm{d}s - \sum_i\int_{\Gamma_{\mathrm{R}}^i}\kappa\frac{\partial u}{\partial n}\,\mathrm{d}s\\ &= \sum_i\int_{\Gamma_{\mathrm{N}}^i} g_i\,\mathrm{d}s + \sum_i\int_{\Gamma_{\mathrm{R}}^i} r_i(u-s_i)\,\mathrm{d}s\,.\end{aligned}$$

We thus obtain the following variational problem:

$$F = \int_\Omega \kappa\nabla u\cdot\nabla v\,\mathrm{d}x + \sum_i\int_{\Gamma_{\mathrm{N}}^i} g_i v\,\mathrm{d}s + \sum_i\int_{\Gamma_{\mathrm{R}}^i} r_i(u-s_i)v\,\mathrm{d}s - \int_\Omega fv\,\mathrm{d}x = 0\,. \tag{4.12}$$

We have been used to writing this variational formulation in the standard notation $a(u,v)=L(v)$, which requires that we identify all integral depending on the trial function u, and collect these in $a(u,v)$, while the remaining integrals go into $L(v)$. The integrals from the Robin condition must for this reason be split into two parts:

$$\int_{\Gamma_{\mathrm{R}}^i} r_i(u-s_i)v\,\mathrm{d}s = \int_{\Gamma_{\mathrm{R}}^i} r_i uv\,\mathrm{d}s - \int_{\Gamma_{\mathrm{R}}^i} r_i s_i v\,\mathrm{d}s\,.$$

We then have

$$a(u,v) = \int_\Omega \kappa\nabla u\cdot\nabla v\,\mathrm{d}x + \sum_i\int_{\Gamma_{\mathrm{R}}^i} r_i uv\,\mathrm{d}s, \tag{4.13}$$

$$L(v) = \int_\Omega fv\,\mathrm{d}x - \sum_i\int_{\Gamma_{\mathrm{N}}^i} g_i v\,\mathrm{d}s + \sum_i\int_{\Gamma_{\mathrm{R}}^i} r_i s_i v\,\mathrm{d}s\,. \tag{4.14}$$

Alternatively, we may keep the formulation (4.12) and either solve the variational problem as a nonlinear problem (`F == 0`) in FEniCS or use the FEniCS functions `lhs` and `rhs` to extract the bilinear and linear parts of `F`:

```
a = lhs(F)
```

```
L = rhs(F)
```

Note that if we choose to solve this linear problem as a nonlinear problem, the Newton iteration will converge in a single iteration.

4.4.4 FEniCS implementation

Let us examine how to extend our Poisson solver to handle general combinations of Dirichlet, Neumann, and Robin boundary conditions. Compared to our previous code, we must consider the following extensions:

- Defining markers for the different parts of the boundary.
- Splitting the boundary integral into parts using the markers.

A general approach to the first task is to mark each of the desired boundary parts with markers 0, 1, 2, and so forth. Here we aim at the four sides of the unit square, marked with 0 ($x = 0$), 1 ($x = 1$), 2 ($y = 0$), and 3 ($y = 1$). The markers will be defined using a `MeshFunction`, but contrary to Section 4.3, this is not a function over cells, but a function over the facets of the mesh. We use a `FacetFunction` for this purpose:

```
boundary_markers = FacetFunction('size_t', mesh)
```

As in Section 4.3 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 boundary $x = 0$ can be marked as follows:

```
class BoundaryX0(SubDomain):
    tol = 1E-14
    def inside(self, x, on_boundary):
        return on_boundary and near(x[0], 0, tol)

bx0 = BoundaryX0()
bx0.mark(boundary_markers, 0)
```

Similarly, we create the classes `BoundaryX1` ($x = 1$), `BoundaryY0` ($y = 0$), and `BoundaryY1` ($y = 1$) boundary, and mark these as subdomains 1, 2, and 3, respectively.

For generality of the implementation, we let the user specify what kind of boundary condition that applies to each of the four boundaries. We set up a Python dictionary for this purpose, with the key as subdomain number and the value as a dictionary specifying the kind of condition as key and a function as its value. For example,

```
boundary_conditions = {0: {'Dirichlet': u_D},
                       1: {'Robin':     (r, s)},
                       2: {'Neumann':   g},
                       3: {'Neumann',   0}}
```

specifies

- a Dirichlet condition $u = u_{\mathrm{D}}$ for $x = 0$;
- a Robin condition $-\kappa\partial_n u = r(u - s)$ for $x = 1$;
- a Neumann condition $-\kappa\partial_n u = g$ for $y = 0$;
- a Neumann condition $-\kappa\partial_n u = 0$ for $y = 1$.

As explained in Section 4.2, multiple Dirichlet conditions must be collected in a list of `DirichletBC` objects. Based on the `boundary_conditions` data structure above, we can construct this list by the following code snippet:

```
bcs = []
for i in boundary_conditions:
    if 'Dirichlet' in boundary_conditions[i]:
        bc = DirichletBC(V, boundary_conditions[i]['Dirichlet'],
                         boundary_markers, i)
        bcs.append(bc)
```

A new aspect of the variational problem is the two distinct boundary integrals over Γ_{N}^i and Γ_{R}^i. Having a mesh function over exterior cell facets (our `boundary_markers` 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.

To express integrals over the boundary parts using `ds(i)`, we must first redefine the measure `ds` in terms of our boundary markers:

```
ds = Measure('ds', domain=mesh, subdomain_data=boundary_markers)
```

Similarly, if we want integration over different parts of the domain, we redefine `dx` as

```
dx = Measure('dx', domain=mesh, subdomain_data=domain_markers)
```

where `domain_markers` is a `CellFunction` defining subdomains in Ω.

Suppose we have a Robin condition with values `r` and `s` on subdomain `R`, and a Neumann condition with value `g` on subdomain `N`. The variational form can then be written

```
a = kappa*dot(grad(u), grad(v))*dx + r*u*v*ds(R)
L = f*v*dx - g*v*ds(N) + r*s*v*ds(R)
```

In our case, things get a bit more complicated since the information about integrals in Neumann and Robin conditions are in the `boundary_conditions` data structure. We can collect all Neumann conditions by the following code snippet:

```
integrals_N = []
for i in boundary_conditions:
```

```
    if 'Neumann' in boundary_conditions[i]:
        if boundary_conditions[i]['Neumann'] != 0:
            g = boundary_conditions[i]['Neumann']
            integrals_N.append(g*v*ds(i))
```

Applying `sum(integrals_N)` will apply the `+` operator to the variational forms in the `integrals_N` list and result in the integrals we need for the right-hand side `L` of the variational form.

The integrals in the Robin condition can similarly be collected in lists:

```
integrals_R_a = []
integrals_R_L = []
for i in boundary_conditions:
    if 'Robin' in boundary_conditions[i]:
        r, s = boundary_conditions[i]['Robin']
        integrals_R_a.append(r*u*v*ds(i))
        integrals_R_L.append(r*s*v*ds(i))
```

We are now in a position to define the `a` and `L` expressions in the variational formulation:

```
a = kappa*dot(grad(u), grad(v))*dx + sum(integrals_R_a)
L = f*v*dx - sum(integrals_N) + sum(integrals_R_L)
```

Alternatively, we may use the FEniCS functions `lhs` and `rhs` as mentioned above to simplify the extraction of terms for the Robin integrals:

```
integrals_R = []
for i in boundary_conditions:
    if 'Robin' in boundary_conditions[i]:
        r, s = boundary_conditions[i]['Robin']
        integrals_R.append(r*(u - s)*v*ds(i))

F = kappa*dot(grad(u), grad(v))*dx + \
    sum(integrals_R) - f*v*dx + sum(integrals_N)
a, L = lhs(F), rhs(F)
```

This time we can more naturally define the integrals from the Robin condition as `r*(u - s)*v*ds(i)`.

The complete code can be found in the function `solver_bcs` in the program `ft10_poisson_extended.py`.

4.4.5 Test problem

We will use the same exact solution $u_e = 1 + x^2 + 2y^2$ as in Chapter 2, and thus take $\kappa = 1$ and $f = -6$. Our domain is the unit square, and we assign Dirichlet conditions at $x = 0$ and $x = 1$, a Robin condition at $y = 0$, and a Neumann condition at $y = 1$. With the given exact solution u_e, we realize that the Neumann condition at $y = 1$ is $-\partial u/\partial n = -\partial u/\partial y = 4y = 4$, while the Robin condition at $y = 0$ can be selected in many ways. Since $-\partial u/\partial n = \partial u/\partial y = 0$

at $y = 0$, we can select $s = u_e$ and specify $r \neq 0$ arbitrarily in the Robin condition. We will set $r = 1000$ and $s = u_e$.

The boundary parts are thus Γ_D^0: $x = 0$, Γ_D^1: $x = 1$, Γ_R^0: $y = 0$, and Γ_N^0: $y = 1$.

When implementing this test problem, and especially other test problems with more complicated expressions, it is advantageous to use symbolic computing. Below we define the exact solution as a `sympy` expression and derive other functions from their mathematical definitions. Then we turn these expressions into C/C++ code, which can then be used to define `Expression` objects.

```
# Define manufactured solution in sympy and derive f, g, etc.
import sympy as sym
x, y = sym.symbols('x[0], x[1]')            # needed by UFL
u = 1 + x**2 + 2*y**2                       # exact solution
u_e = u                                     # exact solution
u_00 = u.subs(x, 0)                         # restrict to x = 0
u_01 = u.subs(x, 1)                         # restrict to x = 1
f = -sym.diff(u, x, 2) - sym.diff(u, y, 2)  # -Laplace(u)
f = sym.simplify(f)                         # simplify f
g = -sym.diff(u, y).subs(y, 1)              # compute g = -du/dn
r = 1000                                    # Robin data, arbitrary
s = u                                       # Robin data, u = s

# Collect variables
variables = [u_e, u_00, u_01, f, g, r, s]

# Turn into C/C++ code strings
variables = [sym.printing.ccode(var) for var in variables]

# Turn into FEniCS Expressions
variables = [Expression(var, degree=2) for var in variables]

# Extract variables
u_e, u_00, u_01, f, g, r, s = variables

# Define boundary conditions
boundary_conditions = {0: {'Dirichlet': u_00},   # x = 0
                       1: {'Dirichlet': u_01},   # x = 1
                       2: {'Robin':     (r, s)}, # y = 0
                       3: {'Neumann':   g}}      # y = 1
```

The complete code can be found in the function `demo_bcs` in the program `ft10_poisson_extended.py`.

