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

The FEniCS Tutorial – Writing State-of-the-Art Finite Element Solvers in Minutes

May 2, 2016

Springer

[*]Email: hpl@simula.no. Center for Biomedical Computing, Simula Research Laboratory and Department of Informatics, University of Oslo.

[†]Email: logg@chalmers.se. Department of Mathematics, Chalmers University of Technology and Center for Biomedical Computing, Simula Research Laboratory.

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. It will also deliver high performance since FEniCS automatically delegates compute-intensive tasks to C++ by help of code generation. 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* [23]. This tutorial is a further development of the opening chapter in [23].

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 [23]. We are particularly thankful to Professor Douglas Arnold for very valuable feedback on early versions of the text. Øystein Sørensen pointed out a lot of typos and contributed with many helpful comments. Many errors and typos were also reported by Mauricio Angeles, Ida Drøsdal, Miroslav Kuchta, Hans Ekkehard Plesser, Marie Rognes, and Hans Joachim Scroll. Ekkehard Ellmann as well as two anonymous reviewers provided a series of suggestions and improvements.

Special thank goes to Benjamin Kehlet for all his work with the `mshr` tool and for quickly implementing our requests in this tutorial.

Oslo, May 2016 *Hans Petter Langtangen, Anders Logg*

Watch out for shortcomings!

This book is still in an initial state so the reader is encouraged to send email to the authors on logg@chalmers.se about typos, errors, and suggestions for improvements.

Chapter 1
Preliminaries

1.1 The FEniCS Project

The FEniCS Project is a research and software project aiming 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 `http://fenicsproject.org`.

FEniCS consists of a number of building blocks (software components) that together form the FEniCS software: DOLFIN, FFC, FIAT, UFL, and a few others. 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++ from high-level mathematical abstractions. FIAT is the finite element backend of FEniCS, responsible for generating finite element basis functions, and UFL implements the abstract mathematical language by which users may express variational problems.

1.2 What you will learn

The goal of this tutorial is introduce the concept of programming finite element solvers for PDEs and get you started with FEniCS through a series of simple examples that demonstrate

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

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 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 of how easy it is to solve PDEs with FEniCS!

1.4 Obtaining the software

Reading this tutorial obviously requires access to FEniCS. 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 (GNU/Linux, Mac OS X, Windows) that 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 the 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. For more installation options, such as building FEniCS from source, check out the official FEniCS installation instructions at `http://fenicsproject.org/download`.

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, just follow the instructions from the Docker web page. Once you have installed Docker, just copy the following line into a terminal window:

Terminal

[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.

```
Terminal> curl -s http://get.fenicsproject.org | sh
```

Mac and Windows users should make sure to run this command inside the Docker Quickstart Terminal!

The command above will install the program `fenicsproject` on your system. This command 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`.

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 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. This step 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[2] 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 codes that can later be extended to solve more complex problems. All example programs are available from the web page of this book at `http://fenicsproject.org/tutorial`. The programs as well as the source code for this text can also be accessed directly from the Git repository for this book.

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

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 have some additional material on the language. A natural starting point for beginners is the classical *Python Tutorial* [11], or a tutorial geared towards scientific computing [20]. In the latter, you will also find lots of pointers to other tutorials for scientific computing in Python. Among ordinary books we recommend the general introduction *Dive into Python* [24] as well as texts that focus on scientific computing with Python [15–19].

Python versions

Python comes in two versions, 2 and 3, and these are not compatible. FEniCS has a code base that runs under both versions. All the programs in this tutorial are also developed such that they can be run under both Python 2 and 3. 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.

To start a FEniCS Python 3 session, users of FEniCS containers should run the command `fenicsproject run stable-py3`.

1.6.2 The finite element method

There are a large number of books on the finite element method. The books typically fall in either of two categories: the abstract mathematical version of the method and the engineering "structural analysis" formulation. FEniCS builds heavily on concepts from the abstract mathematical exposition. The first author has in development a book that explains all details of the finite element method in an intuitive way, though with the abstract mathematical formulations that FEniCS employ.

The finite element text by Larson and Bengzon [22] 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 interest in fluid flow problems. Hughes [14] is also recommended, especially for those 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] as an attractive bridge over to the abstract mathematical formulation that FEniCS builds upon. Those who have a weak background in differential equations in general should consult a more fundamental book, and Eriksson *et al* [9] is a very good choice. On the other hand, FEniCS users with a strong background in mathematics and interest in the mathematical properties of the finite element method, will appreciate the texts by Brenner and Scott [5], Braess [4], Ern and Guermond [10], Quarteroni and Valli [25], or Ciarlet [7].

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 the specification of the mathematical variational problem, applying boundary conditions, calling the FEniCS solver, and plotting the solution.

2.1 Mathematical problem formulation

Let us start by writing a "Hello, World!" program. In the world of PDEs, this must be a program that solves the Poisson equation:

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

$$u(\boldsymbol{x}) = u_0(\boldsymbol{x}), \quad \boldsymbol{x} \text{ on } \partial\Omega\,. \tag{2.2}$$

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

In two space dimensions with coordinates x and y, we can write out the Poisson equation 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 of more complicated systems of PDEs, in particular the Navier–Stokes equations.

Solving a PDE 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 PDE 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 most other software frameworks for PDEs require much more code and more technically difficult programming.

2.1.1 Finite element variational formulation

FEniCS is based on the finite element method, which is a general and efficient mathematical machinery for 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 brief introduction to the topic in this tutorial, but getting and reading a proper book on the finite element method in addition is encouraged. Section 1.6.2 contains a list of some suitable books.

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 test and trial function are used in FEniCS programs too. Suitable function spaces must be specified for the test and trial functions. For standard PDEs arising in physics and mechanics such spaces are well known.

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

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

We then apply integration by parts to the integrand with second-order derivatives. We find that

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

where $\frac{\partial u}{\partial n} = \nabla u \cdot n$ is the derivative of u in the outward normal direction n on the boundary. The test function v is required to vanish on the parts of the boundary where the solution u is known, which in the present problem implies that $v = 0$ on the whole boundary $\partial\Omega$. The second term on the right-hand side of (2.5) therefore vanishes. From (2.4) and (2.5) it follows that

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

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 (possibly 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

$$V = \{v \in H^1(\Omega) : v = u_0 \text{ on } \partial\Omega\},$$
$$\hat{V} = \{v \in H^1(\Omega) : v = 0 \text{ on } \partial\Omega\}\,.$$

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 also the derivatives are continuous, but the Sobolev space $H^1(\Omega)$ allows functions with discontinuous derivatives. This weaker continuity requirement of u in the variational statement (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 defines our approximate numerical solution of Poisson's equation (2.1). The mathematical framework may seem complicated at first glance, but the good news is 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 and 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:

$$a(u, v) = L(v)\,. \tag{2.9}$$

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 are then coded directly in the program.

FEniCS provides all the necessary mathematical notation needed to express the variational problem $a(u,v) = L(v)$. To solve a linear PDE in FEniCS, such as the Poisson equation, a user thus needs to perform only two steps:

- Express the PDE as a (discrete) variational problem: find $u \in V$ such that $a(u,v) = L(v)$ for all $v \in \hat{V}$.
- 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).

2.1.3 Choosing a test problem

The Poisson equation (2.1) has so far featured a general domain Ω and general functions u_b and f. For our first implementation, we must decide on specific choices of Ω, u_b, and f. It will be wise to construct a specific problem where 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

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

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

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

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 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 can 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 5.4.2. 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 u_{b}, f, and Ω 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_b = 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_b, 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

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

# Plot solution
u.rename('u', 'solution')
plot(u)
plot(mesh)

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

# Compute error in L2 norm
error_L2norm = errornorm(u_b, u, 'L2')

# Compute maximum error at vertices
vertex_values_u_b = u_b.compute_vertex_values(mesh)
vertex_values_u  = u.compute_vertex_values(mesh)
import numpy as np
error_vertices = np.max(np.abs(vertex_values_u_b - vertex_values_u))

# Print errors
print('error_L2norm   =', error_L2norm)
print('error_vertices =', error_vertices)

# Hold plot
interactive()
```

The complete code 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 intergrated 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`.

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_e and the error in the maximum norm will be printed. Since we know that our approximate solution should reproduced the exact solution to within machine precision, this error should be small, something on the order of 10^{-15}.

AL 1: Add text here discussing what to expect in terms of plotting. Perhaps we have seamless notebook plotting working soon...

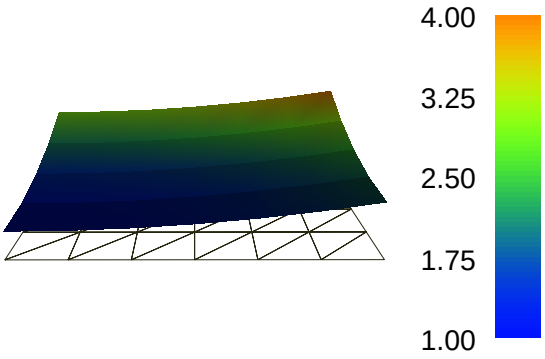


Fig. 2.1 Plot of the solution in the first FEniCS example.

Spyder. Many prefer to work in an integrated development environment where there is an editor for programming, a window for executing code, a window for inspecting objects, etc. The Spyder tool comes with all major Python installations. 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. Start `jupyter notebook` from a terminal window, find the **New** pulldown menu in the upper right part 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 write Shift+Enter to execute the cell. The file `ft01_poisson.py` will the 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**.

hpl 2: Need to describe this with more care. The first program seems to have some problems with printing the error to the notebook unless we drop the plot commands. Anyway, there should be in-browser plot commands.

2.3 Dissection of the program

We shall now dissect this FEniCS program in detail. The program is written in the Python programming language. You may either take a quick look at the official Python tutorial to pick up the basics of Python if you are unfamiliar with the language, or you may learn enough Python as you go along with the examples in the present tutorial. The latter strategy has proven to work for many newcomers to FEniCS. This is because both the amount of abstract mathematical formalism and the amount of Python expertise that is actually needed to be productive with FEniCS is quite limited. And Python is an easy-to-learn language that you will certainly come to love and use far beyond FEniCS programming. Section 1.6.1 lists some relevant Python books.

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_{b}), and the bilinear and linear forms $a(u,v)$ and $L(v)$. Thereafter, the unknown trial function u is computed. Then we can compare the numerical and exact solution as well as visualize the computed solution u.

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.

hpl 3: Note that plot was made by the old partitioning 6×4. Probably no issue.

2.3.3 Defining the finite element function space

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

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

The second argument `'P'` specifies the type of element, while the third argument is the degree of the basis functions of the element. The type of element is here "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].

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 and linearly varying in x and y over each cell in the mesh. Higher-degree polynomial approximations over each cell are trivially obtained by increasing the third parameter 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

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

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

2.3.5 Defining the boundary and the boundary conditions

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

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

where `u_b` 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 $u = u_{\mathrm{b}}$ 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_b` refers to an `Expression` object, which is used to represent a mathematical function. The typical construction is

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

where `formula` is a string containing the mathematical expression. This formula is written with C++ syntax. The expression is automatically turned into an efficient, compiled C++ function. The second argument `degree` is a parameter that specifies how the expression should be treated in computations. FEniCS will interpolate the expression into some finite element space. It is usually a good choice to interpolate expressions into the same space V that is used for the trial and test functions, but in certain cases, one may want to use a more accurate (higher degree) representation of expressions.

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 $u_{\mathrm{b}}(x,y) = 1 + x^2 + 2y^2$, the formula string can be written as `1 + x[0]*x[0] + 2*x[1]*x[1]`:

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

String expressions must have valid C++ syntax!

The string argument to an `Expression` object must obey C++ syntax. Most Python syntax for mathematical expressions are also valid C++ syntax, but power expressions make an exception: `p**a` must be written as `pow(p,a)` in C++ (this is also an alternative Python syntax). The following mathematical functions can be used directly in C++ expressions when defining `Expression` objects: `cos`, `sin`, `tan`, `acos`, `asin`, `atan`, `atan2`, `cosh`, `sinh`, `tanh`, `exp`, `frexp`, `ldexp`, `log`, `log10`, `modf`, `pow`, `sqrt`, `ceil`, `fabs`, `floor`, and `fmod`. Moreover, the number π is available as the symbol `pi`. All the listed functions are taken from the `cmath` C++ header file, and one may hence consult 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=1)
```

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=1,
               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 allows us to have boundaries where u are known also 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

```
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 with 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)
```

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=1)
```

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, 23] and is an integral part of FEniCS.

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 often done 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

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

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

Clicking on `Help` or typing `h` in the plot windows brings up a list of commands. For example, typing `m` brings up the mesh. With the left, middle, and right mouse buttons you can rotate, translate, and zoom (respectively) the plotted surface to better examine what the solution looks like. You must click

`Ctrl+q` to kill the plot window and continue execution beyond the command interactive()‘. In the example program, we have therefore placed the call to `interactive()` at the very end. Alternatively, one may use the command `plot(u, interactive=True)` which again means you can interact with the plot window and that execution will be halted until the plot window is closed.

Figure 2.1 displays the resulting u function.

2.3.10 Exporting and post-processing the solution

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

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

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

Prior to plotting and storing solutions to file it is wise to give `u` a proper name by `u.rename('u', 'solution')`. Then `u` will be used as name in plots (rather than the more cryptic default names like `f_7`).

Once the solution has been stored to file, it can be opened in Paraview by choosing **File - Open**. Find the file `poisson.pvd`, and click the green **Apply** button to the left in the GUI. A 2D color plot of $u(x,y)$ is then shown. You can save the figure to file by **File - Export Scene...** and choosing a suitable filename. For more information about how to install and use Paraview, see the `http://www.paraview.org/`.

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 `u_b`, which in this example happens to be the same as the `Expression` 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

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

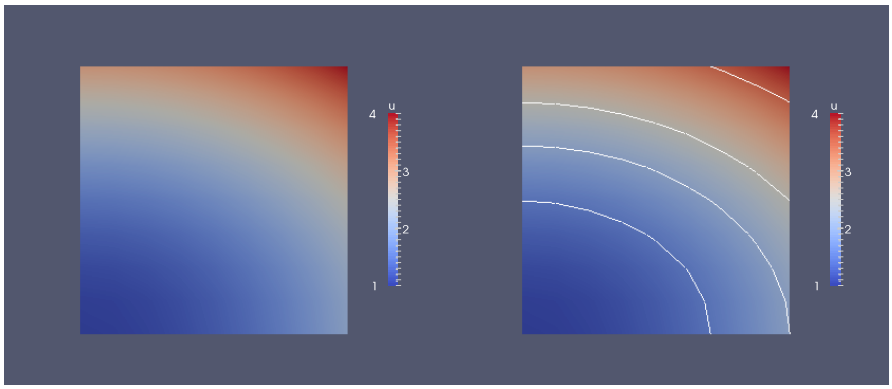


Fig. 2.2 Visualization of test problem in ParaView, with contour lines added in the right plot.

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_L2norm = errornorm(u_b, 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_b` and `u` at all vertices, and then subtract the results:

```
vertex_values_u_b = u_b.compute_vertex_values(mesh)
vertex_values_u  = u.compute_vertex_values(mesh)
import numpy as np
error_vertices = np.max(np.abs(vertex_values_u_b - vertex_values_u))
```

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

How to check that the error vanishes?

With inexact arithmetics, as we always have on a computer, 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 81 linear elements the error is about $2 \cdot 10^{-15}$.

2.3.12 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 $U_1, \ldots, U_N$ values. The $U_1, \ldots, U_N$ values 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 location 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 values U directly by

```
u_nodal_values = 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:

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

With `numpy` arrays we can write MATLAB-like code to analyze the data. Indexing is done with square brackets: `u_array[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 `u_array` is equal to the number of vertices, and each `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, see Section 5.1.3 for details. 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:

```
u_vertex_values = u.compute_vertex_values()
```

Note that `u_array` and `u_vertex_values` are arrays of the same length and containing the same values, albeit in different order.