4.4.6 Debugging boundary conditions

It is easy to make mistakes when implementing a problem with many different types of boundary conditions, as in the present case. One method to debug

boundary conditions is to run through all vertex coordinates and check if the `SubDomain.inside` method marks the vertex as on the boundary. Another useful method is to list which degrees of freedom that are subject to Dirichlet conditions, and for first-order Lagrange (P_1) elements, print the corresponding vertex coordinates as illustrated by the following code snippet:

```
if debug1:

    # Print all vertices that belong to the boundary parts
    for x in mesh.coordinates():
        if bx0.inside(x, True): print('%s is on x = 0' % x)
        if bx1.inside(x, True): print('%s is on x = 1' % x)
        if by0.inside(x, True): print('%s is on y = 0' % x)
        if by1.inside(x, True): print('%s is on y = 1' % x)

    # Print the Dirichlet conditions
    print('Number of Dirichlet conditions:', len(bcs))
    if V.ufl_element().degree() == 1:  # P1 elements
        d2v = dof_to_vertex_map(V)
        coor = mesh.coordinates()
        for i, bc in enumerate(bcs):
            print('Dirichlet condition %d' % i)
            boundary_values = bc.get_boundary_values()
            for dof in boundary_values:
                print('   dof %2d: u = %g' % (dof, boundary_values[dof]))
                if V.ufl_element().degree() == 1:
                    print('    at point %s' %
                          (str(tuple(coor[d2v[dof]].tolist()))))
```

Calls to the `inside` method

In the code snippet above, we call the inside method for each coordinate of the mesh. We could also place a printout inside the `inside` method. Then it will be surprising to see that this method is called not only for the points assoicated with degrees of freedom. For P_1 elements the method is also called for each midpoint on each facet of the cells. This is because a Dirichlet condition is by default set only if the entire facet can be said to be subject to the condition defining the boundary.

4.5 Generating meshes with subdomains

So far, we have worked mostly with simple meshes (the unit square) and defined boundaries and subdomains in terms of simple geometric tests like $x = 0$ or $y \leq 0.5$. For more complex geometries, it is not realistic to specify boundaries and subdomains in this way. Instead, the boundaries and subdomains must be defined as part of the mesh generation process. We will now

look at how to use the FEniCS mesh generation tool `mshr` to generate meshes and define subdomains.

4.5.1 PDE problem

We will again solve the Poisson equation, but this time for a different application. Consider an iron cylinder with copper wires wound around the cylinder as in Figure 4.2. Through the copper wires a static current $J = 1\,\mathrm{A}$ is flowing and we want to compute the magnetic field B in the iron cylinder, the copper wires, and the surrounding vacuum.

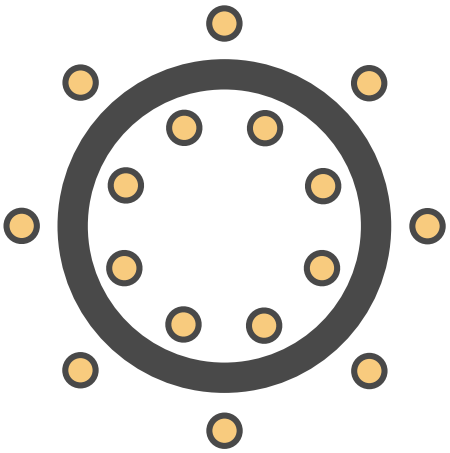


Fig. 4.2 Cross-section of an iron cylinder with copper wires wound around the cylinder, here with $n = 8$ windings. The inner circles are cross-sections of the copper wire coming up ("north") and the outer circles are cross-sections of the copper wire going down into the plane ("south").

First, we simplify the problem to a 2D problem. We can do this by assuming that the cylinder extends far along the z-axis and as a consequence the field is virtually independent of the z-coordinate. Next, we consider Maxwell's equation to derive a Poisson equation for the magnetic field (or rather its potential):

$$\nabla \cdot D = \varrho, \tag{4.15}$$
$$\nabla \cdot B = 0, \tag{4.16}$$
$$\nabla \times E = -\frac{\partial B}{\partial t}, \tag{4.17}$$
$$\nabla \times H = \frac{\partial D}{\partial t} + J. \tag{4.18}$$

Here, D is the displacement field, B is the magnetic field, E is the electric field, and H is the magnetizing field. In addition to Maxwell's equations, we also need a constitutive relation between B and H,

$$B = \mu H, \tag{4.19}$$

which holds for an isotropic linear magnetic medium. Here, μ is the magnetic permeability of the material. Now, since B is solenoidal (divergence free) according to Maxwell's equations, we know that B must be the curl of some vector field A. This field is called the magnetic vector potential. Since the problem is static and thus $\partial D/\partial t = 0$, it follows that

$$J = \nabla \times H = \nabla \times (\mu^{-1} B) = \nabla \times (\mu^{-1} \nabla \times A) = -\nabla \cdot (\mu^{-1} \nabla A). \tag{4.20}$$

In the last step, we have expanded the second derivatives and used the gauge freedom of A to simplify the equations to a simple vector-valued Poisson problem for the magnetic vector potential; if $B = \nabla \times A$, then $B = \nabla \times (A + \nabla\psi)$ for any scalar field ψ (the gauge function). For the current problem, we thus need to solve the following 2D Poisson problem for the z-component A_z of the magnetic vector potential:

$$-\nabla \cdot (\mu^{-1} \nabla A_z) = J_z \quad \text{in } \mathbb{R}^2, \tag{4.21}$$
$$\lim_{|(x,y)| \to \infty} A_z = 0. \tag{4.22}$$

Since we cannot solve this problem on an infinite domain, we will truncate the domain using a large disk and set $A_z = 0$ on the boundary. The current J_z is set to $+1\,\mathrm{A}$ in the interior set of circles (copper wire cross-sections) and to $-1\,\mathrm{A}$ in the exterior set of circles in Figure 4.2.

Once the magnetic vector potential has been computed, we can compute the magnetic field $B = B(x, y)$ by

$$B(x, y) = \left(\frac{\partial A_z}{\partial y}, -\frac{\partial A_z}{\partial x} \right). \tag{4.23}$$

4.5.2 Variational formulation

The variational problem is derived as before by multiplying the PDE with a test function v and integrating by parts. Since the boundary integral vanishes due to the Dirichlet condition, we obtain

$$\int_\Omega \mu^{-1} \nabla A_z \cdot \nabla v \, \mathrm{d}x = \int_\Omega J_z v \, \mathrm{d}x, \tag{4.24}$$

or, in other words, $a(A_z, v) = L(v)$ with

$$a(A_z, v) = \int_\Omega \mu^{-1} \nabla A_z \cdot \nabla v \, \mathrm{d}x, \tag{4.25}$$

$$L(v) = \int_\Omega J_z v \, \mathrm{d}x. \tag{4.26}$$

4.5.3 FEniCS implementation

The first step is to generate a mesh for the geometry described in Figure 4.2. We let a and b be the inner and outer radii of the iron cylinder and let c_1 and c_2 be the radii of the two concentric distributions of copper wire cross-sections. Furthermore, we let r be the radius of a copper wire, R be the radius of our domain, and n be the number of windings (giving a total of $2n$ copper-wire cross-sections). This geometry can be described easily using `mshr` and a little bit of Python programming:

```
# Define geometry for background
domain = Circle(Point(0, 0), R)

# Define geometry for iron cylinder
cylinder = Circle(Point(0, 0), b) - Circle(Point(0, 0), a)

# Define geometry for wires (N = North (up), S = South (down))
angles_N = [i*2*pi/n for i in range(n)]
angles_S = [(i + 0.5)*2*pi/n for i in range(n)]
wires_N = [Circle(Point(c_1*cos(v), c_1*sin(v)), r) for v in angles_N]
wires_S = [Circle(Point(c_2*cos(v), c_2*sin(v)), r) for v in angles_S]
```

The mesh that we generate will be a mesh of the entire disk with radius R but we need the mesh generation to respect the internal boundaries defined by the iron cylinder and the copper wires. We also want `mshr` to label the subdomains so that we can easily specify material parameters (μ) and currents. To do this, we use the `mshr` function `set_subdomain` as follows:

```
# Set subdomain for iron cylinder
domain.set_subdomain(1, cylinder)
```

```
# Set subdomains for wires
for (i, wire) in enumerate(wires_N):
    domain.set_subdomain(2 + i, wire)
for (i, wire) in enumerate(wires_S):
    domain.set_subdomain(2 + n + i, wire)
```

Once the subdomains have been created, we can generate the mesh:

```
mesh = generate_mesh(domain, 32)
```

A detail of the mesh is shown in Figure 4.3.

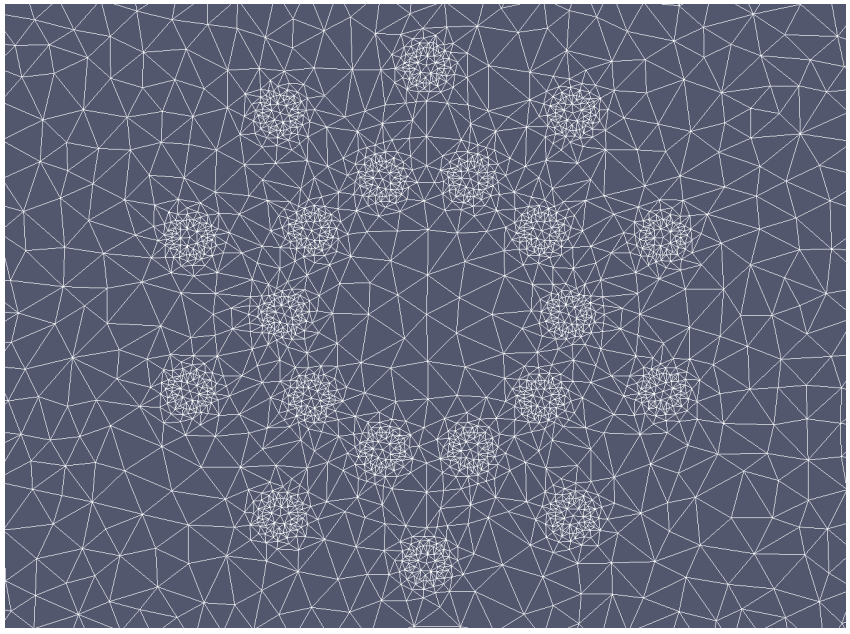


Fig. 4.3 Plot of (part of) the mesh generated for the magnetostatics test problem. The subdomains for the iron cylinder and copper wires are clearly visible

The mesh generated with `mshr` will contain information about the subdomains we have defined. To use this information in the definition of our variational problem and subdomain-dependent parameters, we will need to create a `MeshFunction` that marks the subdomains. This can be easily created by a call to the member function `mesh.domains`, which holds the subdomain data generated by `mshr`:

```
markers = MeshFunction('size_t', mesh, 2, mesh.domains())
```

This line creates a `MeshFunction` with unsigned integer values (the subdomain numbers) with dimension 2, which is the cell dimension for this 2D problem.

We can now use the markers as we have done before to redefine the integration measure `dx`:

```
dx = Measure('dx', domain=mesh, subdomain_data=markers)
```

Integrals over subdomains can then be expressed by `dx(0)`, `dx(1)`, and so on. We use this to define the current $J_z = \pm 1\,\mathrm{A}$ in the coppper wires:

```
J_N = Constant(1.0)
J_S = Constant(-1.0)
A_z = TrialFunction(V)
v = TestFunction(V)
a = (1 / mu)*dot(grad(A_z), grad(v))*dx
L_N = sum(J_N*v*dx(i) for i in range(2, 2 + n))
L_S = sum(J_S*v*dx(i) for i in range(2 + n, 2 + 2*n))
L = L_N + L_S
```

The permeability is defined as an `Expression` that depends on the subdomain number:

```
class Permeability(Expression):
    def __init__(self, mesh, **kwargs):
        self.markers = markers
    def eval_cell(self, values, x, ufc_cell):
        if markers[ufc_cell.index] == 0:
            values[0] = 4*pi*1e-7 # vacuum
        elif markers[ufc_cell.index] == 1:
            values[0] = 1e-5      # iron (should really be 2.5e-1)
        else:
            values[0] = -6.4e-6   # copper (yes, it's negative!)

mu = Permeability(mesh, degree=1)
```

As seen in this code snippet, we have used a somewhat less extreme value for the magnetic permeability of iron. This is to make the solution a little more interesting. It would otherwise be completely dominated by the field in the iron cylinder.