2.4 Deflection of a membrane

The previous problem and code targeted a simple test problem where we can easily verify the implementation. Now we turn the attention to a more physically relevant problem, in a non-trivial geometry, and that results in solutions of somewhat more exciting shape.

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

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

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 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.

2.4.1 Scaling

The localization of the pressure, (x_0,y_0), is for simplicity set to $(0,R_0)$. 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 get

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

where

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

With an appropriate scaling, $\bar{w}$ 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 unit. We shall in particular choose $\alpha = 4$. With this value, the solution is $w(\bar{x}, \bar{y}) = 1 - \bar{x}^2 - \bar{y}^2$. (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 get 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 circle 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 $\beta \to 0$, we have a special case with solution $w = 1 - x^2 - y^2$.

Given a computed w, the physical deflection is given by

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

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

2.4.2 Defining the mesh

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

```
from mshr import *
domain = Circle(Point(0.0, 0.0), 1.0)
n = 20
mesh = generate_mesh(domain, n)
plot(mesh, interactive=True)
```

The `Circle` shape from `mshr` takes the center and radius of the circle as the two first arguments, while `n` is the resolution, here the suggested number of cells per radius.

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)))',
    beta=beta, R0=R0)
```

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

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

2.4.4 Variational form

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 Visualization

It would be of interest to visualize p along with w so that we can examine the pressure force and the membrane's response. We must then transform the formula (`Expression`) to a finite element function (`Function`). The most natural approach is to construct a finite element function whose degrees of freedom are calculated from p. That is, we interpolate p:

```
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`.

We can now plot `w` and `p` on the screen as well as dump the fields to file in VTK format:

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

vtkfile1 = File('membrane_deflection.pvd')
vtkfile1 << w
vtkfile2 = File('membrane_load.pvd')
vtkfile2 << p
```

Figure 2.3 shows the result of the `plot` commands.

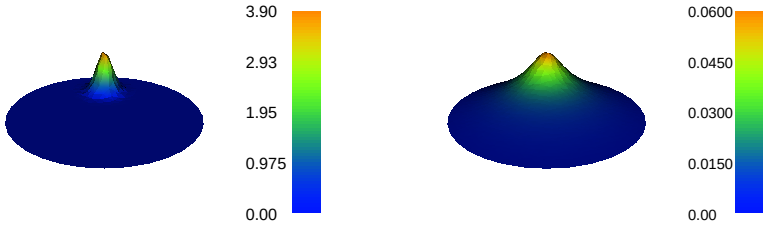


Fig. 2.3 Load (left) and resulting deflection (right) of a circular membrane.

2.4.6 Curve plots through the domain

The best way to compare the load and the deflection is to make a curve plot along the line $x = 0$. This is just a matter of defining a set of points along the line 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 = 1E-8  # 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, 100*w_line, 'r-', y, p_line, 'b--') # magnify w
plt.legend(['100 x deflection', 'load'], loc='upper left')
plt.xlabel('y');  plt.ylabel('$p$ and $100u$')
```

(Remember a `plt.show()` at the end to show the plot on the screen.) The resulting curve plot appears in Figure 2.4. It is seen how the localized input

(p) is heavily damped and smoothened in the output (w). This reflects a typical property of the Poisson equation.

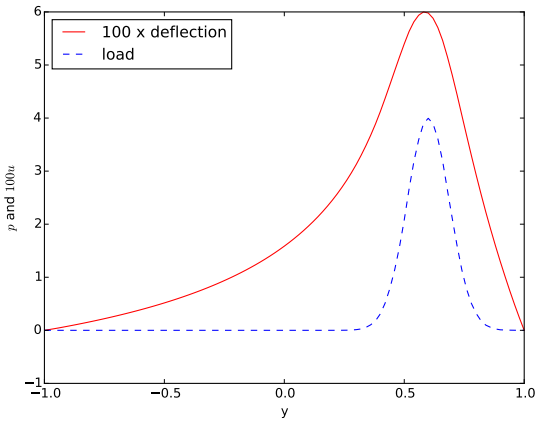


Fig. 2.4 Comparison of membrane load and deflection.

The complete program is available in the file `ft02_membrane.py`.

2.4.7 Running ParaView

ParaView is a very strong and well-developed tool for visualizing scalar and vector fields, including those computed by FEniCS.

Our program file writes w and p to file as finite element functions. The default filenames are `membrane_deflection.pvd` for w and `membrane_load.vtu` for p. These files are in VTK format and their data can be visualized in ParaView.

1. Start the ParaView application.
2. Open a file with **File - Open...**. You will see a list of `.pvd` and `.vtu` files. More specifically you see `membrane_deflection.pvd`. Choose this file.
3. Click on **Apply** to the left (*Properties* pane) in the GUI, and ParaView will visualize the contents of the file, here as a color image.
4. To get rid of the axis in the lower left corner of the plot area and axis cross in the middle of the circle, find the *Show Orientation Axis* and *Show Center* buttons to the right in the second row of buttons at the top of the GUI. Click on these buttons to toggle axis information on/off.
5. If you want a color bar to explain the mapping between w values and colors, go to the *Color Map Editor* in the right of the GUI and use the

Show/hide color legend button. Alternatively, find *Coloring* in the lower left part of the GUI, and toggle the *Show* button.

6. The color map, by default going from blue (low values) to red (high values), can easily be changed. Find the *Coloring* menu in the left part of the GUI, click *Edit*, then in the *Color Map Editor* double click at the left end of the color spectrum and choose another color, say yellow, then double click at the right and of the spectrum and choose pink, scroll down to the bottom of the dialog and click *Update*. The color map now goes from yellow to pink.
7. To save the plot to file, click on **File - Export Scene...**, fill in a filename, and save. See Figure 2.5 (middle).
8. To change the background color of plots, choose **Edit - Settings...**, **Color** tab, click on **Background Color**, and choose it to be, e.g., white. Then choose **Foreground Color** to be something different.
9. To plot the mesh with colors reflecting the size of w, find the *Representation* drop down menu in the left part of the GUI, and replace *Surface* by *Wireframe*.
10. To overlay a surface plot with a wireframe plot, load w and plot as surface, then load w again and plot as wireframe. Make sure both icons in the *Pipeline Browser* in the left part of the GUI are *on* for the `membrane_deflection.pvd` files you want to display. See Figure 2.5 (left).
11. Redo the surface plot. Then we can add some contour lines. Press the semi-sphere icon in the third row of buttons at the top of the GUI (the so-called *filters*). A set of contour values can now be specified at in a dialog box in the left part of the GUI. Remove the default contour (0.578808) and add 0.01, 0.02, 0.03, 0.04, 0.05. Click *Apply* and see an overlay of white contour lines. In the *Pipeline Browser* you can click on the icons to turn a filter on or off.
12. Divide the plot window into two, say horizontally, using the top right small icon. Choose the **3D View** button. Open a new file and load `memberane_load.pvd`. Click on **Apply** to see a plot of the load.
13. To plot a 2D scalar field as a surface, load the field, click **Apply** to plot it, then select from the **Filters** pulldown menu the filter *Wrap By Scalar*, click **Apply**, then toggle the **2D** button to **3D** in the Layout `#1` window (upper row of buttons in that window). Now you can rotate the figure. The height of the surface is very low, so go to the *Properties (Warp By Scalar1)* window to the left in the GUI and give a *Scale Factor* of 20 and re-click **Apply**. Figure 2.5 (right) shows the result. to lift the surface by a factor of 20.

A particularly useful feature of ParaView is that you can record GUI clicks (**Tools - Start/Stop Trace**) and get them translated to Python code. This allows you automate the visualization process. You can also make curve plots along lines through the domain, etc.

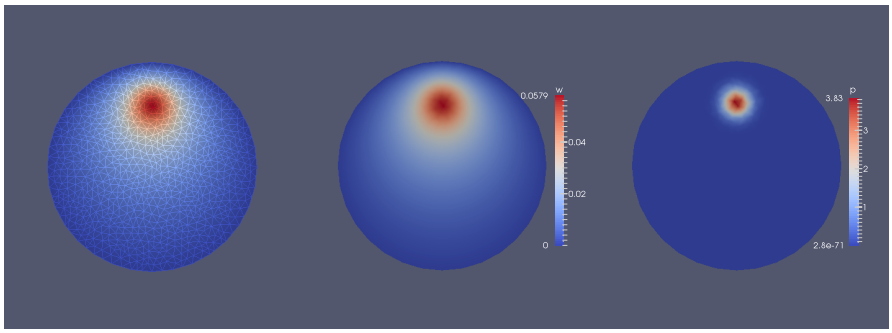


Fig. 2.5 Default visualizations in ParaView: deflection (left, middle) and pressure load (right).

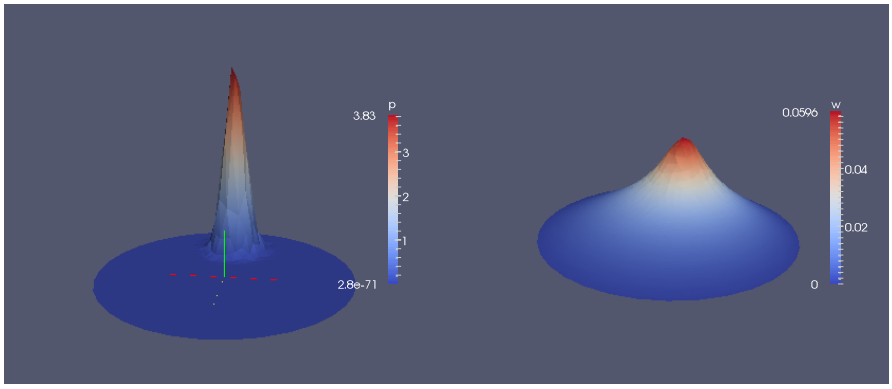


Fig. 2.6 Use of Warp By Scalar filter to create lifted surfaces (with different vertical scales!) in ParaView: load (left) and deflection (right).

For more information, we refer to The ParaView Guide [26] (free PDF available) and to the ParaView tutorial as well as an instruction video.

2.4.8 Using the built-in visualization tool

This section explains some useful visualization features of the built-in visualization tool in FEniCS. The `plot` command applies the VTK package to visualize finite element functions in a very quick and simple way. The command is ideal for debugging, teaching, and initial scientific investigations. The visualization can be interactive, or you can steer and automate it through program statements. More advanced and professional visualizations are usually better created with advanced tools like Mayavi, ParaView, or VisIt.

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

```
plot(mesh, title='Finite element mesh')
plot(w, wireframe=True, title='Solution')
```

Axes can be turned on by the `axes=True` argument, while `interactive=True` makes the program hang at the plot command - you have to type `q` in the plot window to terminate the plot and continue execution.

The left mouse button is used to rotate the surface, while the right button can zoom the image in and out. Point the mouse to the `Help` text down in the lower left corner to get a list of all the keyboard commands that are available.

The plots created by pressing `p` or `P` are stored in filenames having the form `dolfin_plot_X.png` or `dolfin_plot_X.pdf`, where `X` is an integer that is increased by one from the last plot that was made. The file stem `dolfin_plot_` can be set to something more suitable through the `hardcopy_prefix` keyword argument to the `plot` function, for instance, `plot(f, hardcopy_prefix='pressure')`.

Plots stored in PDF format need to be rotated 90 degrees before inclusion in documents. This can be done by the `convert -rotate 90` command (from the ImageMagick utility), but the resulting file has then no more high-resolution PDF vector graphics. A better solution is therefore to use `pdftk` to preserve the vector graphics:

Terminal

```
Terminal> pdftk dolfin_plot_1.pdf cat 1-endnorth output out.pdf
```

For making plots in batch, we can do the following:

```
viz_w = plot(w, interactive=False)
viz_w.elevate(-10)  # adjust (lift) camera from the default view
viz_w.plot(w)       # bring new settings into action
viz_w.write_png('deflection')  # make deflection.png
viz_w.write_pdf('deflection')  # make deflection.pdf
# Rotate pdf file (right) from landscape to portrait
import os
os.system('pdftk deflection.pdf cat 1-endnorth output w.pdf')
```

The ranges of the color scale can be set by the `range_min` and `range_max` keyword arguments to `plot`. The values must be `float` objects. These arguments are important to keep fixed for animations in time-dependent problems.

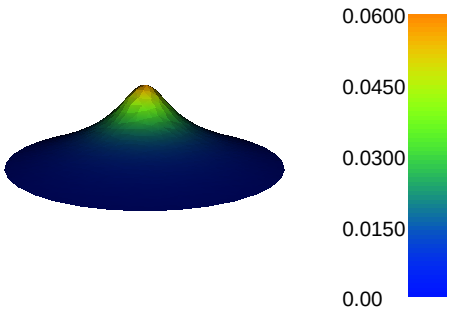


Fig. 2.7 Plot of the deflection of a membrane.

Exercise 2.1: Visualize a solution in a cube

Solve the problem $-\nabla^2 u = f$ on the unit cube $[0,1]\times[0,1]\times[0,1]$ with $u_0 = 1+x^2+2y^2-4z^2$ on the boundary. Visualize the solution. Explore both the built-in visualization tool and ParaView.

Solution. As hinted by the filename in this exercise, a good starting point is the `solver` function in the program `ft06_poisson_func.py`, which solves the corresponding 2D problem. Only two lines in the body of `solver` needs to be changed (!): `mesh = ...`. Replace this line with

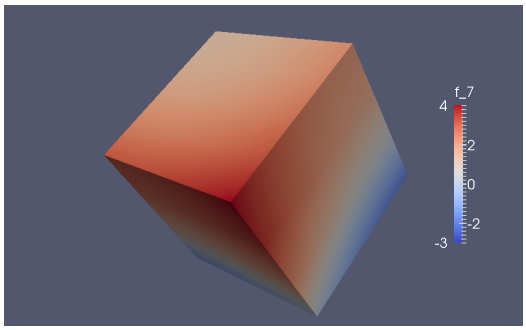
```
mesh = UnitCubeMesh(Nx, Ny, Nz)
```

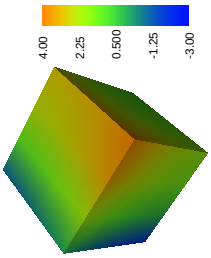
and add `Nz` as argument to `solver`. We implement the new u_0 function in `application_test` and realize that the proper $f(x,y,z)$ function in this new case is 2.

```
u0 = Expression('1 + x[0]*x[0] + 2*x[1]*x[1] - 4*x[2]*x[2]')
f = Constant(2.0)
u = solver(f, u0, 6, 4, 3, 1)
```

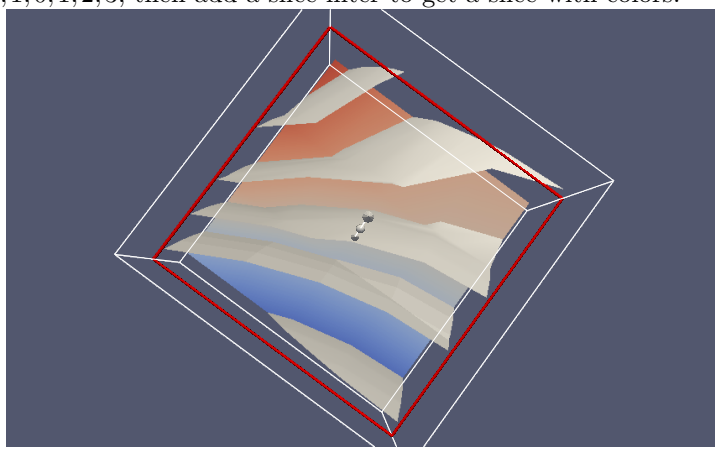
The numerical solution is without approximation errors so we can reuse the unit test from 2D, but it needs an extra `Nz` parameter.