Finally, when A_z has been computed, we can compute the magnetic field:

```
W = VectorFunctionSpace(mesh, 'P', 1)
B = project(as_vector((A_z.dx(1), -A_z.dx(0))), W)
```

We use `as_vector` to interpret `(A_z.dx(1), -A_z.dx(0))` as a vector in the sense of the UFL form language, and not as a Python tuple. The resulting plots of the magnetic vector potential and magnetic field are shown in Figures 4.4 and 4.5.

The complete code for computing the magnetic field follows below.

```
from fenics import *
from mshr import *
from math import sin, cos, pi

a = 1.0   # inner radius of iron cylinder
b = 1.2   # outer radius of iron cylinder
```

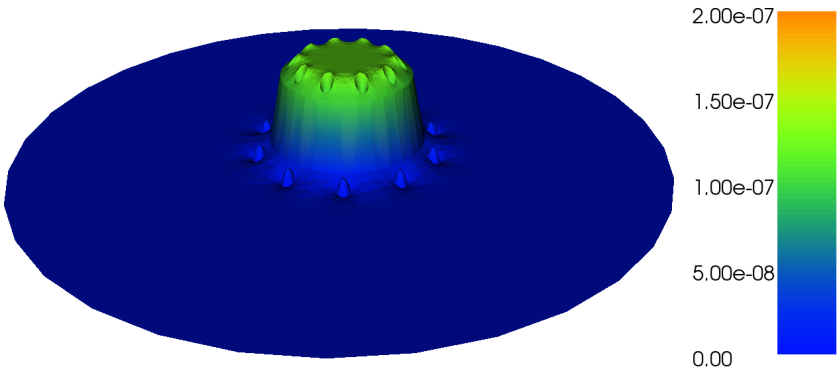


Fig. 4.4 Plot of the z-component A_z of the magnetic vector potential.

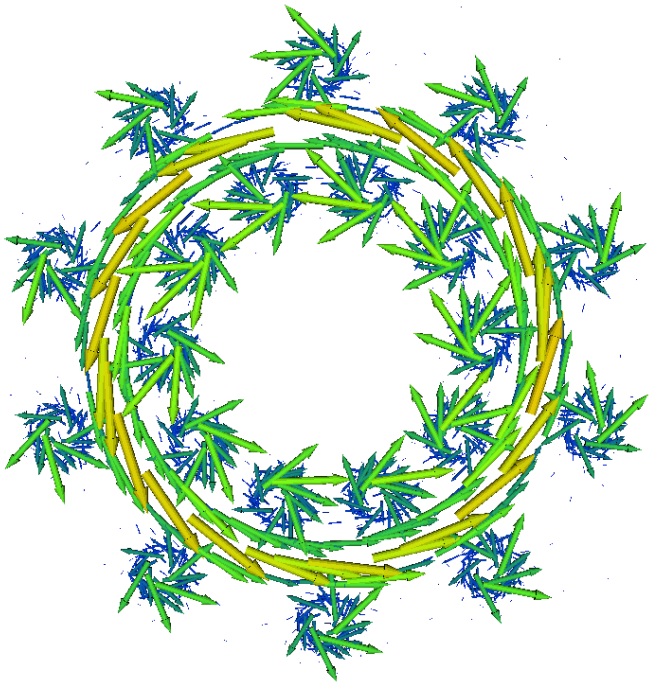


Fig. 4.5 Plot of the magnetic field B in the xy-plane.

```
c_1 = 0.8 # radius for inner circle of copper wires
c_2 = 1.4 # radius for outer circle of copper wires
r = 0.1   # radius of copper wires
R = 5.0   # radius of domain
n = 10    # number of windings

# Define geometry for background
```

```
domain = Circle(Point(0, 0), R)

# Define geometry for iron cylinder
cylinder = Circle(Point(0, 0), b) - Circle(Point(0, 0), a)

# Define geometry for wires (N = North (up), S = South (down))
angles_N = [i*2*pi/n for i in range(n)]
angles_S = [(i + 0.5)*2*pi/n for i in range(n)]
wires_N = [Circle(Point(c_1*cos(v), c_1*sin(v)), r) for v in angles_N]
wires_S = [Circle(Point(c_2*cos(v), c_2*sin(v)), r) for v in angles_S]

# Set subdomain for iron cylinder
domain.set_subdomain(1, cylinder)

# Set subdomains for wires
for (i, wire) in enumerate(wires_N):
    domain.set_subdomain(2 + i, wire)
for (i, wire) in enumerate(wires_S):
    domain.set_subdomain(2 + n + i, wire)

# Create mesh
mesh = generate_mesh(domain, 32)

# Define function space
V = FunctionSpace(mesh, 'P', 1)

# Define boundary condition
bc = DirichletBC(V, Constant(0), 'on_boundary')

# Define subdomain markers and integration measure
markers = MeshFunction('size_t', mesh, 2, mesh.domains())
dx = Measure('dx', domain=mesh, subdomain_data=markers)

# Define current densities
J_N = Constant(1.0)
J_S = Constant(-1.0)

# Define magnetic permeability
class Permeability(Expression):
    def __init__(self, mesh, **kwargs):
        self.markers = markers
    def eval_cell(self, values, x, cell):
        if markers[cell.index] == 0:
            values[0] = 4*pi*1e-7 # vacuum
        elif markers[cell.index] == 1:
            values[0] = 1e-5      # iron (should really be 2.5e-1)
        else:
            values[0] = -6.4e-6   # copper (yes, it's negative!)

mu = Permeability(mesh, degree=1)

# Define variational problem
A_z = TrialFunction(V)
v = TestFunction(V)
```

```
a = (1 / mu)*dot(grad(A_z), grad(v))*dx
L_N = sum(J_N*v*dx(i) for i in range(2, 2 + n))
L_S = sum(J_S*v*dx(i) for i in range(2 + n, 2 + 2*n))
L = L_N + L_S

# Solve variational problem
A_z = Function(V)
solve(a == L, A_z, bc)

# Compute magnetic field (B = curl A)
W = VectorFunctionSpace(mesh, 'P', 1)
B = project(as_vector((A_z.dx(1), -A_z.dx(0))), W)

# Plot solution
plot(A_z)
plot(B)

# Save solution to file
vtkfile_A_z = File('magnetostatics/potential.pvd')
vtkfile_B = File('magnetostatics/field.pvd')
vtkfile_A_z << A_z
vtkfile_B << B

# Hold plot
interactive()
```

This example program can be found in the file `ft11_magnetostatics.py`.

Chapter 5
Extensions: Improving the Poisson solver

The FEniCS programs we have written so far have been designed as flat Python scripts. This works well for solving simple demo problems. However, when you build a solver for an advanced application, you will quickly find the need for more structured programming. In particular, you may want to reuse your solver to solve a large number of problems where you vary the boundary conditions, the domain, and coefficients such as material parameters. In this chapter, we will see how to write general solver functions to improve the usability of FEniCS programs. We will also discuss how to utilize iterative solvers with preconditioners for solving linear systems, how to compute derived quantities, such as, e.g., the flux on a part of the boundary, and how to compute errors and convergence rates.

5.1 Refactoring the Poisson solver

Most programs discussed in this book are "flat"; that is, they are not organized into logical, reusable units in terms of Python functions. Such flat programs are useful for quickly testing ideas and sketching solution algorithms, but are not well suited for serious problem solving. We shall therefore look at how to *refactor* the Poisson solver from Chapter 2. For a start, this means splitting the code into functions. But refactoring is not just a reordering of existing statements. During refactoring, we also try to make the functions we create as reusable as possible in other contexts. We will also encapsulate statements specific to a certain problem into (non-reusable) functions. Being able to distinguish reusable code from specialized code is a key issue when refactoring code, and this ability depends on a good mathematical understanding of the problem at hand (what is general, what is special?). In a flat program, general and specialized code (and mathematics) are often mixed together, which tends to give a blurred understanding of the problem at hand.

5.1.1 A more general solver function

We consider the flat program `ft01_poisson.py` for solving the Poisson problem developed in Chapter 2. Some of the code in this program is needed to solve any Poisson problem $-\nabla^2 u = f$ on $[0,1]\times[0,1]$ with $u = u_{\mathrm{D}}$ on the boundary, while other statements arise from our simple test problem. Let us collect the general, reusable code in a function called `solver`. Our special test problem will then just be an application of our `solver` with some additional statements. We limit the `solver` function to just *compute the numerical solution*. Plotting and comparing the solution with the exact solution are considered to be problem-specific activities to be performed elsewhere.

We parameterize `solver` by f, u_{D}, and the resolution of the mesh. Since it is so trivial to use higher-order finite element functions by changing the third argument to `FunctionSpace`, we also add the polynomial degree of the finite element function space as an argument to `solver`.

```
from fenics import *
import numpy as np

def solver(f, u_D, Nx, Ny, degree=1):
    """
    Solve -Laplace(u) = f on [0,1] x [0,1] with 2*Nx*Ny Lagrange
    elements of specified degree and u = u_D (Expresssion) on
    the boundary.
    """

    # Create mesh and define function space
    mesh = UnitSquareMesh(Nx, Ny)
    V = FunctionSpace(mesh, 'P', degree)

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

    bc = DirichletBC(V, u_D, boundary)

    # Define variational problem
    u = TrialFunction(V)
    v = TestFunction(V)
    a = dot(grad(u), grad(v))*dx
    L = f*v*dx

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

    return u
```

The remaining tasks of our initial program, such as calling the `solver` function with problem-specific parameters and plotting, can be placed in

a separate function. Here we choose to put this code in a function named `run_solver`:

```
def run_solver():
    "Run solver to compute and post-process solution"

    # Set up problem parameters and call solver
    u_D = Expression('1 + x[0]*x[0] + 2*x[1]*x[1]', degree=2)
    f = Constant(-6.0)
    u = solver(f, u_D, 8, 8, 1)

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

    # Save solution to file in VTK format
    vtkfile = File('poisson_solver/solution.pvd')
    vtkfile << u
```

The solution can now be computed, plotted, and saved to file by simply calling the `run_solver` function.

5.1.2 Writing the solver as a Python module

The refactored code is placed in a file `ft12_poisson_solver.py`. We should make sure that such a file can be imported (and hence reused) in other programs. This means that all statements in the main program that are not inside functions should appear within a test `if __name__ == '__main__':`. This test is true if the file is executed as a program, but false if the file is imported. If we want to run this file in the same way as we can run `ft01_poisson.py`, the main program is simply a call to `run_solver` followed by a call to `interactive` to hold the plot:

```
if __name__ == '__main__':
    run_solver()
    interactive()
```

This complete program can be found in the file `ft12_poisson_solver.py`.

5.1.3 Verification and unit tests

The remaining part of our first program is to compare the numerical and the exact solutions. Every time we edit the code we must rerun the test and examine that `error_max` is sufficiently small so we know that the code still works. To this end, we shall adopt *unit testing*, meaning that we create a mathematical test and corresponding software that can run all our tests

automatically and check that all tests pass. Python has several tools for unit testing. Two very popular ones are pytest and nose. These are almost identical and very easy to use. More classical unit testing with test classes is offered by the built-in module `unittest`, but here we are going to use pytest (or nose) since that will result in shorter and clearer code.

Mathematically, our unit test is that the finite element solution of our problem when $f = -6$ equals the exact solution $u = u_{\mathrm{D}} = 1 + x^2 + 2y^2$ at the vertices of the mesh. We have already created a code that finds the error at the vertices for our numerical solution. Because of rounding errors, we cannot demand this error to be zero, but we have to use a tolerance, which depends on the number of elements and the degrees of the polynomials in the finite element basis. If we want to test that the `solver` function works for meshes up to $2 \times (20 \times 20)$ elements and cubic Lagrange elements, 10^{-10} is an appropriate tolerance for testing that the maximum error vanishes.

To make our test case work together with pytest and nose, we have to make a couple of small adjustments to our program. The simple rule is that each test must be placed in a function that

- has a name starting with `test_`,
- has no arguments, and
- implements a test expressed as `assert success, msg`.

Regarding the last point, `success` is a boolean expression that is `False` if the test fails, and in that case the string `msg` is written to the screen. When the test fails, `assert` raises an `AssertionError` exception in Python, and otherwise runs silently. The `msg` string is optional, so `assert success` is the minimal test. In our case, we will write `assert error_max < tol`, where `tol` is the tolerance mentioned above.

A proper *test function* for implementing this unit test in the pytest or nose testing frameworks has the following form. Note that we perform the test for different mesh resolutions and degrees of finite elements.

```
def test_solver():
    "Test solver by reproducing u = 1 + x^2 + 2y^2"

    # Set up parameters for testing
    tol = 1E-10
    u_D = Expression('1 + x[0]*x[0] + 2*x[1]*x[1]', degree=2)
    f = Constant(-6.0)

    # Iterate over mesh sizes and degrees
    for Nx, Ny in [(3, 3), (3, 5), (5, 3), (20, 20)]:
        for degree in 1, 2, 3:
            print('Solving on a 2 x (%d x %d) mesh with P%d elements.'
                  % (Nx, Ny, degree))

            # Compute solution
            u = solver(f, u_D, Nx, Ny, degree)
```

```
            # Extract the mesh
            mesh = u.function_space().mesh()

            # Compute maximum error at vertices
            vertex_values_u_D = u_D.compute_vertex_values(mesh)
            vertex_values_u  = u.compute_vertex_values(mesh)
            error_max = np.max(np.abs(vertex_values_u_D - \
                                      vertex_values_u))

            # Check maximum error
            msg = 'error_max = %g' % error_max
            assert error_max < tol, msg
```

To run the test, we type the following command:

Terminal

```
Terminal> py.test ft12_poisson_solver.py
```

This will run all functions named `test_*` (currently only the `test_solver` function) found in the file and report the results. For more verbose output, add the flags `-s -v`.

We shall make it a habit to encapsulate numerical test problems in unit tests as above, and we strongly encourage the reader to create similar unit tests whenever a FEniCS solver is implemented.

Tip: Print messages in test functions

The `assert` statement runs silently when the test passes so users may become uncertain if all the statements in a test function are really executed. A psychological help is to print out something before `assert` (as we do in the example above) such that it is clear that the test really takes place. Note that `py.test` needs the `-s` option to show printout from the test functions.

Tip: Debugging with iPython

One can easily enter iPython from a Python script by adding the following line anywhere in the code:

```
from IPython import embed; embed()
```

This line starts an interactive Python session which lets you print and plot variables, which can be very helpful for debugging.

5.1.4 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. As an appetizer, go back to the previous programs `ft01_poisson.py` or `ft12_poisson_solver.py` and change the mesh construction from `UnitSquareMesh(8, 8)` to `UnitCubeMesh(8, 8, 8)`. Now the domain is the unit cube partitioned into $8 \times 8 \times 8$ boxes, and each box is divided into six tetrahedron-shaped finite elements for computations. Run the program and observe that we can solve a 3D problem without any other modifications! (In 1D, expressions must be modified to not depend on `x[1]`.) The visualization allows you to rotate the cube and observe the function values as colors on the boundary.

If we want to parameterize the creation of unit interval, unit square, or unit cube over dimension, we can do so by encapsulating this part of the code in a function. Given a list or tuple specifying the division into cells in the spatial coordinates, the following function returns the mesh for a d-dimensional cube:

```
def UnitHyperCube(divisions):
    mesh_classes = [UnitIntervalMesh, UnitSquareMesh, UnitCubeMesh]
    d = len(divisions)
    mesh = mesh_classes[d - 1](*divisions)
    return mesh
```

The construction `mesh_class[d - 1]` will pick the right name of the object used to define the domain and generate the mesh. Moreover, the argument `*divisions` sends all the components of the list `divisions` as separate arguments to the constructor of the mesh construction class picked out by `mesh_class[d - 1]`. For example, in a 2D problem where `divisions` has two elements, the statement

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

is equivalent to

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

The `solver` function from `ft12_poisson_solver.py` may be modified to solve d-dimensional problems by replacing the `Nx` and `Ny` parameters by `divisions`, and calling the function `UnitHyperCube` to create the mesh. Note that `UnitHyperCube` is a *function* and not a *class*, but we have named it using so-called *CamelCase notation* to make it look like a class:

```
mesh = UnitHyperCube(divisions)
```

5.2 Working with linear solvers

Sparse LU decomposition (Gaussian elimination) is used by default to solve linear systems of equations in FEniCS programs. This is a very robust and simple method. It is the recommended method for systems with up to a few thousand unknowns and may hence be the method of choice for many 2D and smaller 3D problems. However, sparse LU decomposition becomes slow and one quickly runs out of memory for larger problems. For large problems, we instead need to use *iterative methods* which are faster and require much less memory. We will now look at how to take advantage of state-of-the-art iterative solution methods in FEniCS.

5.2.1 Choosing a linear solver and preconditioner

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 system matrix, for which the optimal Krylov solver is the Conjugate Gradient (CG) method. For non-symmetric problems, a Krylov solver for non-symmetric systems, such as GMRES, is a better choice. Incomplete LU factorization (ILU) is a popular and robust all-round preconditioner, so let us try the GMRES-ILU pair:

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

Section 5.2.6 lists the most popular choices of Krylov solvers and preconditioners available in FEniCS.

5.2.2 Choosing a linear algebra backend

The actual GMRES and ILU implementations that are brought into action depend 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 FEniCS is compiled with PETSc. If PETSc is not available, then FEniCS falls back to using the Eigen backend. The linear algebra backend in FEniCS can be set using the following command:

```
parameters.linear_algebra_backend = backendname
```

where `backendname` is a string. To see which linear algebra backends are available, you can call the FEniCS function `list_linear_algebra_backends`. Similarly, one may check which linear algebra backend is currently being used by the following command:

```
print(parameters.linear_algebra_backend)
```

5.2.3 Setting solver parameters

We will normally want to control the tolerance in the stopping criterion and the maximum number of iterations when running an iterative method. Such parameters can be controlled at both a *global* and a *local* level. We will start by looking at how to set global parameters. For more advanced programs, one may want to use a number of different linear solvers and set different tolerances and other parameters. Then it becomes important to control the parameters at a *local* level. We will return to this issue in Section 5.3.1.

Changing a parameter in the global FEniCS parameter database affects all linear solvers (created *after* the parameter has been set). The global FEniCS parameter database is simply called `parameters` and it behaves as a nested dictionary. Write

```
info(parameters, verbose=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:

```
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 some norm of the residual, which must be smaller than the absolute tolerance parameter *or* smaller than the relative tolerance parameter times the initial residual.

We remark that default values for the global parameter database can be defined in an XML file. To generate such a file from the current set of parameters in a program, run

```
File('parameters.xml') << parameters
```

If a `dolfin_parameters.xml` 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. Another alternative is to load the XML file (with any name) manually in the program:

```
File('parameters.xml') >> parameters
```

The XML file can also be in gzip'ed form with the extension `.xml.gz`.

5.2.4 An extended solver function

We may extend the previous solver function from `ft12_poisson_solver.py` in Section 5.1.1 such that it also offers the GMRES+ILU preconditioned Krylov solver:

This new `solver` function, found in the file `ft10_poisson_extended.py`, replaces the one in `ft12_poisson_solver.py`. It has all the functionality of the previous `solver` function, but can also solve the linear system with iterative methods.

5.2.5 A remark regarding unit tests

Regarding verification of the new `solver` function in terms of unit tests, it turns out that unit testing for a problem where the approximation error vanishes gets more complicated when we use iterative methods. The problem is to keep the error due to iterative solution smaller than the tolerance used in the verification tests. First of all, this means that the tolerances used in the Krylov solvers must be smaller than the tolerance used in the `assert` test, but this is no guarantee to keep the linear solver error this small. For linear elements and small meshes, a tolerance of 10^{-11} works well in the case of Krylov solvers too (using a tolerance 10^{-12} in those solvers). The interested reader is referred to the `demo_solvers` function in `ft10_poisson_extended.py` for details: this function tests the numerical solution for direct and iterative linear solvers, for different meshes, and different degrees of the polynomials in the finite element basis functions.

5.2.6 List of linear solver methods and preconditioners

Which linear solvers and preconditioners that are available in FEniCS depends on how FEniCS has been configured and which linear algebra backend is currently active. The following table shows an example of which linear solvers that can be available through FEniCS when the PETSc backend is active:

Name	Method
`'bicgstab'`	Biconjugate gradient stabilized method
`'cg'`	Conjugate gradient method
`'gmres'`	Generalized minimal residual method
`'minres'`	Minimal residual method
`'petsc'`	PETSc built in LU solver
`'richardson'`	Richardson method
`'superlu_dist'`	Parallel SuperLU
`'tfqmr'`	Transpose-free quasi-minimal residual method
`'umfpack'`	UMFPACK

The set of available preconditioners also depends on configuration and linear algebra backend. The following table shows an example of which preconditioners may be available:

Name	Method
`'icc'`	Incomplete Cholesky factorization
`'ilu'`	Incomplete LU factorization
`'petsc_amg'`	PETSc algebraic multigrid
`'sor'`	Successive over-relaxation

An up-to-date list of the available solvers and preconditioners for your FEniCS installation can be produced by

```
list_linear_solver_methods()
list_krylov_solver_preconditioners()
```

5.3 High-level and low-level solver interfaces

The FEniCS interface allows different ways to access the core functionality, ranging from very high-level to low-level access. So far, we have mostly used the high-level call `solve(a == L, u, bc)` to solve a variational problem `a == L` with a certain boundary condition `bc`. However, sometimes you may need more fine-grained control of the solution process. In particular, the call to `solve` will create certain objects that are thrown away after the solution has been computed, and it may be practical or efficient to *reuse* those objects.

5.3.1 Linear variational problem and solver objects

In this section, we will look at an alternative interface to solving linear variational problems in FEniCS, which may be preferable in many situations com-

pared to the high-level `solve` function interface. This interface uses the two classes `LinearVariationalProblem` and `LinearVariationalSolver`. Using this interface, the equivalent of `solve(a == L, u, bc)` looks as follows:

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

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

```
solver.parameters.linear_solver = 'gmres'
solver.parameters.preconditioner = 'ilu'
prm = solver.parameters.krylov_solver  # short form
prm.absolute_tolerance = 1E-7
prm.relative_tolerance = 1E-4
prm.maximum_iterations = 1000
```

Settings in the global `parameters` database are propagated to parameter sets in individual objects, with the possibility of being overwritten as above. Note that global parameter values can only affect local parameter values if set before the time of creation of the local object. Thus, changing the value of the tolerance in the global parameter database will not affect the parameters for already created solvers.