The variation in u is only quadratic so a coarse mesh is okay for visualization. Below is plot from the ParaView (left) and the built-in visualization tool (right). The usage is as in 2D, but now one can use the mouse to rotate the 3D cube.





We can in ParaView add a contour filter and define contour surfaces for $u = -2, 1, 0, 1, 2, 3$, then add a slice filter to get a slice with colors:



Filename: `poissin_3d_func`.

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, and the Navier-Stokes equations. These problems illustrate how to solve time-dependent problems, nonlinear problems, vector-valued problems, and systems of PDE. For each problem, we derive the variational formulation and express the problem in Python in a way that closely resembles the mathematics.

AL 4: We say "systems of PDE" but we use a splitting method. We have not really taked about systems (using mixed function spaces).

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 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

$$\frac{\partial u}{\partial t} = \nabla^2 u + f \text{ in } \Omega, \tag{3.1}$$

$$u = u_\mathrm{b} \text{ on } \partial\Omega, \tag{3.2}$$

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

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_b 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:

$$\frac{\partial}{\partial t} u^n = \nabla^2 u^n + f^n\,. \tag{3.4}$$

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

$$\frac{\partial}{\partial t} u^n \approx \frac{u^n - u^{n-1}}{\Delta t}, \tag{3.5}$$

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

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

This is our time-discrete version of the heat equation (3.1). This is a so-called *backward Euler* or *implicit Euler* discretization. Alternatively, we may also view this as a finite element discretization in time in the form of the first order dG(0) method, which here is identical to the backward Euler method.

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

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

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

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:

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

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 (which is natural in the program), the resulting weak form arising from formulation (3.8) can be conveniently written in the standard notation:

$$a(u,v) = L_n(v),$$

where

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

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

The alternative form (3.9) has an abstract formulation

$$F(u;v) = 0,$$

where

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

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 one statement, using either the `project` or `interpolate` function.

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 $u^n \in V$ such that $a(u^n, v) = L_n(v)$ for all $v \in \hat{V}$, or alternatively, $F(u^n, v) = 0$ for all $v \in \hat{V}$, for $n = 1, 2, \ldots$.

3.1.3 A simple FEniCS implementation

Our program needs to implement the time-stepping manually, but can rely on FEniCS to easily compute a_0, L_0, F, a, and L, and solve the linear systems for the unknowns.

Test problem. 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 $u_{\mathrm{b}}(x, y, t) = 1 + x^2 + \alpha y^2 + \beta t$ 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 condition $u_{\mathrm{b}}(x, y, t) = 1 + x^2 + \alpha y^2 + \beta t$. 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_b = 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 parameters can later be updated as in

```
u_b.t = t
```

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

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

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

We shall use `u` for the unknown u^n at the new time level and `u_p` for u^{n-1} at the previous time level. The initial value of `u_p` can be computed by either projection or interpolation of u_0. Since we set `t = 0` for the boundary value `u_b`, we can use this also to specify the initial condition. We can then do

```
u_p = project(u_b, V)
# or
u_p = interpolate(u_b, V)
```

Projecting versus interpolating the initial condition

To actually recover the exact solution (3.15) to machine precision, it is important not to compute the discrete initial condition by projecting u_0, but by interpolating u_0 so that the degrees of freedom have exact values at $t = 0$ (projection results in approximative 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 that go into the bilinear form a and which that 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_p + 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 xrange(num_steps):

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

    # Solve variational problem
```

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


    # Update previous solution
    u_p.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 containing the values at the previous time step. This must be done using the `assign` member function. If we instead try to do `u_p = u`, we will set the `u_p` Python 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_b.t` must be updated before the `solve` statement to enforce computation of Dirichlet conditions at the current time level. (The Dirichlet conditions look up the `u_b` object for values.)

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

```
u_e = interpolate(u_b, V)
error = np.abs(u_e.vector().array() - u.vector().array()).max()
print('error, t=%.2f: %-10.3g' % (t, error))
```

The complete program code for this time-dependent case 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_b = 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_b, boundary)

# Define initial value
u_p = interpolate(u_b, V)
#u_p = project(u_b, 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_p + dt*f)*v*dx
a, L = lhs(F), rhs(F)

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

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

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

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

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

The code is available in the file `ft03_heat.py`.

3.1.4 Diffusion of a Gaussian function

The mathematical problem. Now we want to solve a more relevant test problem, namely the diffusion of a Gaussian hill. It means that the initial value is given by

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

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

FEniCS implementation. Which are the required changes to our previous program? One major change is that the domain is not a unit square anymore. We also want to use much higher resolution. The new domain can be created easily in FEniCS using `RectangleMesh`:

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

We also need to redefine the initial condition and boundary condition. Both are easily changed by defining a new `Expression` and by setting $u = 0$ on the boundary. We will also save the solution to file in VTK format in each time step:

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

AL 5: Remember to output as `(u, t)` also in other examples below.

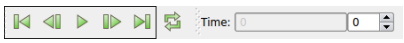
The complete program appears below.

```
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_p = interpolate(u_0, V)
u_p.rename('u', 'initial value')
vtkfile = File('gaussian_diffusion.pvd')
vtkfile << (u_p, 0.0)

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

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

```
# Compute solution
u = Function(V)
u.rename('u', 'solution')
t = 0
for n in xrange(num_steps):

    # Update current time
    t += dt

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

    # Save to file and plot solution
    vtkfile << (u, float(t))
    plot(u)
    time.sleep(0.3)

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

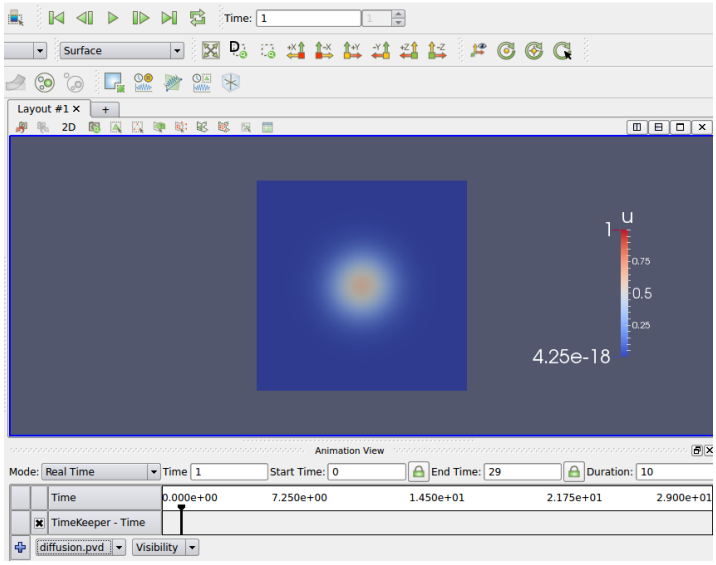
This program is also available in the file `ft04_gaussian_diffusion.py`.

Visualization in ParaView. To visualize the diffusion of the Gaussian hill, start ParaView, choose **File - Open**, open the file `gaussian_diffusion.pvd`, click the green **Apply** button on the left to see the initial condition being plotted. Choose **View - Animation View**. Click on the play button or (better) the next frame button in the row of buttons at the top of the GUI to see the evolution of the scalar field you have just computed:

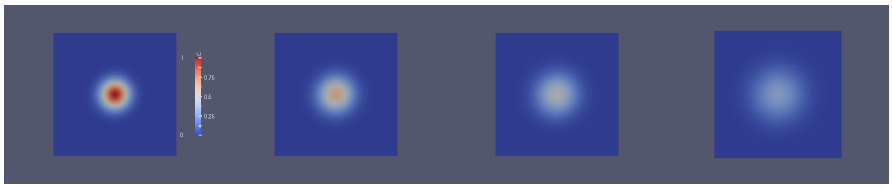


The cross in the middle of the plot can be turned off by the **Show Center** button:

Choose **File - Save Animation...** to save the animation to the AVI or OGG video format.



Once the animation has been saved to file, you can play the animation offline using a player such as mplayer or VLC, or upload your animation to YouTube. Below is a sequence of snapshots of the solution.



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 sample PDE for the implementation of nonlinear problems, we take the following nonlinear Poisson equation:

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

in Ω, with $u = u_{\mathrm{b}}$ on the boundary $\partial\Omega$. The coefficient $q(u)$ 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

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

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 a nonlinear function of u, the variational statement gives rise to a system of nonlinear algebraic equations in the unknowns $U_1,\ldots,U_N$.

3.2.3 A simple 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_{b}. Previously, we have worked with manufactured solutions that can be reproduced without approx-

imation 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 try out a two-dimensional manufactured solution that is linear in the unknowns:

```
# 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):
    """Nonlinear coefficient in the PDE."""
    return 1 + u**2

# Use SymPy to compute f given 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)
```

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)
```

Sometimes, we need some editing of 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 our case here, the output of `c_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_b` as

```
u_b = Expression(u_code)
```

Similarly, we define the right-hand side function as

```
f = Expression(f_code)
```

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 as fe` and then prefix all FEniCS object names by `fe`. 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 working solver for the nonlinear Poisson equation is as easy to implement as a solver for the corresponding linear problem. 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):
    """Nonlinear coefficient in the PDE."""
    return 1 + u**2

mesh = UnitSquareMesh(32, 32)
V = FunctionSpace(mesh, 'P', 1)
u_b = Expression(...)

def boundary(x, on_boundary):
    return on_boundary

bc = DirichletBC(V, u_b, boundary)

# Define variational problem
u = Function(V)
v = TestFunction(V)
f = Expression(...)
F = q(u)*dot(grad(u), grad(v))*dx - f*v*dx

# Compute solution
```

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

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 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.

AL 6: Should we display all codes like this one? **hpl 7**: Now experimenting with refernce to repo and in html and sphinx a button will fold out the code.

The complete code is found in the file `ft05_nlpoisson.py`.

Running the code gives output that tells how the Newton iteration progresses. With $2(6\times 4)$ cells we get convergence in 7 iterations with a tolerance of 10^{-9}, and the error in the numerical solution is about 10^{-11}. With $2(3\times 3)$ and $2(8\times 8)$ cells the error is identically zero. Other resolutions may bring the error up to the level of the tolerance in the Newton iterations. These results bring evidence for a correct implementation. Thinking in terms of finite differences on a uniform mesh, P1 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 P1 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 in 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. Therefore, we return to this problem in Chapter 3 in [21] and show how we can implement our own solution algorithms for nonlinear equations and also how we can steer the parameters in the automated Newton method used above. You will then see how easy it is to implement tailored solution strategies for nonlinear problems in FEniCS.

3.3 The equations of linear elasticity

Analysis of structures is one of the major activities of modern engineering, thus making the PDEs for deformation of elastic bodies likely the most popular PDE model 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 Lame's elasticity parameters for the material in Ω, I is the identity tensor, tr is the trace operator on a tensor, ε is the strain 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 splitting of the equations 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 test vector 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 assume that it is prescribed at a part $\partial\Omega_T$ of the boundary and set $T = \sigma\cdot n$. On the remaining

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

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

Inserting the expression (3.23) for σ gives the variational form with u as unknown. Note that the boundary integral on the remaining part $\partial\Omega \setminus \Omega_T$ vanishes due to the Dirichlet condition ($v = 0$).

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 a non-symmetric tensor B vanishes. If we express ∇v as a sum of its symmetric and non-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 is a more popular formulation than (3.25).

3.3.3 A simple FEniCS implementation

Test problem. As a test example, we may look at a clamped beam deformed under its own weight. Then $f = (0, 0, -\varrho g)$ is the body force per unit volume with ϱ the density of the beam and g the acceleration of gravity. The beam is box-shaped with length L and square cross section of width W. We set

$u = (0,0,0)$ at the clamped end, $x = 0$. The rest of the boundary is traction free; that is, we set $T = 0$.

The code. We first list the code and then comment upon the new constructions compared to the Poisson equation case.