5.3.2 Explicit assembly and solve

As we saw already in Section 3.4, linear variational problems can be assembled explicitly in FEniCS into matrices and vectors using the `assemble` function. This allows even more fine-grained control of the solution process compared to using the high-level `solve` function or using the classes `LinearVariationalProblem` and `LinearVariationalSolver`. We will now look more closely into how to use the `assemble` function and how to combine this with low-level calls for solving the assembled linear systems.

Given a variational problem $a(u,v) = L(v)$, 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

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

which is nothing but a linear system,

$$AU = b,$$

where the entries of A and b are given by

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

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

```
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 the same as before; that is, `a` refers to the bilinear form involving a `TrialFunction` object `u` and a `TestFunction` object `v`, and `L` involves the same `TestFunction` object `v`. From `a` and `L`, the `assemble` function can compute A and b.

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.

The matrix A and vector b are first assembled without incorporating essential (Dirichlet) boundary conditions. Thereafter, the call `bc.apply(A, b)` performs the necessary modifications of the linear system such that `u` is guaranteed to equal the prescribed boundary values. When we have multiple Dirichlet conditions stored in a list `bcs`, we must apply each condition in `bcs` to the system:

```
for bc in bcs:
    bc.apply(A, b)

# Alternative syntax using list comprehension
[bc.apply(A, b) for bc in bcs]
```

Alternatively, we can use the function `assemble_system`, which takes the boundary conditions into account when assembling the linear system. This method preserves the symmetry of the linear system for a symmetric bilinear form. Even if the matrix `A` that comes out of the call to `assemble` is symmetric (for a symmetric bilinear form `a`), the call to `bc.apply` will break the symmetry. Preserving the symmetry of a variational problem is important

when using particular linear solvers designed for symmetric systems, such as the conjugate gradient method.

Once the linear system has been assembled, we need to compute the solution $U = A^{-1}b$ and store the solution U in the vector `U = u.vector()`. In the same way as linear variational problems can be programmed using different interfaces in FEniCS—the high-level `solve` function, the class `LinearVariationalSolver`, and the low-level `assemble` function—linear systems can also be programmed using different interfaces in FEniCS. The high-level interface to solving a linear system in FEniCS is also named `solve`:

```
solve(A, U, b)
```

By default, `solve(A, U, b)` uses sparse LU decomposition to compute the solution. Specification of an iterative solver and preconditioner can be made through two optional arguments:

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

Appropriate names of solvers and preconditioners are found in Section 5.2.6.

This high-level interface is useful for many applications, but sometimes more fine-grained control is needed. One can then create one or more `KrylovSolver` objects that are then used to solve linear systems. Each different solver object can have its own set of parameters and selection of iterative method and preconditioner. Here is an example:

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

The function `solver_linalg` in the program file `ft10_poisson_extended.py` implements such a solver.

The choice of start vector for the iterations in a linear solver is often important. By default, the values of `u` and thus the vector `U = u.vector()` will be initialized to zero. If we instead wanted to initialize `U` with random numbers in the interval $[-100, 100]$ this can be done as follows:

```
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 both turn off the default behavior of setting the start vector ("initial guess") to zero, and also set the values of the vector `U` to nonzero values. This is useful if we happen to know a good initial guess for the solution.

Using a nonzero initial guess can be particularly important for time-dependent problems or when solving a linear system as part of a nonlinear iteration, since then the previous solution vector `U` will often be a good initial guess for the solution in the next time step or iteration. In this case, the values in the vector `U` will naturally be initialized with the previous solution vector (if we just used it to solve a linear system), so the only extra step necessary is to set the parameter `nonzero_initial_guess` to `True`.

5.3.3 Examining matrix and vector values

When calling `A = assemble(a)` and `b = assemble(L)`, the object `A` will be of type `Matrix`, while `b` and `u.vector()` are of type `Vector`. To examine the values, we may convert the matrix and vector data to `numpy` arrays by calling the `array` method as shown before. For example, if you wonder how essential boundary conditions are incorporated into linear systems, you can print out `A` and `b` before and after the `bc.apply(A, b)` call:

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

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

Writing `load Ab.mat` in MATLAB or Octave will then make the array 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. FEniCS also has an interface to the eigensolver package SLEPc, which is the preferred tool for computing the eigenvalues of large, sparse matrices of the type encountered in PDE problems (see `demo/documented/eigenvalue/python/` in the FEniCS/DOLFIN source code tree for a demo).

5.4 Degrees of freedom and function evaluation

5.4.1 Examining the degrees of freedom

We have seen before how to grab the degrees of freedom array from a finite element function `u`:

```
nodal_values = u.vector().array()
```

For a finite element function from a standard continuous piecewise linear function space (P_1 Lagrange elements), these values will be the same as the values we get by the following statement:

```
vertex_values = u.compute_vertex_values(mesh)
```

Both `nodal_values` and `vertex_values` will be `numpy` arrays and they will be of the same length and contain the same values (for P_1 elements), but with possibly different ordering. The array `vertex_values` will have the same ordering as the vertices of the mesh, while `nodal_values` will be ordered in a way that (nearly) minimizes the bandwidth of the system matrix and thus improves the efficiency of linear solvers.

A fundamental question is: what are the coordinates of the vertex whose value is `nodal_values[i]`? To answer this question, we need to understand how to get our hands on the coordinates, and in particular, the numbering of degrees of freedom and the numbering of vertices in the mesh.

The function `mesh.coordinates` returns the coordinates of the vertices as a `numpy` array with shape (M,d), M being the number of vertices in the mesh and d being the number of space dimensions:

```
>>> from fenics import *
>>> mesh = UnitSquareMesh(2, 2)
>>> coordinates = mesh.coordinates()
>>> coordinates
array([[ 0. ,  0. ],
       [ 0.5,  0. ],
       [ 1. ,  0. ],
       [ 0. ,  0.5],
       [ 0.5,  0.5],
       [ 1. ,  0.5],
       [ 0. ,  1. ],
       [ 0.5,  1. ],
       [ 1. ,  1. ]])
```

We see from this output that for this particular mesh, the vertices are first numbered along $y = 0$ with increasing x coordinate, then along $y = 0.5$, and so on.

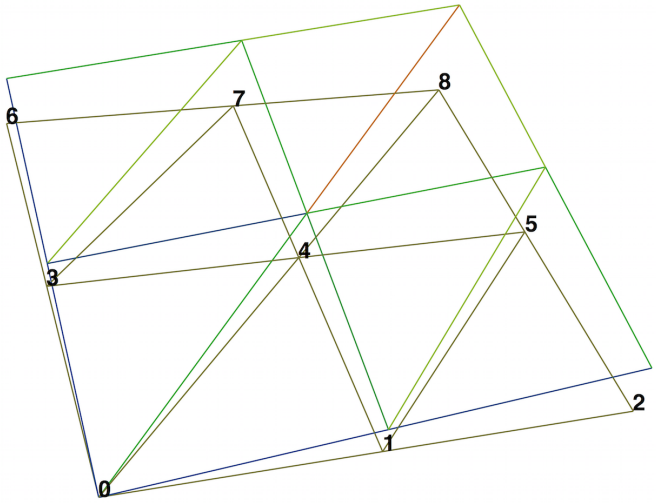
Next we compute a function `u` on this mesh. Let's take $u = x + y$:

```
>>> V = FunctionSpace(mesh, 'P', 1)
>>> u = interpolate(Expression('x[0] + x[1]', degree=1), V)
```

```
>>> plot(u, interactive=True)
>>> nodal_values = u.vector().array()
>>> nodal_values
array([ 1. ,  0.5,  1.5,  0. ,  1. ,  2. ,  0.5,  1.5,  1. ])
```

We observe that `nodal_values[0]` is *not* the value of $x+y$ at vertex number 0, since this vertex has coordinates $x = y = 0$. The numbering of the nodal values (degrees of freedom) $U_1, \ldots, U_N$ is obviously not the same as the numbering of the vertices.

The vertex numbering may be examined by using the FEniCS `plot` command. To do this, plot the function `u`, press `w` to turn on wireframe instead of a fully colored surface, `m` to show the mesh, and then `v` to show the numbering of the vertices.



Let's instead examine the values by calling `u.compute_vertex_values`:

```
>>> vertex_values = u.compute_vertex_values()
>>> for i, x in enumerate(coordinates):
...     print('vertex %d: vertex_values[%d] = %g\tu(%s) = %g' %
...           (i, i, vertex_values[i], x, u(x)))
vertex 0: vertex_values[0] = 0          u([ 0.  0.]) = 8.46545e-16
vertex 1: vertex_values[1] = 0.5        u([ 0.5  0. ]) = 0.5
vertex 2: vertex_values[2] = 1          u([ 1.  0.]) = 1
vertex 3: vertex_values[3] = 0.5        u([ 0.   0.5]) = 0.5
vertex 4: vertex_values[4] = 1          u([ 0.5  0.5]) = 1
vertex 5: vertex_values[5] = 1.5        u([ 1.   0.5]) = 1.5
vertex 6: vertex_values[6] = 1          u([ 0.  1.]) = 1
vertex 7: vertex_values[7] = 1.5        u([ 0.5  1. ]) = 1.5
vertex 8: vertex_values[8] = 2          u([ 1.  1.]) = 2
```

We can ask FEniCS to give us the mapping from vertices to degrees of freedom for a certain function space V:

```
v2d = vertex_to_dof_map(V)
```

Now, `nodal_values[v2d[i]]` will give us the value of the degree of freedom corresponding to vertex `i` (`v2d[i]`). In particular, `nodal_values[v2d]` is an array with all the elements in the same (vertex numbered) order as `coordinates`. The inverse map, from degrees of freedom number to vertex number is given by `dof_to_vertex_map(V)`. This means that we may call `coordinates[dof_to_vertex_map(V)]` to get an array of all the coordinates in the same order as the degrees of freedom. Note that these mappings are only available in FEniCS for P_1 elements.

For Lagrange elements of degree larger than 1, there are degrees of freedom (nodes) that do not correspond to vertices. For these elements, we may get the vertex values by calling `u.compute_vertex_values(mesh)`, and we can get the degrees of freedom by the call `u.vector().array()`. To get the coordinates associated with all degrees of freedom, we need to iterate over the elements of the mesh and ask FEniCS to return the coordinates and dofs associated with each element (cell). This information is stored in the `FiniteElement` and `DofMap` object of a `FunctionSpace`. The following code illustrates how to iterate over all elements of a mesh and print the coordinates and degrees of freedom associated with the element.

```
element = V.element()
dofmap = V.dofmap()
for cell in cells(mesh):
    print(element.tabulate_dof_coordinates(cell))
    print(dofmap.cell_dofs(cell.index()))
```

5.4.2 Setting the degrees of freedom

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 $\max_j |U_j| = 1$. Then we must divide all U_j values by $\max_j |U_j|$. The following function performs the task:

```
def normalize_solution(u):
    "Normalize u: return u divided by max(u)"
    u_array = u.vector().array()
    u_max = np.max(np.abs(u_array))
    u_array /= u_max
    u.vector()[:] = u_array
    #u.vector().set_local(u_array)  # alternative
    return u
```

When using Lagrange elements, this (approximately) ensures that the maximum value of the function u is 1.