```
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 conditions
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)

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()  # no of space dim
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 and mesh
plot(u, title='Displacement', mode='displacement')

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')
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())
```

New feature: vector function space. 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 finite element function.

New feature: constant vectors. In the boundary condition $u = 0$, we must set a vector value to zero, not just a scalar, and a constant zero vector 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))`.

New feature: `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.

New feature: stress computation. As soon as `u` is computed, we can compute various stress measures, here 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.

Scaling. Before doing simulations, it is often advantageous to scale the problem as it reduces the need for setting physical parameters, and one obtains dimensionsless numbers that reflect the competition of parameters and physical effects. These numbers are often easy to assign values for scientific investigations.

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

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

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

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

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

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

is also 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.

AL 8: Need to change above scaling argument now that ϱ is not part of the equation? **hpl 9**: No, we just inserted our particular f which is ϱg.

AL 10: W not defined below. Width? **hpl 11**: Defined in the test problem in the intro. But a comment what L/W is, is now inserted.

Sometimes, one will argue to chose U to make γ unity ($U=\varrho g L^2/\mu$). However, in elasticity, this leads us 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. Actually, for a clamped beam, one has a deflection formula which gives $U=\frac{3}{2}\varrho g L^2\delta^2/E$, where $\delta=L/W$ is a parameter reflecting how slender the beam is. 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, we realize that $\gamma\sim\delta^{-2}$. Experiments with the code point to $\gamma=0.4\delta^{-2}$ as an appropriate choice of γ.

The simulation code implements the problem with dimensions and physical parameters λ, μ, ϱ, g, L, and W. However, we can usually easily reuse this code for a scaled problem. In the present case, we just set $\mu=\varrho=L=1$, W as W/L, $g=\gamma$, and $\lambda=\beta$.

AL 12: I find this somewhat confusing. First we talk about a rescaled equation but then we solve the unscaled equation, but we choose the parameters so that it is somehow related to the scaled problem...?

AL 13: Need to look at code again once I have understood the scaling.

hpl 14: This is the way to do it ;-) Implement with dimensions to have the code as general as possible. Then scale a particular problem – and the scaling is restricted to this problem. It makes it much easier to set parameters for numerical investigations and to understand the model and what the competing effects are. You can run the scaled model with the original program by a proper choice of parameters.

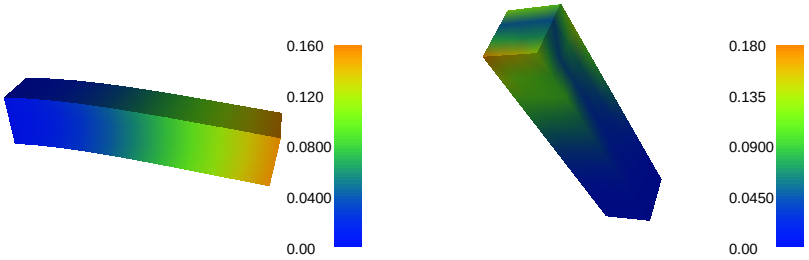


Fig. 3.1 Gravity-induced deformation of a clamped beam: deflection (left) and stress intensity seen from below (right).

3.4 The Navier–Stokes equations

As our final example in this chapter, 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.

3.4.1 PDE problem

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

hpl 15: Quite uncommon to write $\dot{u}$ for the time-derivative in the N-S equations. Only Claes Johnson comes to my mind... In mechanics, the dot is reserved for ODEs. I suggest the more common notation $\partial u/\partial t$, also since we use this elsewhere in the books.

$$\varrho(\dot{u} + u \cdot \nabla u) = \nabla \cdot \sigma(u,p) + f, \tag{3.29}$$

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

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

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

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 the two additional terms $\varrho(\dot{u}+u\cdot\nabla u)$ and the different expression for the stress tensor. The two extra terms express the acceleration $\rho\ddot{x}$ balanced by the force $F=f+\nabla\cdot\sigma$ 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 face two challenges. First, 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 (preconditioners and iterative methods) to be solved efficiently.

Instead, we will apply a simpler and often very efficient approach which is to use a *splitting method*. In a splitting method, we 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 [27], 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) using the pressure p^{n-1} from the previous time interval. We will also be using the velocity u^{n-1} in the nonlinear term $u\cdot\nabla u$. The variational problem for this first step is:

hpl 16: This equation applies `textit` and `multiline` in math. At least `textit` does not work in MathJax. Must test what works in Sphinx. Answer: Sphinx MathJax did not accept anything of this.

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

This notation 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 means the value of u at the midpoint of the interval, usually approximated by an arithmetic mean

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

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 e 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 $pn - \nu \nabla u \cdot n$ which is the term appearing in the variational problem (3.32).

hpl 17: Here a boundary term $(\mu n \cdot \nabla u^{n-\frac{1}{2}}, v)$ is missing. This is the intricate discussions we had back in 2009-2010 with Harish on using N-S with σ or $\nabla^2 u$.

AL 18: Apostrophes like "these" don't look like what I would expect in LaTeX. **hpl 19**: No, double quotes must be written as in LaTeX "quotes". Two backticks and two forward ticks become the right double quotes in various output formats.

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

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

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

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 alternative 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 $u \cdot \nabla u$.

hpl 20: I like the straightforward formulation above, but it should be complemented by the arguments below since it is not a matter of taste in the end, but dictated by derivation of the PDE and what ∇ was meant to be there.

To be more precise, there are three different notations used for PDEs involving gradient, divergence, and curl operators. One employs $\mathrm{grad}\, u$, $\mathrm{div}\, u$, and $\mathrm{curl}\, u$ operators. Another employs ∇u as a synonym for $\mathrm{grad}\, u$, $\nabla \cdot u$ means $\mathrm{div}\, u$, and $\nabla \times u$ is the name for $\mathrm{curl}\, u$. 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 ($(\nabla u)_{i,j} = \partial u_j/\partial x_i = (\mathrm{grad} u)^T$). 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 $(u \cdot \nabla)u$), ∇ is a vector operator, reading `dot(u, nabla_grad(u))` in FEniCS, but if we see $\nabla u \cdot u$ or $(\mathrm{grad} u) \cdot u$, the corresponding FEniCS expression is `dot(grad(u), u)`.

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:

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

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

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

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

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

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

hpl 21: Check that ρ is correctly handled in the three steps.

In summary, we may thus solve the incompressible Navier-Stokes equations efficiently by solving a sequence of three linear variational problems (steps 1, 2, 3) in each time step.

3.4.3 A simple FEniCS implementation

Test problem 1. As a first test problem, we compute the flow between two infinite plates, so-called channel or Poiseuille flow, since 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 in diffusion accross the channel: $t_c = H^2/\nu$. We let U, some characteristic inflow, 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):

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

Here, Re is the Reynolds number $\rho UH/\mu$. Because of the time and pressure scale, which are different from convection-dominated fluid flow, the Reynolds number is associated with the convective term and not the viscosity term (as usual). Note that the last term in the first equation is zero, but we included this term as it arises naturally from the original $\nabla\cdot\sigma$ 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 $\partial p/\partial x$ is a constant, here called $-\beta$. This is the driving force of the flow and specified as known in the problem. Integrating $u_x''(y) = -\beta$ over the width of the channel, $[0,1]$, and requiring $u = 0$ at the

channel walls, results in $u_x = \frac{1}{2}\beta y(1-y)$. The characteristic inlet flow in the channel, U, can be taken as the maximum inflow at $x = 1/2$, implying that $\beta = 8$. 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. The pressure can then be set to $p = 0$ at the outlet $x = 1$, giving $p(x) = 8(1-x)$ and $u_x = 4y(1-y)$.

The boundary conditions can be taken as $p = 1$ on $x = 0$, $p = 0$ on $x = 1$ and $u = 0$ 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.

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 $\rho = \mu = 1$ in the original Navier-Stokes equations resembles the scaled model.

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 splitting scheme we will use to solve the Navier-Stokes equations we 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 `P` 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 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 piecewise 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. Alternatively, we may give the boundary definition as a string of C++ code, much like we have previously defined expressions such as `u0 = Expression('1 + x[0]*x[0] + 2*x[1]*x[1]')`. 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 boundary to C++ from Python, 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 $p = 1$ 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 an initial stationary state. 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's look at the definition of the first variational problem:

```
U   = 0.5*(u0 + u)
n   = FacetNormal(mesh)
f   = Constant((0, 0))
k   = Constant(dt)
mu  = Constant(mu)
rho = Constant(rho)
```

This expression for `F1` is very similar to the mathematical definition (3.32). Since the variational problem contains a mix of known and unknown quantities – the unknown u^n (which we name `u` in the variational problem) and the known value u^{n-1} (which we name `u0`) – it is convenient to use the FEniCS functions `lhs` and `rhs` to extract the left- and right-hand sides of the variational problem.

In the definition of the variational problem, 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:

```
def epsilon(u):
    return sym(nabla_grad(u))

def sigma(u, p):
    return 2*mu*epsilon(u) - p*Identity(len(u))
```

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:

```
b1 = assemble(L1)
[bc.apply(b1) for bc in bcu]
solve(A1, u1.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.

Finally, let's look at an important detail in how we use parameters such as the time step `dt` and viscosity `mu` 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)
mu = Constant(mu)
```

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 when a float value for the time step or viscosity is changed. By wrapping these parameters using `Constant`, FEniCS will treat these parameters as generic constants and not specific numerical values, 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.

hpl 22: Some pure fluid mechanics guys will think of Poiseulle as 1D, so they get confused why you need to launch 2D/3D when it's about $u'' = 4$... Renamed file to `navier_stokes_channel.py`.

The complete code for simulating 2D channel flow with FEniCS looks as follows:

```
from fenics import *
import numpy as np

T = 10.0           # final time
num_steps = 500    # number of time steps
dt = T / num_steps # time step size
mu = 1             # kinematic viscosity
rho = 1            # density

# Create mesh and define function spaces
mesh = UnitSquareMesh(16, 16)
V = VectorFunctionSpace(mesh, 'P', 2)
Q = FunctionSpace(mesh, 'P', 1)
```

```
# 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]

# 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
u0 = Function(V)
u1 = Function(V)
p0 = Function(Q)
p1 = Function(Q)

# Define expressions used in variational forms
U   = 0.5*(u0 + u)
n   = FacetNormal(mesh)
f   = Constant((0, 0))
k   = Constant(dt)
mu  = Constant(mu)
rho = Constant(rho)

# 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 - u0) / k, v)*dx + \
     rho*dot(dot(u0, nabla_grad(u0)), v)*dx \
   + inner(sigma(U, p0), epsilon(v))*dx \
   + dot(p0*n, v)*ds - dot(mu*nabla_grad(U)*n, v)*ds \
   - rho*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(p0), nabla_grad(q))*dx - (1/k)*div(u1)*q*dx

# Define variational problem for step 3
```

```
a3 = dot(u, v)*dx
L3 = dot(u1, v)*dx - k*dot(nabla_grad(p1 - p0), 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]

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

    # Update current time
    t += dt

    # Step 1: Tentative velocity step
    b1 = assemble(L1)
    [bc.apply(b1) for bc in bcu]
    solve(A1, u1.vector(), b1)

    # Step 2: Pressure correction step
    b2 = assemble(L2)
    [bc.apply(b2) for bc in bcp]
    solve(A2, p1.vector(), b2)

    # Step 3: Velocity correction step
    b3 = assemble(L3)
    solve(A3, u1.vector(), b3)

    # Plot solution
    plot(u1)

    # Compute error
    u_e = Expression(('4*x[1]*(1.0 - x[1])', '0'), degree=2)
    u_e = interpolate(u_e, V)
    error = np.abs(u_e.vector().array() - u1.vector().array()).max()
    print('t = %.2f: error = %.3g' % (t, error))
    print('max u:', u1.vector().array().max())

    # Update previous solution
    u0.assign(u1)
    p0.assign(p1)

# Hold plot
interactive()
```

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 `u0` and `u1`, to zero. Since the exact solution is quadratic, we expect the solution to be exact to within machine precision at the nodes at the final time. For our implementation, the error quickly approaches zero and is approximately 10^{-9} at final time $T = 10$.

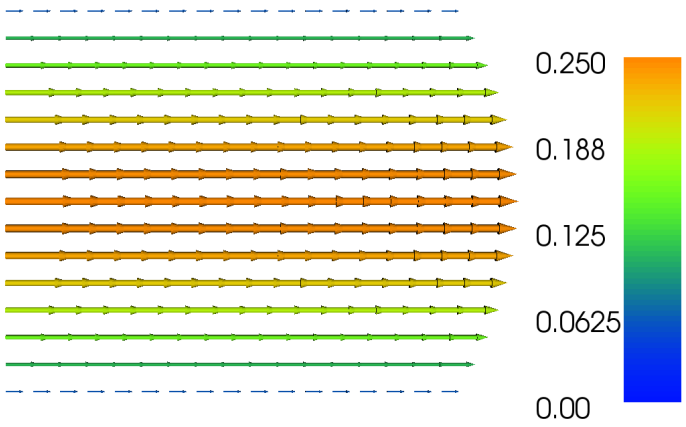


Fig. 3.2 Plot of the velocity profile at the final time for the Navier-Stokes Poiseuille flow example.

3.4.4 Flow past a cylinder

We now turn our attention to a more challenging physical example: flow past a circular cylinder. The geometry and parameters are taken from problem DFG 2D-2 in the FEATFLOW/1995-DFG benchmark suite and is illustrated in Figure 3.3. The kinematic viscosity is given by $\nu = 0.001 = \mu/\rho$ and the inflow velocity profile is specified as

$$u(x, y, t) = \left(1.5 \cdot \frac{4y(1-y)}{0.41^2}, 0\right),$$

which has a maximum magnitude of 1.5 at $y = 0.41/2$. We do not scale anything in this benchmark.

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 in FEniCS using the built-in meshes `UnitIntervalMesh` (1D), `UnitSquareMesh` (2D), `UnitCubeMesh` (3D), `IntervalMesh` (1D), `RectangleMesh` (2D), `BoxMesh` (3D), and `UnitDiscMesh` (2D). FEniCS supports the creation

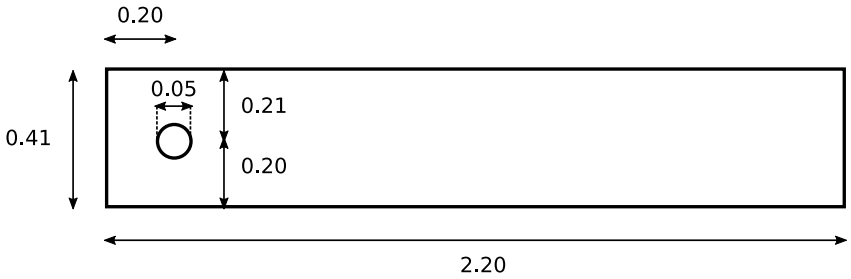


Fig. 3.3 Geometry for the flow past a cylinder test problem. Notice the slightly perturbed and unsymmetric geometry. .

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.

AL 23: Need to cite mshr.

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)
geometry = channel - cylinder
```

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

hpl 24: Should do some refinement of the boundary layer? Can we mark elements in a distance from the cylinder and ask these elements to be refine a given number of times?