The `/=` operator implies an in-place modification of the object on the left-hand side: all elements of the array `nodal_values` are divided by the value `u_max`. Alternatively, we could do `nodal_values = nodal_values / u_max`, which implies creating a new array on the right-hand side and assigning this array to the name `nodal_values`.

Be careful when manipulating degrees of freedom

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 (i.e., a copy), manipulate it, and insert it back with `u.vector()[:] =` or use `u.set_local(...)`.

5.4.3 Function evaluation

A FEniCS `Function` object is uniquely defined in the interior of each cell of the finite element mesh. For continuous (Lagrange) function spaces, the function values are also uniquely defined on cell boundaries. A `Function` object `u` can be evaluated by simply calling

```
u(x)
```

where `x` is either a `Point` or a Python tuple of the correct space dimension. When a `Function` is evaluated, FEniCS must first find which cell of the mesh that contains the given point (if any), and then evaluate a linear combination of basis functions at the given point inside the cell in question. FEniCS uses efficient data structures (bounding box trees) to quickly find the point, but building the tree is a relatively expensive operation so the cost of evaluating a `Function` at a single point is costly. Repeated evaluation will reuse the computed data structures and thus be relatively less expensive.

Cheap vs expensive function evaluation

A `Function` object `u` can be evaluated in various ways:

1. `u(x)` for an arbitrary point `x`
2. `u.vector().array()[i]` for degree of freedom number `i`
3. `u.compute_vertex_values()[i]` at vertex number `i`

The first method, though very flexible, is in general expensive while the other two are very efficient (but limited to certain points).

To demonstrate the use of point evaluation of `Function` objects, we print the value of the computed finite element solution `u` for the Poisson problem at the center point of the domain and compare it with the exact solution:

```
center = (0.5, 0.5)
error = u_D(center) - u(center)
print('Error at %s: %g' % (center, error))
```

For a $2 \times (3 \times 3)$ mesh, the output from the previous snippet becomes

```
Error at (0.5, 0.5): -0.0833333
```

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 `u_D` is a quadratic function. When the center point is a node, as in a $2 \times (2 \times 2)$ or $2 \times (4 \times 4)$ mesh, the error is of the order 10^{-15}.

5.5 Postprocessing computations

As the final theme in this chapter, we will look at how to *postprocess computations*; that is, how to compute various derived quantities from the computed solution of a PDE. The solution u itself may be of interest for visualizing general features of the solution, but sometimes one is interested in computing the solution of a PDE to compute a specific quantity that derives from the solution, such as, e.g., the flux, a point-value, or some average of the solution.

5.5.1 Test problem

As a test problem, we consider again the variable-coefficient Poisson problem with a single Dirichlet boundary condition:

$$-\nabla \cdot (\kappa \nabla u) = f \quad \text{in } \Omega, \tag{5.1}$$

$$u = u_{\mathrm{D}} \quad \text{on } \partial\Omega\,. \tag{5.2}$$

Let us continue to use our favorite solution $u(x,y) = 1 + x^2 + 2y^2$ and then prescribe $\kappa(x,y) = x + y$. It follows that $u_{\mathrm{D}}(x,y) = 1 + x^2 + 2y^2$ and $f(x,y) = -8x - 10y$.

As before, the variational formulation for this model problem can be specified in FEniCS as

```
a = kappa*dot(grad(u), grad(v))*dx
L = f*v*dx
```

with the coefficient κ and right-hand side f given by

```
kappa = Expression('x[0] + x[1]', degree=1)
f = Expression('-8*x[0] - 10*x[1]', degree=1)
```

5.5.2 Flux computations

It is often of interest to compute the flux $Q = -\kappa\nabla u$. Since $u = \sum_{j=1}^{N} U_j\phi_j$, it follows that

$$Q = -\kappa \sum_{j=1}^{N} U_j \nabla\phi_j \,.$$

We note that the gradient of a piecewise continuous finite element scalar field is a discontinuous vector field since the basis functions $\{\phi_j\}$ have discontinuous derivatives at the boundaries of the cells. For example, using Lagrange elements of degree 1, u is linear over each cell, and the gradient 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, using the same elements for the components of w as we used for u. This process is known as *projection*.

Projection is a common operation in finite element analysis and, as we have already seen, FEniCS has a function for easily performing the projection: `project(expression, W)`, which returns the projection of some expression into the space `W`.

In our case, the flux $Q = -\kappa\nabla u$ is vector-valued and we need to pick `W` as the vector-valued function space of the same degree as the space `V` where `u` resides:

```
V = u.function_space()
mesh = V.mesh()
degree = V.ufl_element().degree()
W = VectorFunctionSpace(mesh, 'P', degree)

grad_u = project(grad(u), W)
flux_u = project(-k*grad(u), W)
```

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.

Plotting the flux vector field is naturally as easy as plotting anything else:

```
plot(flux_u, title='flux field')

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

For data analysis of the nodal values of the flux field, we can grab the underlying `numpy` arrays (which demands a `deepcopy=True` in the split of `flux`):

```
flux_x_nodal_values = flux_x.vector().dofs()
flux_y_nodal_values = flux_y.vector().dofs()
```

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

```
flux_u_nodal_values = flux_u.vector().array()
```

However, this is a flat `numpy` array containing the degrees of freedom for both the x and y components of the flux and the ordering of the components may be mixed up by FEniCS in order to improve computational efficiency.

The function `demo_flux` in the program `ft10_poisson_extended.py` demonstrates the computations described above.

Manual projection.

Although you will always use `project` to project a finite element function, it can be instructive to look at how to formulate the projection mathematically and implement its steps manually in FEniCS.

Let's say we have an expression $g = g(u)$ that we want to project into some space W. The mathematical formulation of the (L^2) projection $w = P_W g$ into W is the variational problem

$$\int_\Omega wv \,\mathrm{d}x = \int_\Omega gv \,\mathrm{d}x \tag{5.3}$$

for all test functions $v \in W$. In other words, we have a standard variational problem $a(w,v) = L(v)$ where now

$$a(w,v) = \int_\Omega wv \,\mathrm{d}x, \tag{5.4}$$

$$L(v) = \int_\Omega gv \,\mathrm{d}x\,. \tag{5.5}$$

Note that when the functions in W are vector-valued, as is the case when we project the gradient $g(u) = \nabla u$, we must replace the products above by $w \cdot v$ and $g \cdot v$.

The variational problem is easy to define in FEniCS.

```
w = TrialFunction(W)
v = TestFunction(W)

a = w*v*dx  # or dot(w, v)*dx when w is vector-valued
L = g*v*dx  # or dot(g, v)*dx when g is vector-valued
w = Function(W)
solve(a == L, w)
```

The boundary condition argument to `solve` is dropped since there are no essential boundary conditions in this problem.

5.5.3 Computing functionals

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

$$\frac{1}{2}||\nabla u||^2 = \frac{1}{2}\int_\Omega \nabla u \cdot \nabla u \,\mathrm{d}x, \tag{5.6}$$

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

$$||u_\mathrm{e} - u|| = \left(\int_\Omega (u_\mathrm{e} - u)^2 \,\mathrm{d}x\right)^{1/2}, \tag{5.7}$$

where u_e is the exact solution. The error is of particular interest when studying convergence properties of finite element methods. Other times, we may instead be interested in computing the flux out through a part Γ of the boundary $\partial\Omega$,

$$F = -\int_\Gamma \kappa \nabla u \cdot n \,\mathrm{d}s, \tag{5.8}$$

where n is the outward-pointing unit normal on Γ.

All these functionals are easy to compute with FEniCS, as we shall see in the examples below.

Energy functional. The integrand of the energy functional (5.6) is described in the UFL language in the same manner as we describe weak forms:

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

The functional `energy` is evaluated by calling the `assemble` function that we have previously used to assemble matrices and vectors. FEniCS will recognize that the form has "rank 0" (since it contains no trial and test functions) and return the result as a scalar value.

Error functional. The functional (5.7) can be computed as follows:

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

The exact solution u_e is here represented by a `Function` or `Expression` object `u_e`, while `u` is the finite element approximation (and thus a `Function`). Sometimes, for very small error values, the result of `assemble(error)` can be a (very small) negative number, so we have used `abs` in the expression for `E` above to ensure a positive value for the `sqrt` function.

As will be explained and demonstrated in Section 5.5.4, the integration of `(u_e - u)**2*dx` can result in too optimistic convergence rates unless one is careful how the difference `u_e - u` is evaluated. The general recommendation for reliable error computation is to use the `errornorm` function:

```
E = errornorm(u_e, u)
```

Flux Functional. To compute flux integrals like $F = -\int_\Gamma \kappa \nabla u \cdot n \, \mathrm{d}s$, we need to define the n vector, referred to as a *facet normal* in FEniCS. If the surface domain Γ in the flux integral is the complete boundary, we can perform the flux computation by

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

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

It is possible to restrict the integration to a part of the boundary by using a mesh function to mark the relevant part, as explained in Section 4.4. Assuming that the part corresponds to subdomain number `i`, the relevant syntax for the variational formulation of the flux is `-k*dot(grad(u), n)*ds(i)`.

A note on the accuracy of integration

As we have seen before, FEniCS `Expressions` must be defined using a particular `degree`. The degree tells FEniCS into which local finite element space the expression should be interpolated when performing local computations (integration). As an illustration, consider the com-

putation of the integral $\int_0^1 \cos x \, \mathrm{d}x = \sin 1$. This may be computed in FEniCS by

```
mesh = UnitIntervalMesh(1)
I = assemble(Expression('cos(x[0])', degree=degree)*dx(domain=mesh))
```

Note that we must here specify the argument `domain=mesh` to the measure `dx`. This is normally not necessary when defining forms in FEniCS but is necessary here since `cos(x[0])` is not associated with any domain (as is the case when we integrate a `Function` from some `FunctionSpace` defined on some `Mesh`).

Varying the degree between 0 and 5, the value of $|\sin(1) - I|$ is `0.036`, `0.071`, `0.00030`, `0.00013`, `4.5E-07`, and `2.5E-07`.

FEniCS also allows expressions to be expressed directly as part of a form. This requires the creation of a `SpatialCoordinate`. In this case, the accuracy is dictated by the accuracy of the integration, which may be controlled by a `degree` argument to the integration measure `dx`. The `degree` argument specifies that the integration should be exact for polynomials of that degree.

The following code snippet shows how to compute the integral $\int_0^1 \cos x \, \mathrm{d}x$ using this approach:

```
mesh = UnitIntervalMesh(1)
x = SpatialCoordinate(mesh)
I = assemble(cos(x[0])*dx(degree=degree))
```

Varying the degree between 0 and 5, the value of $|\sin(1) - I|$ is `0.036`, `0.036`, `0.00020`, `0.00020`, `4.3E-07`, `4.3E-07`. Note that the quadrature degrees are only available for odd degrees so that degree 0 will use the same quadrature rule as degree 1, degree 2 will give the same quadrature rule as degree 3 and so on.

5.5.4 Computing convergence rates

A central question for any numerical method is its *convergence rate*: how fast does the error approach zero when the resolution is increased? For finite element methods, this typically corresponds to proving, theoretically or empirically, that the error $e = u_\mathrm{e} - u$ is bounded by the mesh size h to some power r; that is, $\|e\| \leq Ch^r$ for some constant C. The number r is called the *convergence rate* of the method. Note that different norms, like the L^2-norm $\|e\|$ or H_0^1-norm $\|\nabla e\|$ typically have different convergence rates.

To illustrate how to compute errors and convergence rates in FEniCS, we have included the function `compute_convergence_rates` in the tutorial program `ft10_poisson_extended.py`. This is a tool that is very handy when verifying finite element codes and will therefore be explained in detail here.

Computing error norms. As we have already seen, the L^2-norm of the error $u_e - u$ can be implemented in FEniCS by

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

As above, we have used `abs` in the expression for `E` above to ensure a positive value for the `sqrt` function.

It is important to understand how FEniCS computes the error from the above code, since we may otherwise run into subtle issues when using the value for computing convergence rates. The first subtle issue is that if `u_e` is not already a finite element function (an object created using `Function(V)`), which is the case if `u_e` is defined as an `Expression`, FEniCS must interpolate `u_e` into some local finite element space on each element of the mesh. The degree used for the interpolation is determined by the mandatory keyword argument to the `Expression` class, for example:

```
u_e = Expression('sin(x[0])', degree=1)
```

This means that the error computed will not be equal to the actual error $\|u_e - u\|$ but rather the difference between the finite element solution u and the piecewise linear interpolant of u_e. This may yield a too optimistic (too small) value for the error. A better value may be achieved by interpolating the exact solution into a higher-order function space, which can be done by simply increasing the degree:

```
u_e = Expression('sin(x[0])', degree=3)
```

The second subtle issue is that when FEniCS evaluates the expression `(u_e - u)**2`, this will be expanded into `u_e**2 + u**2 - 2*u_e*u`. If the error is small (and the solution itself is of moderate size), this calculation will correspond to the subtraction of two positive numbers (`u_e**2 + u**2` ~ 1 and `2*u_e*u` ~ 1) yielding a small number. Such a computation is very prone to round-off errors, which may again lead to an unreliable value for the error. To make this situation worse, FEniCS may expand this computation into a large number of terms, in particular for higher order elements, making the computation very unstable.

To help with these issues, FEniCS provides the built-in function `errornorm` which computes the error norm in a more intelligent way. First, both `u_e` and `u` are interpolated into a higher-order function space. Then, the degrees of freedom of `u_e` and `u` are subtracted to produce a new function in the higher-order function space. Finally, FEniCS integrates the square of the difference function and then takes the square root to get the value of the error norm. Using the `errornorm` function is simple:

```
E = errornorm(u_e, u, normtype='L2')
```

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

```
def errornorm(u_e, u):
```

```
    V = u.function_space()
    mesh = V.mesh()
    degree = V.ufl_element().degree()
    W = FunctionSpace(mesh, 'P', degree + 3)
    u_e_W = interpolate(u_e, W)
    u_W = interpolate(u, W)
    e_W = Function(W)
    e_W.vector()[:] = u_e_W.vector().array() - u_W.vector().array()
    error = e_W**2*dx
    return sqrt(abs(assemble(error)))
```

Sometimes it is of interest to compute the error of the gradient field: $||\nabla(u_{\mathrm{e}} - u)||$, often referred to as the H_0^1 or H^1 seminorm of the error. This can either be expressed as above, replacing the expression for `error` by `error = dot(grad(e_W), grad(e_W))*dx`, or by calling `errornorm` in FEniCS:

```
E = errornorm(u_e, u, norm_type='H10')
```

Type `help(errornorm)` in Python for more information about available norm types.

The function `compute_errors` in `ft10_poisson_extended.py` illustrates the computation of various error norms in FEniCS.

Computing convergence rates. Let's examine how to compute convergence rates in FEniCS. The `solver` function in `ft10_poisson_extended.py` allows us to easily compute solutions for finer and finer meshes and enables us to study the convergence rate. Define the element size $h = 1/n$, where n is the number of cell divisions in the x and y directions (`n = 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_2 and so forth. Assuming $E_i = Ch_i^r$ for unknown constants C and r, we can compare two consecutive experiments, $E_{i-1} = Ch_{i-1}^r$ and $E_i = Ch_i^r$, and solve for r:

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

The r values should approach the expected convergence rate (typically the polynomial degree + 1 for the L^2-error) as i increases.

The procedure above can easily be turned into Python code. Here we run through a list of element degrees (P_1, P_2, and P_3), perform experiments over a series of refined meshes, and for each experiment report the six error types as returned by `compute_errors`.

Test problem. To demonstrate the computation of convergence rates, we pick an exact solution u_{e}, this time a little more interesting than for the test problem in Chapter 2:

$$u_{\mathrm{e}}(x, y) = \sin(\omega\pi x)\sin(\omega\pi y).$$

This choice implies $f(x,y) = 2\omega^2\pi^2 u(x,y)$. With ω restricted to an integer, it follows that the boundary value is given by $u_{\mathrm{D}} = 0$.

We need to define the appropriate boundary conditions, the exact solution, and the f function in the code:

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

bc = DirichletBC(V, Constant(0), boundary)

omega = 1.0
u_e = Expression('sin(omega*pi*x[0])*sin(omega*pi*x[1])',
                 degree=6, omega=omega)

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

Experiments. An implementation of the computation of the convergence rate can be found in the function `demo_convergence_rates` in the demo program `ft10_poisson_extended.py`. We achieve some interesting results. Using the infinity norm of the difference of the degrees of freedom, we obtain the following table:

element	$n = 8$	$n = 16$	$n = 32$	$n = 64$
P_1	1.99	2.00	2.00	2.00
P_2	3.99	4.00	4.00	4.01
P_3	3.95	3.99	3.99	3.92

An entry like 3.99 for $n = 32$ and P_3 means that we estimate the rate 3.99 by comparing two meshes, with resolutions $n = 32$ and $n = 16$, using P_3 elements. Note the superconvergence for P_2 at the nodes. The best estimates of the rates appear in the right-most column, since these rates are based on the finest resolutions and are hence deepest into the asymptotic regime (until we reach a level where round-off errors and inexact solution of the linear system starts to play a role).

The L^2-norm errors computed using the FEniCS `errornorm` function show the expected h^{d+1} rate for u:

element	$n = 8$	$n = 16$	$n = 32$	$n = 64$
P_1	1.97	1.99	2.00	2.00
P_2	3.00	3.00	3.00	3.00
P_3	4.04	4.02	4.01	4.00

However, using `(u_e - u)**2` for the error computation, with the same degree for the interpolation of `u_e` as for `u`, gives strange results:

element	$n=8$	$n=16$	$n=32$	$n=64$
P_1	1.97	1.99	2.00	2.00
P_2	3.00	3.00	3.00	3.01
P_3	4.04	4.07	1.91	0.00

This is an example where it is important to interpolate `u_e` to a higher-order space (polynomials of degree 3 are sufficient here). This is handled automatically by using the `errornorm` function.

Checking convergence rates is an excellent method for verifying PDE codes.

5.5.5 Taking advantage of structured mesh data

Many readers have extensive experience with visualization and data analysis of 1D, 2D, and 3D scalar and vector fields on *uniform, structured meshes*, while FEniCS solvers exclusively work with *unstructured* meshes. Since it can many times be practical to work with structured data, we discuss in this section how to extract structured data for finite element solutions computed with FEniCS.

A necessary first step is to transform our `Mesh` object to an object representing a rectangle (or a 3D box) with equally-shaped *rectangular* cells. 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 mesh. 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, so using software tools to ease the work is advantageous.

In the directory of example programs included with this book, we have included the Python module `boxfield` which provides utilities for working with structured mesh data in FEniCS. Given a finite element function `u`, the following function returns a `BoxField` object that represents `u` on a structured mesh:

```
from boxfield import *
u_box = FEniCSBoxField(u, (nx, ny))
```

The `u_box` object contains several useful data structures:

- `u_box.grid`: object for the structured mesh
- `u_box.grid.coor[X]`: grid coordinates in `X=0` direction
- `u_box.grid.coor[Y]`: grid coordinates in `Y=1` direction
- `u_box.grid.coor[Z]`: grid coordinates in `Z=2` direction
- `u_box.grid.coorv[X]`: vectorized version of `u_box.grid.coor[X]`
- `u_box.grid.coorv[Y]`: vectorized version of `u_box.grid.coor[Y]`

- `u_box.grid.coorv[Z]`: vectorized version of `u_box.grid.coor[Z]`
- `u_box.values`: `numpy` array holding the `u` values; `u_box.values[i,j]` holds `u` at the mesh point with coordinates `(u_box.grid.coor[X][i], u_box.grid.coor[Y][j])`

Iterating over points and values. Let us now use the `solver` function from the `ft10_poisson_extended.py` code to compute `u`, map it onto a `BoxField` object for a structured mesh representation, and print the coordinates and function values at all mesh points:

```
u = solver(p, f, u_b, nx, ny, 1, linear_solver='direct')
u_box = structured_mesh(u, (nx, ny))
u_ = u_box.values

# Iterate over 2D mesh points (i, j)
for j in range(u_.shape[1]):
    for i in range(u_.shape[0]):
        print('u[%d, %d] = u(%g, %g) = %g' %
              (i, j,
               u_box.grid.coor[X][i], u_box.grid.coor[Y][j],
               u_[i, j]))
```

Computing finite difference approximations. Using the multidimensional array `u_ = u_box.values`, we can easily express finite difference approximations of derivatives:

```
x = u_box.grid.coor[X]
dx = x[1] - x[0]
u_xx = (u_[i - 1, j] - 2*u_[i, j] + u_[i + 1, j]) / dx**2
```

Making surface plots. The ability to access a finite element field as structured data is handy in many occasions, e.g., for visualization and data analysis. Using Matplotlib, we can create a surface plot, as shown in Figure 5.1 (upper left):

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D  # necessary for 3D plotting
from matplotlib import cm
fig = plt.figure()
ax = fig.gca(projection='3d')
cv = u_box.grid.coorv  # vectorized mesh coordinates
ax.plot_surface(cv[X], cv[Y], u_, cmap=cm.coolwarm,
                rstride=1, cstride=1)
plt.title('Surface plot of solution')
```

The key issue is to know that the coordinates needed for the surface plot is in `u_box.grid.coorv` and that the values are in `u_`.

Making contour plots. A contour plot can also be made by Matplotlib:

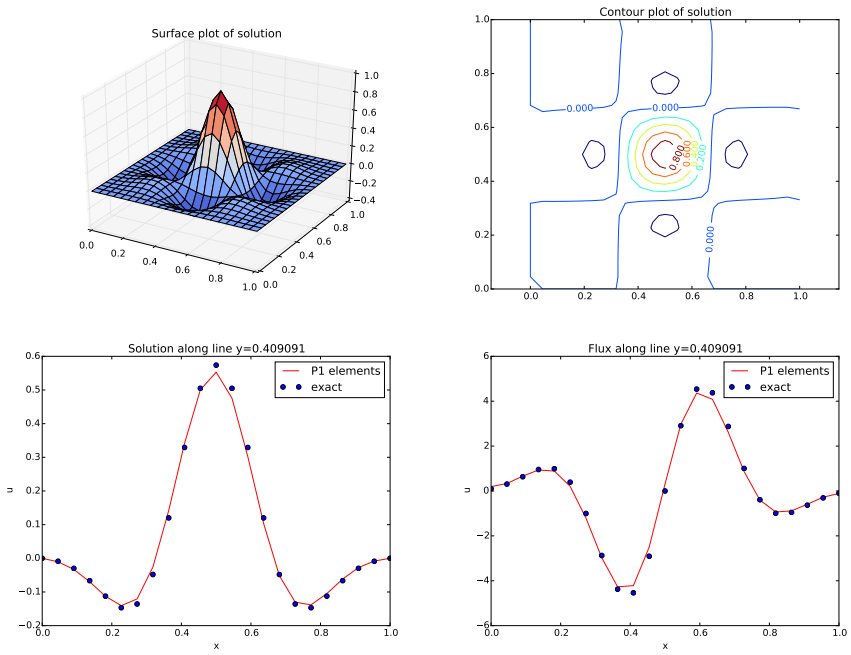


Fig. 5.1 Various plots of the solution on a structured mesh.

```
fig = plt.figure()
ax = fig.gca()
levels = [1.5, 2.0, 2.5, 3.5]
cs = ax.contour(cv[X], cv[Y], u_, levels=levels)
plt.clabel(cs)  # add labels to contour lines
plt.axis('equal')
plt.title('Contour plot of solution')
```

The result appears in Figure 5.1 (upper right).