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

To solve the cylinder test problem, we only need to make a few minor changes to the code we wrote for the Poiseuille 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'
```

hpl 25: I did not understand the `cylinder` line. Seems to be all points in a square that also lie on the boundary? Are there any?

AL 26: We set $p = 0$ at the outflow. This seems to be necessary, but we should really not need to specify the pressure at all. **hpl 27**: Need to specify the pressure at one point, mathematically.

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)

t = 0.0
for n in xrange(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 Poiseuille 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). This can be done by adding the keyword `bicgstab` in the call to `solve`. We also add a preconditioner, `ilu` to further speed up the computations:

```
solve(A1, u1.vector(), b1, 'bicgstab', 'ilu')
```

```
solve(A2, p1.vector(), b2, 'bicgstab', 'ilu')
solve(A3, u1.vector(), b3, 'bicgstab')
```

Finally, to be able to postprocess the computed solution in Paraview, we store the solution to file in each time step. To avoid cluttering our working directory with a large number of solution files, we make sure to store the solution in a subdirectory:

```
vtkfile_u = File('solutions/velocity.pvd')
vtkfile_p = File('solutions/pressure.pvd')
```

Note that one does not need to create the directory before running the program. It will be created automatically by FEniCS.

Figures 3.4 and 3.5 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.

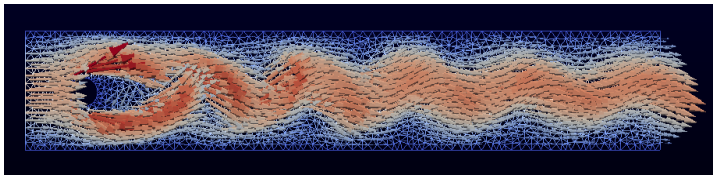


Fig. 3.4 Plot of the velocity for the cylinder test problem at final time.

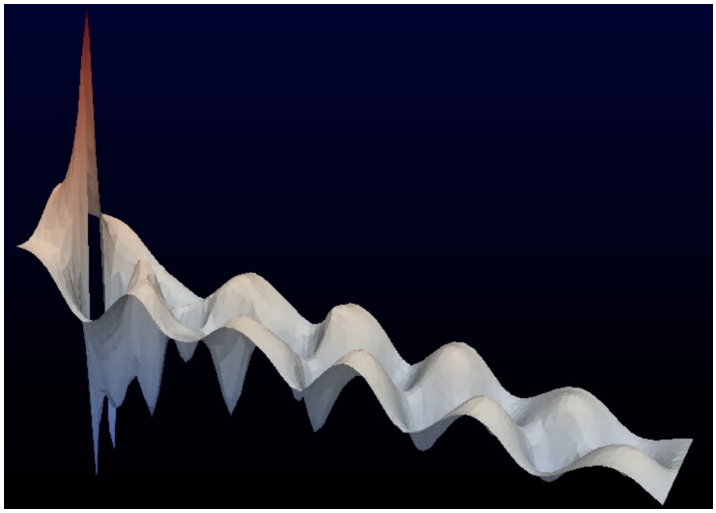


Fig. 3.5 Plot of the pressure for the cylinder test problem at final time.

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
rho = 1            # density

# Create mesh
channel = Rectangle(Point(0, 0), Point(2.2, 0.41))
cylinder = Circle(Point(0.2, 0.2), 0.05)
geometry = channel - cylinder
mesh = generate_mesh(geometry, 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
u0 = Function(V)
u1 = Function(V)
p0 = Function(Q)
p1 = Function(Q)

# Define expressions used in variational forms
U   = 0.5*(u0 + u)
n   = FacetNormal(mesh)
```

```
f   = Constant((0, 0))
k   = Constant(dt)
mu  = Constant(mu)

# 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 - u0) / k, v)*dx \
   + rho*dot(dot(u0, nabla_grad(u0)), v)*dx \
   + inner(sigma(U, p0), epsilon(v))*dx \
   + dot(p0*n, v)*ds - dot(mu*nabla_grad(U)*n, v)*ds \
   - rho*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(p0), nabla_grad(q))*dx - (1/k)*div(u1)*q*dx

# Define variational problem for step 3
a3 = dot(u, v)*dx
L3 = dot(u1, v)*dx - k*dot(nabla_grad(p1 - p0), 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 VTK files for saving solution
vtkfile_u = File('ns/velocity.pvd')
vtkfile_p = File('ns/pressure.pvd')

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

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

    # Update current time
    t += dt

    # Step 1: Tentative velocity step
```

```
    b1 = assemble(L1)
    [bc.apply(b1) for bc in bcu]
    solve(A1, u1.vector(), b1, 'bicgstab', 'ilu')

    # Step 2: Pressure correction step
    b2 = assemble(L2)
    [bc.apply(b2) for bc in bcp]
    solve(A2, p1.vector(), b2, 'bicgstab', 'ilu')

    # Step 3: Velocity correction step
    b3 = assemble(L3)
    solve(A3, u1.vector(), b3, 'bicgstab')

    # Plot solution
    plot(u1, title='Velocity')
    plot(p1, title='Pressure')

    # Save solution to file
    vtkfile_u << (u1, t)
    vtkfile_p << (p1, t)

    # Update previous solution
    u0.assign(u1)
    p0.assign(p1)

    # Update progress bar
    progress.update(t / T)
    print('u max:', u1.vector().array().max())

# Hold plot
interactive()
```

Chapter 4
Mesh generation, subdomains and boundary conditions

In this chapter, we focus on a fundamental step in the solution of many PDE problems: the generation of a mesh, and the specification of subdomains and boundary conditions. Our starting point is the 2D Poisson equation where we introduce the basic concepts, before applying them in a more challenging 3D convection-diffusion problem.

4.1 Multiple domains and boundaries

hpl 28: Need a little intro.

4.1.1 Combining Dirichlet and Neumann conditions

Let us make a slight extension of our two-dimensional Poisson problem with Dirichlet conditions on the entire boundary and add a Neumann boundary condition. The domain is still the unit square, but now we set the Dirichlet condition $u = u_{\mathrm{b}}$ 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 $y = 1$. The Neumann condition is also known as a *natural boundary condition* (in contrast to an essential boundary condition).

PDE problem. Let Γ_D and Γ_N denote the parts of $\partial\Omega$ where the Dirichlet and Neumann conditions apply, respectively. The complete boundary-value problem can be written as

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

$$u = u_\mathrm{b} \text{ on } \Gamma_D, \tag{4.2}$$

$$-\frac{\partial u}{\partial n} = g \text{ on } \Gamma_N\,. \tag{4.3}$$

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

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

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

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

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

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

and since $v = 0$ on Γ_D,

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

by applying the boundary condition on Γ_N. The resulting weak form reads

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

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

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

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

FEniCS implementation. How does the Neumann condition impact the implementation? Let us go back to the very simplest file, `ft01_poisson.py`, from Section 2.2, we realize that the statements remain almost the same. Only two adjustments are necessary:

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

The first adjustment can be coded as

```
def Dirichlet_boundary(x, on_boundary):
    if on_boundary:
        if x[0] == 0 or x[0] == 1:
            return True
        else:
            return False
    else:
        return False
```

A more compact implementation reads

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

Never use == for comparing real numbers!

A list 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

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

Comparison of real numbers need to use 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 around unity, 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 appropriate tolerance must be found.

Testing for `x[0] == 1` should therefore be implemented as

```
tol = 1E-14
if abs(x[0] - 1) < tol:
    ...
```

Here is a new boundary function using tolerances in the test:

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

This function can be written a bit more elegantly using the `near` function in FEniCS:

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

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

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

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

4.1.2 Multiple Dirichlet conditions

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

Let us return to the case from Section 4.1.1 and define two separate functions for the two Dirichlet conditions:

$$
\begin{aligned}
-\nabla^2 u &= -6 \text{ in } \Omega, \\
u &= u_L \text{ on } \Gamma_{D,0}, \\
u &= u_R \text{ on } \Gamma_{D,1}, \\
-\frac{\partial u}{\partial n} &= g \text{ on } \Gamma_N .
\end{aligned}
$$

Here, $\Gamma_{D,0}$ is the boundary $x = 0$, while $\Gamma_{D,1}$ corresponds to the boundary $x = 1$. We have that $u_L = 1 + 2y^2$, $u_R = 2 + 2y^2$, and $g = -4y$.

For the left boundary Γ_0 we define the usual triple of a function for the boundary value, a function for defining the boundary of interest, and a `DirichletBC` object:

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

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

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

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

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

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

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

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

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

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

4.1.3 Working with subdomains

Solving PDEs in domains made up of different materials is a frequently encountered task. In FEniCS, these kind of problems are handled by defining subdomains inside the domain. The subdomains may represent the various

materials. We can thereafter define material properties through functions, known in FEniCS as *mesh functions*, that are piecewise constant in each subdomain. A simple example with two materials (subdomains) in 2D will demonstrate the basic steps in the process.

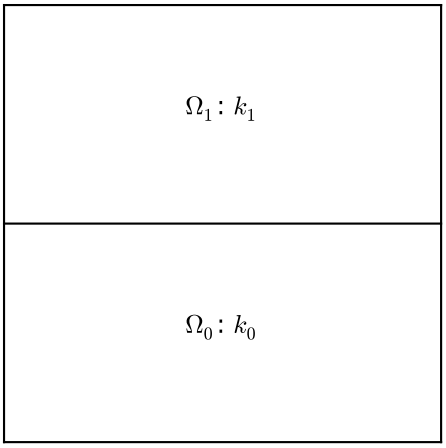


Fig. 4.1 Medium with discontinuous material properties.

Suppose we want to solve

$$\nabla \cdot [k(x,y)\nabla u(x,y)] = 0, \tag{4.7}$$

in a domain Ω consisting of two subdomains where k takes on a different value in each subdomain. For simplicity, yet without loss of generality, we choose for the current implementation the domain $\Omega = [0,1] \times [0,1]$ 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 $k(x,y) = k_0$ in Ω_0 and $k(x,y) = k_1$ in Ω_1, where $k_0 > 0$ and $k_1 > 0$ are given constants.

Physically, the present problem may correspond to heat conduction, where the heat conduction in Ω_1 is more efficient than in Ω_0. An alternative interpretation is flow in porous media with two geological layers, where the layers' ability to transport the fluid differ.

Expression objects with if test. The simplest way of implementing a variable k is to define an `Expression` object where we return the appropriate k value depending on the position in space. Since we need some testing on the coordinates, the most straightforward approach is to define a subclass of `Expression`, where we can use a full Python method instead of just a C++ string formula for specifying a function. The method that defines the function is called `eval`:

```
class K(Expression):
    def set_k_values(self, k0, k1):
        self.k0, self.k1 = k0, k1

    def eval(self, value, x):
        """x: spatial point, value[0]: function value."""
    # Fill in-place value[0] for scalar function,
    # value[:] for vector function (no return)

    tol = 1E-14  # Tolerance for coordinate comparisons
        if x[1] <= 0.5+tol:
        value[0] = self.k0
    else:
        value[0] = self.k1

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

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

```
tol = 1E-14
k0 = 1.0
k1 = 0.01
k = Expression('x[1] <= 0.5+tol? k0 : k1',
               tol=tol, k0=k0, k1=k1)
```

The method with if tests on the location is feasible when the subdomains have very simple shapes. A completely general method, utilizing *mesh functions*, is described next.

Mesh functions. We now address how to specify the subdomains Ω_0 and Ω_1 so that the method also works for subdomains of any shape. For this purpose we need to use subclasses of class `SubDomain`, not only plain functions as we have used so far for specifying boundaries. Consider the boundary function

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

for defining the boundary $x = 0$. Instead of using such a stand-alone function, we can create an instance (or object) of a subclass of `SubDomain`, which implements the `inside` method as an alternative to the `boundary` function:

```
class Boundary(SubDomain):
```

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

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

A word about computer science terminology may be used here: The term *instance* means a Python object of a particular type (such as `SubDomain`, `Function`, `FunctionSpace`, etc.). Many use *instance* and *object* as interchangeable terms. In other computer programming languages one may also use the term *variable* for the same thing. We mostly use the well-known term *object* in this text.

A subclass of `SubDomain` with an `inside` method offers functionality for marking parts of the domain or the boundary. Now we need to define one class for the subdomain Ω_0 where $y \leq 1/2$ and another for the subdomain Ω_1 where $y \geq 1/2$:

```
tol = 1E-14  # Tolerance for coordinate comparisons

class Omega0(SubDomain):
    def inside(self, x, on_boundary):
        return x[1] <= 0.5+tol

class Omega1(SubDomain):
    def inside(self, x, on_boundary):
        return x[1] >= 0.5-tol
```

Notice the use of `<=` and `>=` in both tests. For a cell to belong to, e.g., Ω_1, the `inside` method must return `True` for all the vertices `x` of the cell. So to make the cells at the internal boundary $y = 1/2$ belong to Ω_1, we need the test `x[1] >= 0.5`. However, because of potential rounding errors in the coordinates `x[1]`, we use a tolerance in the comparisons: `x[1] >= 0.5-tol`.

The next task is to use a *mesh function* to mark all cells in Ω_0 with the subdomain number 0 and all cells in Ω_1 with the subdomain number 1. Our convention is to number subdomains as $0, 1, 2, \ldots$.

A `MeshFunction` object is a discrete function that can be evaluated at a set of so-called *mesh entities*. Examples of mesh entities are cells, facets, and vertices. A `MeshFunction` over cells is suitable to represent subdomains (materials), while a `MeshFunction` over facets is used to represent pieces of external or internal boundaries. Mesh functions over vertices can be used to describe continuous fields. The specialized classes `CellFunction` and `FacetFunction` are used to construct mesh functions of cells and facets, respectively.

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

We start with creating a `CellFunction` whose values are non-negative integers (`'uint'`) for numbering the subdomains. The appropriate code for two subdomains then reads

```
materials = CellFunction('size_t', mesh)
# Mark subdomains with numbers 0 and 1
subdomain0 = Omega0()
subdomain0.mark(materials, 0)
subdomain1 = Omega1()
subdomain1.mark(materials, 1)

# Alternative
materials.set_all(0)
subdomain1.mark(materials, 1)
```

Calling `materials.array()` returns a `numpy` array of the subdomain values. That is, `materials.array()[i]` is the subdomain value of cell number `i`. This array is used to look up the subdomain or material number of a specific element.

We need a function `k` that is constant in each subdomain Ω_0 and Ω_1. Since we want `k` to be a finite element function, it is natural to choose a space of functions that is constant over each element. The family of discontinuous Galerkin methods, in FEniCS denoted by `'DG'`, is suitable for this purpose. Since we want functions that are piecewise constant, the value of the degree parameter is zero:

```
V0 = FunctionSpace(mesh, 'DG', 0)
k  = Function(V0)
```

To fill `k` with the right values in each element, we loop over all cells (i.e., indices in `materials.array()`), extract the corresponding subdomain number of a cell, and assign the corresponding k value to the `k.vector()` array:

```
k_values = [1.5, 50]  # values of k in the two subdomains
for cell_no in range(len(materials.array())):
    material_no = materials.array()[cell_no]
    k.vector()[cell_no] = k_values[material_no]
```

Long loops in Python are known to be slow, so for large meshes it is preferable to avoid such loops and instead use *vectorized code*. Normally this implies that the loop must be replaced by calls to functions from the `numpy` library that operate on complete arrays (in efficient C code). The functionality we want in the present case is to compute an array of the same size as `materials.array()`, but where the value `i` of an entry in `materials.array()` is replaced by `k_values[i]`. Such an operation is carried out by the `numpy` function `choose`:

```
help = numpy.asarray(materials.array(), dtype=numpy.int32)
k.vector()[:] = numpy.choose(help, k_values)
```

The `help` array is required since `choose` cannot work with `materials.array()` because this array has elements of type `uint32`. We must therefore transform this array to an array `help` with standard `int32` integers.

The next section exemplifies a complete solver with a piecewise constant coefficient, like k, defined through `SubDomain` objects, combined with different types of boundary conditions.

C++ strings for subdomain definitions. The `SubDomain` class in Python is convenient, but leads to lots of function calls from C++ to Python, which are slow. In large problems, the subdomains should be defined through C++ code. This is easy to achieve using the `CompiledSubDomain` object. Consider the definition of classes `Omega0` and `Omega1` above in Python. The key strings that define these subdomain can be expressed in C++ syntax and fed to `CompiledSubDomain` as follows:

```
tol = 1E-14  # Tolerance for coordinate comparisons

subdomain0 = CompiledSubDomain(
                'x[1] <= boundary+tol', tol=1E-14, boundary=0.5)
subdomain1 = CompiledSubDomain(
                'x[1] >= boundary-tol', tol=1E-14, boundary=0.5)
```

As seen, one can have parameters in the strings and specify their values by keyword arguments. The resulting objects, `subdomain0` and `subdomain1`, can be used as ordinary `SubDomain` objects.

Compiled subdomain strings can be applied for specifying boundaries as well, e.g.,

```
y_R = CompiledSubDomain('on_boundary && near(x[1], R, eps=tol)',
                        tol=1E-14, R=2)   # y=2
```

It is 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[1], 2, 1E-14)')
```

Exercise 4.1: Efficiency of Python vs C++ expressions

Consider a cube mesh with N cells in each spatial direction. We want to define a `Function` on this mesh where the values are given by the mathematical function $f(x,y,z) = a\sin(bxyz)$, where a and b are two parameters. Write a `class SineXYZ`:

```
class SineXYZ(Expression):
    def __init__(self, a, b):
        self.a, self.b = a, b
```

```
    def eval(self, value, x):
        value[0] = self.a*sin(self.b*x[0]*x[1]*x[2])
```

Create an alternative `Expression` based on giving the formula for $f(x,y,z)$ as a C++ code string. Compare the computational efficiency of the two implementations (e.g., using `time.clock()` to measure the CPU time).

The `sin` function used in class `SineXYZ.eval` can mean many things. This is an advanced FEniCS function if imported from `fenics`. Much more efficient versions for sin of numbers are found in `math.sin` and `numpy.sin`. Compare the use `sin` from `fenics`, `math`, `numpy`, and `sympy` (note that `sin` from `sympy` is very slow).

Solution. Here is an appropriate program:

```
from __future__ import print_function
from fenics import *
import time

def make_sine_Function(N, method):
    """Fill a Function with sin(x*y*z) values."""
    mesh = UnitCubeMesh(N, N, N)
    V = FunctionSpace(mesh, 'Lagrange', 2)

    if method.startswith('Python'):
        if method.endswith('fenics.sin'):
            # Need sin as local variable in this function
            from fenics import sin
        elif method.endswith('math.sin'):
            from math import sin
        elif method.endswith('numpy.sin'):
            from numpy import sin
        elif method.endswith('sympy.sin'):
            from sympy import sin
        else:
            raise NotImplementedError('method=%s' % method)
        print('sin:', sin, type(sin))

        class SineXYZ(Expression):
            def __init__(self, a, b):
                self.a, self.b = a, b

            def eval(self, value, x):
                value[0] = self.a*sin(self.b*x[0]*x[1]*x[2])

        expr = SineXYZ(a=1, b=2)

    elif method == 'C++':
        expr = Expression('a*sin(b*x[0]*x[1]*x[2])', a=1, b=2)

    t0 = time.clock()
    u = interpolate(expr, V)
    t1 = time.clock()
    return u, t1-t0
```

```
def main(N):
    u, cpu_py_fenics  = make_sine_Function(N, 'Python-fenics.sin')
    u, cpu_py_math    = make_sine_Function(N, 'Python-math.sin')
    u, cpu_py_numpy   = make_sine_Function(N, 'Python-numpy.sin')
    u, cpu_py_sympy   = make_sine_Function(N, 'Python-sympy.sin')
    u, cpu_cpp = make_sine_Function(N, 'C++')
    print("""DOFs: %d
Python:
fenics.sin: %.2f
math.sin:   %.2f
numpy.sin:  %.2f
sympy.sin:  %.2f
C++:        %.2f
Speed-up:   math: %.2f  sympy: %.2f""" %
          (u.function_space().dim(),
           cpu_py_fenics, cpu_py_math,
           cpu_py_numpy, cpu_py_sympy,
           cpu_cpp,
           cpu_py_math/float(cpu_cpp),
           cpu_py_sympy/float(cpu_cpp)))

def profile():
    import cProfile
    prof = cProfile.Profile()
    prof.runcall(main)
    prof.dump_stats("tmp.profile")
    # http://docs.python.org/2/library/profile.html

main(20)
#profile()
```

Running the program shows that `sin` from `math` is the most efficient choice, but a string C++ runs 40 times faster. Note that `fenics.sin`, which is a sine function in the UFL language that can work with symbolic expressions in finite element forms, is (naturally) less efficient than the `sin` functions for numbers in `math` and `numpy`.
Filename: `Expression_efficiency`.

4.1.4 Multiple Neumann, Robin, and Dirichlet conditions

Consider the model problem from Section 4.1.2 where we had both Dirichlet and Neumann conditions. The term `v*g*ds` in the expression for `L` implies a boundary integral over the complete boundary, or in FEniCS terms, an integral over all exterior facets. However, the contributions from the parts of the boundary where we have Dirichlet conditions are erased when the linear system is modified by the Dirichlet conditions. We would like, from an efficiency

point of view, to integrate `v*g*ds` only over the parts of the boundary where we actually have Neumann conditions. And more importantly, in other problems one may have different Neumann conditions or other conditions like the Robin type condition. With the mesh function concept we can mark different parts of the boundary and integrate over specific parts. The same concept can also be used to treat multiple Dirichlet conditions. The forthcoming text illustrates how this is done.

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 $\Gamma_{D,0}$, $\Gamma_{D,1}$, ..., of the boundary:

$$u_{0,0} \text{ on } \Gamma_{D,0}, \quad u_{0,1} \text{ on } \Gamma_{D,1}, \ldots$$

where $u_{0,i}$ are prescribed functions, $i = 0, 1, \ldots$ On other parts, $\Gamma_{N,0}$, $\Gamma_{N,1}$, and so on, we have Neumann conditions

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

Finally, we have *Robin conditions*

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

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 $\Gamma_{R,0}$, $\Gamma_{R,1}$, and so forth:

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

A general model problem. With the notation above, the model problem to be solved with multiple Dirichlet, Neumann, and Robin conditions can formally be defined as

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

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

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

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

Variational formulation. Integration by parts of $-\int_\Omega v\nabla \cdot (p\nabla u)\,\mathrm{d}x$ becomes as usual

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

The boundary integral does not apply to the parts of the boundary where we have Dirichlet conditions ($\Gamma_{D,i}$). Moreover, on the remaining parts, we must split the boundary integral into the parts where we have Neumann and Robin conditions such that we insert the right conditions as integrands. Specifically, we have

$$\begin{aligned}-\int_{\partial\Omega} p\frac{\partial u}{\partial n}v\,\mathrm{d}s &= -\sum_i\int_{\Gamma_{N,i}} p\frac{\partial u}{\partial n}\,\mathrm{d}s - \sum_i\int_{\Gamma_{R,i}} p\frac{\partial u}{\partial n}\,\mathrm{d}s\\ &= \sum_i\int_{\Gamma_{N,i}} g_i\,\mathrm{d}s + \sum_i\int_{\Gamma_{R,i}} r_i(u-s_i)\,\mathrm{d}s\,.\end{aligned}$$

The variational formulation then becomes

$$F = \int_\Omega p\nabla u\cdot\nabla v\,\mathrm{d}x + \sum_i\int_{\Gamma_{N,i}} g_i v\,\mathrm{d}s + \sum_i\int_{\Gamma_{R,i}} r_i(u-s_i)v\,\mathrm{d}s - \int_\Omega fv\,\mathrm{d}x = 0\,. \tag{4.12}$$

We have been used to writing this variational formulation in the standard notation $a(u,v) = L(v)$, which requires that we identify all integrals with *both* u and v, and collect these in $a(u,v)$, while the remaining integrals with v and not u go into $L(v)$. The integral from the Robin condition must of this reason be split in two parts:

$$\int_{\Gamma_{R,i}} r_i(u-s_i)v\,\mathrm{d}s = \int_{\Gamma_{R,i}} r_i uv\,\mathrm{d}s - \int_{\Gamma_{R,i}} r_i s_i v\,\mathrm{d}s\,.$$

We then have

$$a(u,v) = \int_\Omega p\nabla u\cdot\nabla v\,\mathrm{d}x + \sum_i\int_{\Gamma_{R,i}} r_i uv\,\mathrm{d}s, \tag{4.13}$$

$$L(v) = \int_\Omega fv\,\mathrm{d}x - \sum_i\int_{\Gamma_{N,i}} g_i v\,\mathrm{d}s + \sum_i\int_{\Gamma_{R,i}} r_i s_i v\,\mathrm{d}s\,. \tag{4.14}$$

4.1.5 FEniCS implementation of boundary conditions

Looking at our previous `solver` functions for solving the 2D Poisson equation, the following new aspects must be taken care of:

1. definition of a mesh function over the boundary,

2. marking each side as a subdomain, using the mesh function,
3. splitting a boundary integral into parts.

A general approach to the first task is to mark each of the desired boundaries 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 marking of boundaries makes use of a mesh function object, but contrary to Section 4.1.3, this is not a function over cells, but a function over cell facets. We apply the `FacetFunction` for this purpose:

```
boundary_parts = FacetFunction('size_t', mesh)
```

As in Section 4.1.3 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 $x = 0$ boundary can be marked by

```
class BoundaryX0(SubDomain):
    def inside(self, x, on_boundary):
        return on_boundary and abs(x[0]) < tol

bx0 = BoundaryX0()
bx0.mark(boundary_parts, 0)
```

Similarly, we make the classes `BoundaryX1` for the $x = 1$ boundary, `BoundaryY0` for the $y = 0$ boundary, and `BoundaryY1` for the $y = 1$ 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_b},
  1: {'Robin': (r, s)},
  2: {'Neumann: g}},
  3: {'Neumann', 0}}
```

specifies

- a Dirichlet condition, with values implemented by an `Expression` or `Constant` object `u_b`, on subdomain 0, i.e., the $x = 1$ boundary;
- a Robin condition (4.1.4) on subdomain 1, $x = 1$, with `Expression` or `Constant` objects `r` and `s` specifying r and s;
- a Neumann condition $\partial u/\partial n = g$ on subdomain 2, $y = 0$, where an `Expression` or `Constant` object `g` implements the value g;
- a homogeneous Neumann condition $\partial u/\partial n = 0$ on subdomain 3, $y = 1$.

As explained in Section 4.1.2, 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 snippet:

```
bcs = []  # List of Dirichlet conditions
for n in boundary_conditions:
    if 'Dirichlet' in boundary_conditions[n]:
        bcs.append(
            DirichletBC(V, boundary_conditions[n]['Dirichlet'],
                        boundary_parts, n))
```

The new aspect of the variational problem is the two distinct boundary integrals over $\Gamma_{N,i}$ and $\Gamma_{R,i}$. Having a mesh function over exterior cell facets (our `boundary_parts` object), where subdomains (boundary parts) are numbered as $0, 1, 2, \ldots$, the special symbol `ds(0)` implies integration over subdomain (part) 0, `ds(1)` denotes integration over subdomain (part) 1, and so on. The idea of multiple `ds`-type objects generalizes to volume integrals too: `dx(0)`, `dx(1)`, etc., are used to integrate over subdomain 0, 1, etc., inside Ω.

Before we have `ds(n)` for integers `n` defined, we must do

```
ds = Measure('ds', domain=mesh, subdomain_data=boundaries_parts)
```

Similarly, if we want integration of different parts of the domain, we redefine `dx` as

```
dx = Measure('dx', domain=mesh, subdomain_data=domains)
```

where `domains` is a `CellFunction` defining subdomains in Ω.

Suppose we have a Robin condition with values `r` and `s` on subdomain `R`, a Neumann condition with value `g` on subdomain `N`, the variational form can be written

```
a = 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 code

```
u = TrialFunction(V)
v = TestFunction(V)
Neumann_integrals = []
for n in boundary_conditions:
    if 'Neumann' in boundary_conditions[n]:
        if boundary_conditions[n]['Neumann'] != 0:
            g = boundary_conditions[n]['Neumann']
            Neumann_integrals.append(g*v*ds(n))
```

Applying `sum(Nemann_integrals)` will apply the `+` operator to the variational forms in the `Numeann_integrals` 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:

```
Robin_a_integrals = []
Robin_L_integrals = []
for n in boundary_conditions:
    if 'Robin' in boundary_conditions[n]:
```

```
        r, s = boundary_conditions[n]['Robin']
        Robin_a_integrals.append(r*u*v*ds(n))
        Robin_L_integrals.append(r*s*v*ds(n))
```

We are now in a position to define the `a` and `L` expressions in the variational formulation:

```
a = dot(p*grad(u), grad(v))*dx + \
    sum(Robin_a_integrals)
L = f*v*dx - sum(Neumann_integrals) + sum(Robin_L_integrals)
```

Simplified handling of the variational formulation. We carefully ordered the terms in the variational formulation above into the a and L parts. This requires a splitting of the Robin condition and makes the `a` and `L` expressions less readable (still we think understanding this splitting is key for any finite element programmer!). Fortunately, UFL allows us to specify the complete variational form (4.12) as one expression and offer tools to extract what goes into the bilinear form $a(u,v)$ and the linear form $L(v)$:

```
F = dot(p*grad(u), grad(v))*dx + \
    sum(Robin_integrals) - f*v*dx + sum(Neumann_integrals)
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(n)`:

```
Robin_integrals = []
for n in boundary_conditions:
    if 'Robin' in boundary_conditions[n]:
        r, s = boundary_conditions[n]['Robin']
        Robin_integrals.append(r*(u-s)*v*ds(n))
```

The complete code is in the `solver_bc` function in the `ft08_poisson_vc.py` file.

Test problem. Let us continue to use $u_{\mathrm{e}} = 1 + x^2 + 2y^2$ as the exact solution, and set $p = 1$ and $f = -6$ in the PDE. Our domain is the unit square, and we assign Dirichlet conditions at $x = 0$ and $x = 1$, a Neumann condition at $y = 1$, and a Robin condition at $y = 0$. With the given u_{e}, we realize that the Neumann condition is $-4y$ (which means -4 at $y = 1$), while the Robin condition can be selected in many ways. Since $\partial u/\partial n = -\partial u/\partial y = 0$ at $y = 0$, we can select $s = u$ and have r arbitrary in the Robin condition.

The boundary parts are $\Gamma_{D,0}$: $x = 0$, $\Gamma_{D,1}$: $x = 1$, $\Gamma_{R,0}$: $y = 0$, and $\Gamma_{N,0}$: $y = 1$.

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 be fed into `Expression` objects.

```
def application_bc_test():
    # Define manufactured solution in sympy and derive f, g, etc.
    import sympy as sym
    x, y = sym.symbols('x[0] x[1]')  # UFL needs x[0] for x etc.
    u = 1 + x**2 + 2*y**2
    f = -sym.diff(u, x, 2) - sym.diff(u, y, 2)  # -Laplace(u)
    f = sym.simplify(f)
    u_00 = u.subs(x, 0)  # x=0 boundary
    u_01 = u.subs(x, 1)  # x=1 boundary
    g = -sym.diff(u, y).subs(y, 1)  # x=1 boundary, du/dn=-du/dy
    r = 1000 # any function can go here
    s = u

    # Turn to C/C++ code for UFL expressions
    f = sym.printing.ccode(f)
    u_00 = sym.printing.ccode(u_00)
    u_01 = sym.printing.ccode(u_01)
    g = sym.printing.ccode(g)
    r = sym.printing.ccode(r)
    s = sym.printing.ccode(s)
    print('Test problem (C/C++):\nu = %s\nf = %s' % (u, f))
    print('u_00: %s\nu_01: %s\ng = %s\nr = %s\ns = %s' %
          (u_00, u_01, g, r, s))

    # Turn into FEniCS objects
    u_00 = Expression(u_00)
    u_01 = Expression(u_01)
    f = Expression(f)
    g = Expression(g)
    r = Expression(r)
    s = Expression(s)
    u_exact = Expression(sym.printing.ccode(u))

    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

    p = Constant(1)
    Nx = Ny = 2
    u, p = solver_bc(
        p, f, boundary_conditions, Nx, Ny, degree=1,
        linear_solver='direct',
        debug=2*Nx*Ny < 50,  # for small problems only
        )
```

This simple test problem is turned into a real unit test for different function spaces in the function `test_solver_bc`.

Debugging the setting of boundary conditions. It is easy to make mistakes when implementing a problem with many different types of boundary conditions, as in the present case. Some helpful debugging output is to run through all vertex coordinates and check if the `SubDomain.inside` method

marks the vertex as on the boundary. Another useful printout is to list which degrees of freedom that are subject to Dirichlet conditions, and for first-order Lagrange elements, add the corresponding vertex coordinate to the output.