Making curve plots through the domain. A handy feature of `BoxField` objects is the ability to give a starting point in the domain and a direction, and then extract the field and corresponding coordinates along the nearest line of *mesh points*. We have already seen how to interpolate the solution along a line in the mesh, but with `BoxField` you can pick out the computational points (vertices) for examination of these points. Numerical methods often show improved behavior at such points so this is of interest. For 3D fields one can also extract data in a plane.

Say we want to plot u along the line $y = 0.4$. The mesh points, `x`, and the u values along this line, `u_val`, can be extracted by

```
start = (0, 0.4)
x, u_val, y_fixed, snapped = u_box.gridline(start, direction=X)
```

The variable `snapped` is true if the line is snapped onto to nearest gridline and in that case `y_fixed` holds the snapped (altered) y value. The keyword argument `snap` is by default `True` to avoid interpolation and force snapping.

A comparison of the numerical and exact solution along the line $y \approx 0.41$ (snapped from $y = 0.4$) is made by the following code:

```
# Plot u along a line y = const and compare with exact solution
start = (0, 0.4)
x, u_val, y_fixed, snapped = u_box.gridline(start, direction=X)
u_e_val = [u_D((x_, y_fixed)) for x_ in x]
plt.figure()
plt.plot(x, u_val, 'r-')
plt.plot(x, u_e_val, 'bo')
plt.legend(['P1 elements', 'exact'], loc='best')
plt.title('Solution along line y=%g' % y_fixed)
plt.xlabel('x');  plt.ylabel('u')
```

See Figure 5.1 (lower left) for the resulting curve plot.

Making curve plots of the flux. Let us also compare the numerical and exact fluxes $-\kappa \partial u/\partial x$ along the same line as above:

```
# Plot the numerical and exact flux along the same line
flux_u = flux(u, kappa)
flux_u_x, flux_u_y = flux_u.split(deepcopy=True)
flux2_x = flux_u_x if flux_u_x.ufl_element().degree() == 1 \
          else interpolate(flux_x,
               FunctionSpace(u.function_space().mesh(), 'P', 1))
flux_u_x_box = FEniCSBoxField(flux_u_x, (nx,ny))
x, flux_u_val, y_fixed, snapped = \
   flux_u_x_box.gridline(start, direction=X)
y = y_fixed
plt.figure()
plt.plot(x, flux_u_val, 'r-')
plt.plot(x, flux_u_x_exact(x, y_fixed), 'bo')
plt.legend(['P1 elements', 'exact'], loc='best')
plt.title('Flux along line y=%g' % y_fixed)
plt.xlabel('x');  plt.ylabel('u')
```

The function `flux` called at the beginning of the code snippet is defined in the example program `ft10_poisson_extended.py` and interpolates the flux back into the function space.

Note that Matplotlib is one choice of plotting package. With the unified interface in the SciTools package[1] one can access Matplotlib, Gnuplot, MATLAB, OpenDX, VisIt, and other plotting engines through the same API.

Test problem. The graphics referred to in Figure 5.1 correspond to a test problem with prescribed solution $u_{\mathrm{e}} = H(x)H(y)$, where

$$H(x) = e^{-16(x-\frac{1}{2})^2} \sin(3\pi x)\,.$$

[1] `https://github.com/hplgit/scitools`

The corresponding right-hand side f is obtained by inserting the exact solution into the PDE and differentiating as before. Although it is easy to carry out the differentiation of f by hand and hardcode the resulting expressions in an `Expression` object, a more reliable habit is to use Python's symbolic computing engine, SymPy, to perform mathematics and automatically turn formulas into C++ syntax for `Expression` objects. A short introduction was given in Section 3.2.3.

We start out with defining the exact solution in `sympy`:

```
from sympy import exp, sin, pi  # for use in math formulas
import sympy as sym

H = lambda x: exp(-16*(x-0.5)**2)*sin(3*pi*x)
x, y = sym.symbols('x[0], x[1]')
u = H(x)*H(y)
```

Turning the expression for `u` into C or C++ syntax for `Expression` objects needs two steps. First we ask for the C code of the expression:

```
u_code = sym.printing.ccode(u)
```

Printing `u_code` gives (the output is here manually broken into two lines):

```
-exp(-16*pow(x[0] - 0.5, 2) - 16*pow(x[1] - 0.5, 2))*
sin(3*M_PI*x[0])*sin(3*M_PI*x[1])
```

The necessary syntax adjustment is replacing the symbol `M_PI` for π in C/C++ by `pi` (or `DOLFIN_PI`):

```
u_code = u_code.replace('M_PI', 'pi')
u_b = Expression(u_code, degree=1)
```

Thereafter, we can progress with the computation of $f = -\nabla\cdot(\kappa\nabla u)$:

```
kappa = 1
f = sym.diff(-kappa*sym.diff(u, x), x) + \
    sym.diff(-kappa*sym.diff(u, y), y)
f = sym.simplify(f)
f_code = sym.printing.ccode(f)
f_code = f_code.replace('M_PI', 'pi')
f = Expression(f_code, degree=1)
```

We also need a Python function for the exact flux $-\kappa\partial u/\partial x$:

```
flux_u_x_exact = sym.lambdify([x, y], -kappa*sym.diff(u, x),
                              modules='numpy')
```

It remains to define `kappa = Constant(1)` and set `nx` and `ny` before calling `solver` to compute the finite element solution of this problem.

5.6 Taking the next step

If you have come this far, you have learned how to both write simple script-like solvers for a range of PDEs, and how to structure Python solvers using functions and unit tests. Solving a more complex PDE and writing a more full-featured PDE solver is not much harder and the first step is typically to write a solver for a stripped-down test case as a simple Python script. As the script matures and becomes more complex, it is time to think about design, in particular how to modularize the code and organize it into reusable pieces that can be used to build a flexible and extensible solver.

On the FEniCS web site you will find more extensive documentation, more example programs, and links to advanced solvers and applications written on top of FEniCS. Get inspired and develop your own solver for your favorite application, publish your code and share your knowledge with the FEniCS community and the world!

PS: *Stay tuned for the FEniCS Tutorial Volume 2!*

References

[1] M. S. Alnæs, A. Logg, K. B. Ølgaard, M. E. Rognes, and G. N. Wells. Unified Form Language: A domain-specific language for weak formulations of partial differential equations. *ACM Transactions on Mathematical Software*, 40(2), 2014. doi:10.1145/2566630, arXiv:1211.4047.

[2] Douglas N. Arnold and Anders Logg. Periodic table of the finite elements. *SIAM News*, 2014.

[3] W. B. Bickford. *A First Course in the Finite Element Method*. Irwin, 2nd edition, 1994.

[4] D. Braess. *Finite Elements*. Cambridge University Press, Cambridge, third edition, 2007.

[5] 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.

[6] A. J. Chorin. Numerical solution of the Navier-Stokes equations. *Math. Comp.*, 22:745–762, 1968.

[7] P. G. Ciarlet. *The Finite Element Method for Elliptic Problems*, volume 40 of *Classics in Applied Mathematics*. SIAM, Philadelphia, PA, 2002. Reprint of the 1978 original [North-Holland, Amsterdam; MR0520174 (58 #25001)].

[8] J. Donea and A. Huerta. *Finite Element Methods for Flow Problems*. Wiley Press, 2003.

[9] K. Eriksson, D. Estep, P. Hansbo, and C. Johnson. *Computational Differential Equations*. Cambridge University Press, 1996.

[10] A. Ern and J.-L. Guermond. *Theory and Practice of Finite Elements*. Springer, 2004.

[11] Python Software Foundation. The Python tutorial. `http://docs.python.org/2/tutorial`.

[12] M. Gockenbach. *Understanding and Implementing the Finite Element Method*. SIAM, 2006.

[13] 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.

[14] T. J. R. Hughes. *The Finite Element Method: Linear Static and Dynamic Finite Element Analysis*. Prentice-Hall, 1987.

[15] J. M. Kinder and P. Nelson. *A Student's Guide to Python for Physical Modeling*. Princeton University Press, 2015.

[16] Robert C. Kirby. Fiat, a new paradigm for computing finite element basis functions. *ACM Transactions on Mathematical Software*, 30(4):502–516, 2004.

[17] Robert C. Kirby and Anders Logg. A compiler for variational forms. *ACM Transactions on Mathematical Software*, 32(3):417–444, 2006.

[18] J. Kiusalaas. *Numerical Methods in Engineering With Python*. Cambridge University Press, 2005.

[19] R. H. Landau, M. J. Paez, and C. C. Bordeianu. *Computational Physics: Problem Solving with Python*. Wiley, third edition, 2015.

[20] H. P. Langtangen. *Python Scripting for Computational Science*. Springer, third edition, 2009.

[21] H. P. Langtangen. *A Primer on Scientific Programming With Python*. Texts in Computational Science and Engineering. Springer, fifth edition, 2016.

[22] H. P. Langtangen and L. R. Hellevik. Brief tutorials on scientific Python, 2016. `http://hplgit.github.io/bumpy/doc/web/index.html`.

[23] H. P. Langtangen and A. Logg. *Solving PDEs in Hours – The FEniCS Tutorial Volume II*. Springer, 2016.

[24] H. P. Langtangen and K.-A. Mardal. *Introduction to Numerical Methods for Variational Problems*. 2016. `http://hplgit.github.io/fem-book/doc/web/`.

[25] M. G. Larson and F. Bengzon. *The Finite Element Method: Theory, Implementation, and Applications*. Texts in Computational Science and Engineering. Springer, 2013.

[26] A. Logg, K.-A. Mardal, and G. N. Wells. *Automated Solution of Partial Differential Equations by the Finite Element Method*. Springer, 2012.

[27] Anders Logg and Garth N. Wells. DOLFIN: Automated finite element computing. *ACM Transactions on Mathematical Software*, 37(2), 2010.

[28] M. Pilgrim. *Dive into Python*. Apress, 2004. `http://www.diveintopython.net`.

[29] A. Quarteroni and A. Valli. *Numerical Approximation of Partial Differential Equations*. Springer Series in Computational Mathematics. Springer, 1994.

[30] A. Henderson Squillacote. *The ParaView Guide*. Kitware, 2007.

[31] R. Temam. Sur l'approximation de la solution des équations de Navier-Stokes. *Arc. Ration. Mech. Anal.*, 32:377–385, 1969.

Index