```
if debug:
    # Print the vertices that are on the boundaries
    coor = mesh.coordinates()
    for x in coor:
        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('No of Dirichlet conditions:', len(bcs))
    d2v = dof_to_vertex_map(V)
    for bc in bcs:
        bc_dict = bc.get_boundary_values()
        for dof in bc_dict:
            print('dof %2d: u=%g' % (dof, bc_dict[dof]))
            if V.ufl_element().degree() == 1:
                print('   at point %s' %
                      (str(tuple(coor[d2v[dof]].tolist()))))
```

In addition, it is helpful to print the exact and the numerical solution at all the vertices as shown in Section 5.1.3.

4.1.6 FEniCS implementation of multiple subdomains

Section 4.1.3 explains how to deal with multiple subdomains of Ω and a piecewise constant coefficient function p that takes on different constant values in the different subdomains. We can easily add this type of p coefficient to the `solver_bc` function. The signature of the function is

```
def solver_bc(
    p, f,                   # Coefficients in the PDE
    boundary_conditions,    # Dict of boundary conditions
    Nx, Ny,                 # Cell division of the domain
    degree=1,               # Polynomial degree
    subdomains=[],          # List of SubDomain objects in domain
    linear_solver='Krylov', # Alt: 'direct'
    abs_tol=1E-5,           # Absolute tolerance in Krylov solver
    rel_tol=1E-3,           # Relative tolerance in Krylov solver
    max_iter=1000,          # Max no of iterations in Krylov solver
    log_level=PROGRESS,     # Amount of solver output
    dump_parameters=False,  # Write out parameter database?
    debug=False,
    ):
...
    return u, p   # p may be modified
```

If `subdomain` is an empty list, we assume there are no subdomains, and p is an `Expression` or `Constant` object specifying a formula for p. If not, `subdomain` is a list of `SubDomain` objects, defining different parts of the domain. The first element is a dummy object, defining "the rest" of the domain. The next elements define specific geometries in the `inside` methods. We start by marking all elements with subdomain number 0, this will then be "the rest" after marking subdomains 1, 2, and so on. The next step is to define `p` as a piecewise constant function over cells and fill it with values. We assume that the user-argument `p` is an array (or list) holding the values of p in the different parts corresponding to `subdomains`. The returned `p` is needed for flux computations. If there are no subdomains, the returned `p` is just the original `p` argument.

The appropriate code for computing `p` becomes

```
import numpy as np
if subdomains:
    # subdomains is list of SubDomain objects,
    # p is array of corresponding constant values of p
    # in each subdomain
    materials = CellFunction('size_t', mesh)
    materials.set_all(0)  # "the rest"
    for m, subdomain in enumerate(subdomains[1:], 1):
        subdomain.mark(materials, m)

    p_values = p
    V0 = FunctionSpace(mesh, 'DG', 0)
    p  = Function(V0)
    help = np.asarray(materials.array(), dtype=np.int32)
    p.vector()[:] = np.choose(help, p_values)
```

We define $p(x,y) = p_0$ in Ω_0 and $k(x,y) = p_1$ in Ω_1, where $p_0 > 0$ and $p_1 > 0$ are given constants. As boundary conditions, we choose $u = 0$ at $y = 0$, $u = 1$ at $y = 1$, and $\partial u/\partial n = 0$ at $x = 0$ and $x = 1$. One can show that the exact solution is now given by

$$u(x,y) = \begin{cases} \frac{2yp_1}{p_0+p_1}, & y \leq 1/2 \\ \frac{(2y-1)p_0+p_1}{p_0+p_1}, & y \geq 1/2 \end{cases} \tag{4.15}$$

As long as the element boundaries coincide with the internal boundary $y = 1/2$, this piecewise linear solution should be exactly recovered by Lagrange elements of any degree. We can use this property to verify the implementation and make a unit test for a series of function spaces:

```
def test_solvers_bc_2mat():
    tol = 2E-13  # Tolerance for comparisons

    class Omega0(SubDomain):
        def inside(self, x, on_boundary):
            return x[1] <= 0.5+tol
```

```
    class Omega1(SubDomain):
        def inside(self, x, on_boundary):
            return x[1] >= 0.5-tol

    subdomains = [Omega0(), Omega1()]
    p_values = [2.0, 13.0]
    boundary_conditions = {
        0: {'Neumann': 0},
        1: {'Neumann': 0},
        2: {'Dirichlet': Constant(0)}, # y=0
        3: {'Dirichlet': Constant(1)}, # y=1
        }

    f = Constant(0)
    u_exact = Expression(
        'x[1] <= 0.5? 2*x[1]*p_1/(p_0+p_1) : '
        '((2*x[1]-1)*p_0 + p_1)/(p_0+p_1)',
        p_0=p_values[0], p_1=p_values[1])

    for Nx, Ny in [(2,2), (2,4), (8,4)]:
        for degree in 1, 2, 3:
            u, p = solver_bc(
                p_values, f, boundary_conditions, Nx, Ny, degree,
                linear_solver='direct', subdomains=subdomains,
                debug=False)

            # Compute max error in infinity norm
            u_e = interpolate(u_exact, u.function_space())
            import numpy as np
            max_error = np.abs(u_e.vector().array() -
                               u.vector().array()).max()
            assert max_error < tol, 'max error: %g' % max_error
```

4.2 Heat loss in a pipe

We will simulate the transport of heat in a non-perfectly insulated pipe via both diffusion and convection (transport). The inner diameter of the pipe is 4 cm, its thickness is 4mm, the thickness of the surrounding insulation is 10 mm, and its length is 50 cm. We assume that the temperature of the water at the inlet is 42 degrees centigrade and the temperature of the surrounding air is 22 degrees centigrade. We also assume that the pipe transmits 0.1 liters of water per second. A sketch of the pipe is given in Figure X.

AL 29: Add nice 3D graphics here...

To compute the temperature distribution in the pipe, we need to set boundary conditions, both at the inflow and outflow, as well as on the boundary of the insulating layer which is exposed to the surrounding air. We do this by assuming that the temperature at the inlet is 42 degrees, for the water as well as for the pipe and the insulating layer. We similarly assume the

temperature to be 22 degrees in all three layers at the outflow. The boundary of the insulating layer is also assumed to have the temperature 22 degrees.

hpl 30: How to you realize this setup in practice? One big water reservoir at 42 C, and then a long pipe from that reservoir surrounded by free air? Convection is very efficient compared to diffusion, so the pipe has to be very long for the water to lose heat. At the outlet out the pipe, I don't think you can assume 22 C in the water; it will flow out with little heat loss - 40 C, e.g. A possible boundary condition for the water is to say "hardly no change" in the temperature, i.e., $\partial u/\partial n = 0$. That will pick up heat loss along the pipe and be an okay approximation.

To compute the temperature distribution, we also need to know the value of the thermal conductivity λ $[\mathrm{W}\cdot\mathrm{m}^{-1}\cdot\mathrm{K}^{-1}]$ for water, pipe, and insulation. From a book of physical tables or from the internet, we find the following values that we will use for our simulation: $\lambda_{\text{water}} = 0.6\,\mathrm{W}\cdot\mathrm{m}^{-1}\cdot\mathrm{K}^{-1}$, $\lambda_{\text{pipe}} = 18\,\mathrm{W}\cdot\mathrm{m}^{-1}\cdot\mathrm{K}^{-1}$ (stainless steel), and $\lambda_{\text{insul}} = 0.035\,\mathrm{W}\cdot\mathrm{m}^{-1}\cdot\mathrm{K}^{-1}$ (styrofoam). We will also need to know the density and specific heat of water, which we set to $\rho = 1\,\mathrm{kg/dm}^3$ and $c = 4.184\,\mathrm{kJ/(kg\cdot K)}$ (one kcal per $\mathrm{kg}\cdot\mathrm{K}$), respectively. **hpl 31**: You also need $c\varrho$ for steel and the other material. The idea with scaling is that you don't need to find 3 parameters per material, all you have to do is to find the fractions. Of course, if you now you have steel and styrofoam, then that fraction requires you to look up the values.

Finally, we need to know the velocity field for the water flowing through the pipe. We know the total flow rate of water (0.1 liters per second). If the flow is laminar (Poiseuille flow), we know that the velocity profile is a quadratic function in the radius with its maximum at the center and zero velocity on the boundary. We may thus compute the velocity profile from the dimensions of the pipe. In particular, we know that the velocity profile takes the form

$$\beta(x) = (0, 0, C(a-r)^2),$$

where $r = \sqrt{x^2+y^2}$. The flow rate is then

$$Q = \int_0^a C(a-r)^2\, 2\pi r\,\mathrm{d}r = 2\pi C a^4 \int_0^1 s(1-s)^2\,\mathrm{d}s = \pi C a^4/6$$

Knowing that $Q = 0.1\,\mathrm{dm}^3/\mathrm{s}$, we can solve for C and find $C = 0.6\,\mathrm{dm}^3/(\pi a^4)$.

4.3 Mathematical problem formulation

hpl 32: I think you should start with the incompressible version. I don't remember the compressible version of the heat equation - it has more terms than you include here (e.g., pressure work $p\nabla\cdot u$). In the incompressible case,

which we deal with anyway, $c\varrho$ is outside of the derivative even with $c\varrho$ vary. Remember that they vary here between all the materials. I would also drop the f term as the scaling with and without f will often be different.

We will model the conduction of heat in the pipe using the standard convection-diffusion equation:

$$-\nabla\cdot(\lambda\nabla u)+\nabla\cdot(c\rho\beta u)=f. \tag{4.16}$$

Here, u denotes the temperature, λ is the thermal conductivity, c is the specific heat, ρ is the density, β is the velocity field, and f is the source term. Since our problem does not have a source term, we will set $f=0$.

The convection-diffusion equation is often stated in the following slightly modified form:

$$-\nabla\cdot(\lambda\nabla u)+c\rho\beta\cdot\nabla u=f.$$

This formulation is equivalent to (4.16) if the velocity field β is divergence free; that is, if $\nabla\cdot\beta=0$: $\nabla\cdot(\beta u)=(\nabla\cdot\beta)u+\beta\cdot\nabla u=\beta\cdot\nabla u$.

4.4 Scaling the equation

Before we can solve the PDE, we must first introduce dimensionless quantities since, strictly speaking, our program can not work with units, only with numbers. We let L be a reference length, let T be a reference time length, let U be a reference temperature, and let P be a reference power (energy per unit time). We then introduce the following dimensionless quantities:

$$\bar{x}=\frac{x}{L},\ \bar{y}=\frac{y}{L},\ \bar{z}=\frac{z}{L},\ \bar{u}=\frac{u}{U},\ \bar{\beta}=\frac{\beta}{LT^{-1}},\bar{f}=\frac{f}{PL^{-3}}.$$

hpl 33: You will scale β by some characteristic velocity V and avoid introducing any time here as the problem is stationary. You need to scale ϱ, c, and λ too as they vary in space. These are piecewise constant functions, so you can scale with the value in one of the materials.

The dimensionless coordinates $\bar{x}$, $\bar{y}$ and $\bar{z}$ also lead to dimensionless derivatives: $\bar{\nabla}=(\partial/\partial\bar{x},\partial/\partial\bar{y},\partial/\partial\bar{z})=L\nabla$. Inserting $u=U\bar{u}$, $\beta=L^{-1}T\bar{\beta}$, and $f=L^{-3}P\bar{f}$ into the convection-diffusion equation (4.16) and using $\nabla=L^{-1}\bar{\nabla}$, we obtain

$$-L^{-1}\bar{\nabla}\cdot(\lambda L^{-1}\bar{\nabla}(U\bar{u}))+L^{-1}\bar{\nabla}\cdot(c\rho LT^{-1}\bar{\beta}U\bar{u})=L^{-3}P\bar{f}.$$

Rearranging the factors L,U,T,P, we obtain

$$-\bar{\nabla}\cdot(LUP^{-1}\lambda\bar{\nabla}\bar{u})+\bar{\nabla}\cdot\left(L^3UT^{-1}P^{-1}c\rho\bar{\beta}\bar{u}\right)=\bar{f}.$$

Finally, we identify the two dimensionless parameters $\bar{\lambda}$ and $\bar{c}$ given by

$$\bar{\lambda} = LUP^{-1}\lambda, \quad \bar{c} = L^3UT^{-1}P^{-1}c\rho.$$

hpl 34: You will typically get some Peclet number here as the only dimensionless parameter, and then you will have scaled ϱ, c, and λ with fractions of material parameters in different materials. I can do this :-)

Let's double-check that these are indeed dimensionless quantities. We have

$$\begin{aligned}[\bar{\lambda}] &= [LUP^{-1}\lambda] = [LUP^{-1}]\cdot[\lambda] = [LUP^{-1}]\cdot[\mathrm{W}\cdot\mathrm{m}^{-1}\cdot\mathrm{K}^{-1}] \\ &= [LUP^{-1}]\cdot[L^{-1}U^{-1}P] = [LUP^{-1}\cdot L^{-1}U^{-1}P] = [1].\end{aligned}$$

Similarly, we have

$$\begin{aligned}[L^3UT^{-1}P^{-1}c\rho] &= [L^3UT^{-1}P^{-1}]\cdot[c]\cdot[\rho] \\ &= [L^3UT^{-1}P^{-1}]\cdot[\mathrm{kJ}/(\mathrm{kg}\cdot\mathrm{K})]\cdot[\mathrm{kg}/\mathrm{dm}^3] \\ &= [L^3UT^{-1}P^{-1}]\cdot[PTU^{-1}\cdot L^{-3}] = [1].\end{aligned}$$

We thus obtain the dimensionless and fully scaled convection-diffusion equation

$$-\bar{\nabla}\cdot(\bar{\lambda}\bar{\nabla}\bar{u}) + \bar{\nabla}\cdot\left(c\bar{\beta}\bar{u}\right) = \bar{f}.$$

The reference quantities L, T, U and P may be chosen arbitrarily. **hpl 35**: Normally, these are implied by the problem setting and the desire to have all variables of unit size. L can be the length of the pipe or the diameter (I would choose the diameter). U is given from the boundary conditions (here 42 C is natural), V is given by Q. If working with very large or very small quantities, one may want to choose for example L to obtain a domain of unit size. However, for our problem this is not necessary. We will therefore make the most straightforward choice which is to use standard SI units. We thus take $L = 1\,\mathrm{m}$, $T = 1\,\mathrm{s}$, $U = 1\,\mathrm{K}$, and $P = 1\,\mathrm{W}$. The scaled variables and parameters $\bar{u}$, $\bar{\beta}$, $\bar{\lambda}$ and so on may then be easily computed by expressing everything in standard SI units and then simply dropping the units.

In the following we will simply write u in place of $\bar{u}$ but remember that the u we compute is actually $\bar{u}$ and the actual temperature will be $u\,\mathrm{K}$:

$$-\nabla\cdot(\lambda\nabla u) + \nabla\cdot(c\beta u) = f. \tag{4.17}$$

hpl 36: I would use the incompressible version of the heat equation everywhere; then we know it is correct. Furthermore, since only derivatives of u appear in the equation, adding a constant offset to u does not change the equation. We may therefore work in Kelvin (K) as well as in degrees centigrade without needing to rescale the equation.

4.5 Finite element variational formulation

The finite element variational formulation of the convection diffusion equation is obtained in the same way as for the Poisson equation in the previous chapter, by multiplying the equation with a test function v, integrating over the domain Ω, and integrating terms with second-derivatives by parts. For the (scaled) convection-diffusion equation (4.17), we have one such term, namely $-\nabla\cdot(\lambda u)$. For this term, integration by parts gives

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

Since we assume Dirichlet boundary conditions on the entire boundary, we take the test function v to be zero on $\partial\Omega$ and thus obtain the following variational problem: find $u\in V$ such that

$$\int_\Omega \lambda\nabla u\cdot\nabla v\,\mathrm{d}x + \int_\Omega \nabla\cdot(c\beta u)\,v\,\mathrm{d}x = \int_\Omega f v\,\mathrm{d}x,$$

for all test functions $v\in V$.

The variational problem can be stated in FEniCS as follows:

```
a = lmdba*dot(grad(u), grad(v))*dx + div(c*beta*u)*v*dx
L = f*v*dx
```

Note the intentional misspelling of `lmbda` to avoid a name clash with the built-in Python keyword `lmbda`.

4.6 Mesh generation

4.7 Subdomain markers

4.8 The complete program

Here is the code:

```
from fenics import *
from mshr import *

# Parameters for geometry
a = 0.04
b = a + 0.004
c = a + 0.01
L = 0.5
```

```
# Define cylinders
cylinder_a = Cylinder(Point(0, 0, 0), Point(0, 0, L), a, a)
cylinder_b = Cylinder(Point(0, 0, 0), Point(0, 0, L), b, b)
cylinder_c = Cylinder(Point(0, 0, 0), Point(0, 0, L), c, c)

# Define domain and set subdomains
domain = cylinder_c
domain.set_subdomain(1, cylinder_b)
domain.set_subdomain(2, cylinder_a)

# Generate mesh
mesh = generate_mesh(domain, 16)

xmlfile = File('pipe.xml')
xmlfile << mesh

vtkfile = File('pipe.pvd')
vtkfile << mesh
```

Chapter 5
Common extensions in PDE solvers

hpl 37: I don't like this title, but have no other good alternative... **AL 38**: Experimenting with new title **hpl 39**: More additional experiments...

This chapter goes through common improvements of the codes presented in the previous chapter. In particular, we show how to

- write general solver functions
- utilize iterative solvers with preconditioners for solving linear systems
- compute derived quantities (e.g., flux at a part of the boundary)
- specify subdomains and parts of the boundary

5.1 Refactored implementation

Our first programs in this book are all "flat". That is, they are not organized into logical, reusable units in terms of Python functions. Such flat programs are popular for quickly testing out some software, but not well suited for serious problem solving. We shall therefore at once *refactor* the program, meaning that we divide it into functions, but this is just a reordering of the existing statements. During refactoring, we try make functions as reusable as possible in other contexts, but statements specific to a certain problem or task are also encapsulated in (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) is often mixed together.

5.1.1 A more general solver function

We consider the flat program developed in Section 2.2. 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 $u = u_{\mathrm{b}}$ on the boundary, while other statements arise from our simple test problem. Let us collect the general, reusable code in a function `solver`. Our special test problem will then just be an application of `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_{b}, 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 let also the degree of the polynomials in the finite element basis functions be an argument to `solver`.

```
from fenics import *

def solver(f, u_b, 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_b (Expresssion) on
    the boundary.
    """
    # Create mesh and define function space
    mesh = UnitSquareMesh(Nx, Ny)
    V = FunctionSpace(mesh, 'P', degree)

    def boundary(x, on_boundary):
        return on_boundary

    bc = DirichletBC(V, u_b, 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
```

Plotting for the test problem. The additional tasks we did in our initial program can be placed in other functions. For example, plotting the solution in our particular test problem is placed in an `application_test` function:

```
def application_test():
    """Plot the solution in the test problem."""
    u_b = Expression('1 + x[0]*x[0] + 2*x[1]*x[1]')
```

```
    f = Constant(-6.0)
    u = solver(f, u_b, 6, 4, 1)
    # Dump solution to file in VTK format
    u.rename('u', 'u')  # name 'u' is used in plot
    vtkfile = File("poisson.pvd")
    vtkfile << u
    # Plot solution and mesh
    plot(u)
```

Make a module! The refactored code is put in a file `ft06_poisson_func.py`. We should make sure that such a file can be imported (and hence reused) in other programs. Then all statements in the main program should appear with 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 `ft06_poisson_func.py`, the main program is simply a call to `application_test()` followed by a call `interactive()` to hold the plot:

```
if __name__ == '__main__':
    application_test()
    # Hold plot
    interactive()
```

5.1.2 Verification and unit tests

The remaining part of our first program is to compare the numerical and the exact solution. Every time we edit the code we must rerun the test and examine that `max_error` 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 tool `unittest`, but here we are going to use pytest (or nose) since it demands shorter and clearer code.

Mathematically, our unit test is that the finite element solution of our problem when $f = -6$ equals the exact solution $u = u_{\mathrm{b}} = 1 + x^2 + 2y^2$. We have already created code that finds the maximum error in the numerical solution. Because of rounding errors, we cannot demand this maximum error to be zero, but we have to use a tolerance, which depends to the number of elements and the degrees of the polynomials in the finite element basis functions. If we want to test that `solver` works for meshes up to $2(20 \times 20)$ elements and cubic Lagrange elements, 10^{-11} is an appropriate tolerance for testing that the maximum error vanishes (see Section 2.3).

Only three statements are necessary to carry out the unit test. However, we shall embed these statements in software that the testing frameworks pytest and nose can recognize. This means that each unit test must be placed in a function that

- has a name starting with `test_`
- has no arguments
- implements the test 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, otherwise the statement runs silently. The `msg` string is optional, so `assert success` is the minimal test. In our case, we will do `assert max_error < tol`, where `tol` is the tolerance (10^{-11}) 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():
    """Reproduce u=1+x^2+2y^2 to "machine precision"."""
    tol = 1E-11  # This problem's precision
    u_b = Expression('1 + x[0]*x[0] + 2*x[1]*x[1]')
    f = Constant(-6.0)
    for Nx, Ny in [(3,3), (3,5), (5,3), (20,20)]:
        for degree in 1, 2, 3:
            print('solving on 2(%dx%d) mesh with P%d elements'
                  % (Nx, Ny, degree))
            u = solver(f, u_b, Nx, Ny, degree)
            # Make a finite element function of the exact u_b
            V = u.function_space()
            u_b_Function = interpolate(u_b, V)  # exact solution
            # Check that dof arrays are equal
            u_b_array = u_b_Function.vector().array()  # dof values
            max_error = (u_b_array - u.vector().array()).max()
            msg = 'max error: %g for 2(%dx%d) mesh and degree=%d' %\
                  (max_error, Nx, Ny, degree)
            assert max_error < tol, msg
```

We can at any time run

Terminal

```
Terminal> py.test -s -v ft06_poisson_func.py
```

and the pytest tool will run all functions `test_*()` in the file and report how the tests go.

We shall make it a habit in this book to encapsulate numerical test problems in unit tests as done above, and we strongly encourage the reader to create similar unit tests whenever a FEniCS solver is implemented. We dare to assert that this is the only serious way do reliable computational science with FEniCS.

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.)

5.1.3 Writing out the discrete solution

We have seen how to grab the degrees of freedom array from a finite element function `u`:

```
u_array = u.vector().array()
```

The elements in `u_array` correspond to function values of `u` at nodes in the mesh. Now, a fundamental question is: What are the coordinates of node `i` whose value is `u_array[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. We start with P1 (1st order Lagrange) elements where all the nodes are 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)
>>> coor = mesh.coordinates()
>>> coor
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. ]])
```

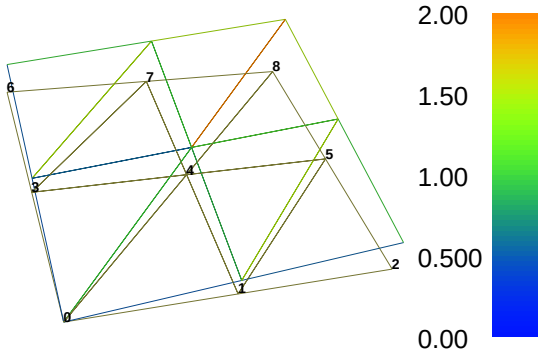
We see from this output that vertices are first numbered along $y = 0$ with increasing x coordinate, then along $y = 0.5$, and so on.

Next we compute a function `u` on this mesh, e.g., the $u = x + y$:

```
>>> V = FunctionSpace(mesh, 'P', 1)
>>> u = interpolate(Expression('x[0]+x[1]'), V)
>>> plot(u, interactive=True)
>>> u_array = u.vector().array()
>>> u_array
array([ 1. ,  0.5,  1.5,  0. ,  1. ,  2. ,  0.5,  1.5,  1. ])
```

We observe that `u_array[0]` is *not* the value of $x+y$ at vertex number 0, since this vertex has coordinates $x=y=0$. The numbering of the degrees of freedom $U_1,\ldots,U_N$ is obviously not the same as the numbering of the vertices.

In the plot of `u`, type `w` to turn on wireframe instead of fully colored surface, `m` to show the mesh, and then `v` to show the numbering of the vertices.



Already in Section 2.3 we explained that the vertex values of a `Function` object can be extracted `u.compute_vertex_values()`, which returns an array where element `i` is the value of `u` at vertex `i`:

```
>>> u_at_vertices = u.compute_vertex_values()
>>> for i, x in enumerate(coor):
...     print('vertex %d: u_at_vertices[%d]=%g\tu(%s)=%g' %
...           (i, i, u_at_vertices[i], x, u(x)))
vertex 0: u_at_vertices[0]=0    u([ 0.  0.])=8.46545e-16
vertex 1: u_at_vertices[1]=0.5  u([ 0.5  0. ])=0.5
vertex 2: u_at_vertices[2]=1    u([ 1.  0.])=1
vertex 3: u_at_vertices[3]=0.5  u([ 0.   0.5])=0.5
vertex 4: u_at_vertices[4]=1    u([ 0.5  0.5])=1
vertex 5: u_at_vertices[5]=1.5  u([ 1.   0.5])=1.5
vertex 6: u_at_vertices[6]=1    u([ 0.  1.])=1
vertex 7: u_at_vertices[7]=1.5  u([ 0.5  1. ])=1.5
vertex 8: u_at_vertices[8]=2    u([ 1.  1.])=2
```

Alternatively, we can ask for the mapping from vertex numbering to degrees of freedom numbering in the space V:

```
v2d = vertex_to_dof_map(V)
```

Now, `u_array[v2d[i]]` will give us the value of the degree of freedom in `u` corresponding to vertex `i` (`v2d[i]`). In particular, `u_array[v2d]` is an array with all the elements in the same (vertex numbered) order as `coor`. The inverse map, from degrees of freedom number to vertex number is given by `dof_to_vertex_map(V)`, so `coor[dof_to_vertex_map(V)]` results in an array of all the coordinates in the same order as the degrees of freedom.

For Lagrange elements of degree larger than 1, there are degrees of freedom (nodes) that do not correspond to vertices. **hpl 40**: Anders, is the following true? There is no simple way of getting the coordinates associated with the non-vertex degrees of freedom, so if we want to write out the values of a finite element solution, the following code snippet does the task at the vertices, and this will work for all kinds of Lagrange elements.

```
def compare_exact_and_numerical_solution(Nx, Ny, degree=1):
    u_b = Expression('1 + x[0]*x[0] + 2*x[1]*x[1]')
    f = Constant(-6.0)
    u = solver(f, u_b, Nx, Ny, degree, linear_solver='direct')
    # Grab exact and numerical solution at the vertices and compare
    V = u.function_space()
    u_b_Function = interpolate(u_b, V)
    u_b_at_vertices = u_b_Function.compute_vertex_values()
    u_at_vertices = u.compute_vertex_values()
    coor = V.mesh().coordinates()
    for i, x in enumerate(coor):
        print('vertex %2d (%9g,%9g): error=%g'
              % (i, x[0], x[1],
                 u_b_at_vertices[i] - u_at_vertices[i]))
        # Could compute u_b(x) - u_at_vertices[i] but this
        # is much more expensive and gives more rounding errors
    center = (0.5, 0.5)
    error = u_b(center) - u(center)
    print('numerical error at %s: %g' % (center, error))
```

As expected, the error is either identically zero or about 10^{-15} or 10^{-16}.

Cheap vs expensive function evaluation

Given a `Function` object `u`, we can evaluate its values 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 very expensive while the other two are very efficient (but limited to certain points).

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

```
center = (0.5, 0.5)
error = u_b(center) - u(center)
print('numerical error at %s: %g' % (center, error)
```

Trying a $2(3\times 3)$ mesh, the output from the previous snippet becomes

```
numerical 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_b` is a quadratic function. When the center point is a node, as in a $2(t\times 2)$ or $2(4\times 4)$ mesh, the error is of the order 10^{-15}.

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 = u_array.max()
    u_array /= u_max
    u.vector()[:] = u_array
    u.vector().set_local(u_array)  # alternative
    return u
```

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

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 `u.set_local(...)`.

All the code in this subsection can be found in the file `ft07_poisson_iter.py`.

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. We will conveniently make use of this feature in forthcoming examples. As an appetizer, go back to the introductory programs `ft01_poisson.py` or `ft06_poisson_func.py` and change the mesh construction from `UnitSquareMesh(6, 4)` to `UnitCubeMesh(6, 4, 5)`. Now the domain is the unit cube partitioned into $6 \times 4 \times 5$ boxes, and each box is divided into six tetrahedra-shaped finite elements for computations. Run the program and observe that we can solve a 3D problem without any other modifications (!). The visualization allows you to rotate the cube and observe the function values as colors on the boundary.

Generating a hypercube. The syntax for generating a unit interval, square, or box is different, so we need to encapsulate this part of the code. Given a list or tuple with the divisions into cells in the various spatial direction, the following function returns the mesh in a d-dimensional problem:

```
def unit_hypercube(divisions, degree):
    mesh_classes = [UnitIntervalMesh, UnitSquareMesh, UnitCubeMesh]
    d = len(divisions)
    mesh = mesh_classes[d-1](*divisions)
    V = FunctionSpace(mesh, 'P', degree)
    return V, 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 component of the list `divisions` as separate arguments. 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])
```

Replacing the `Nx` and `Ny` parameters by `divisions` and calling `unit_hypercube` to create the mesh are the two modifications that we need in any of the previously shown `solver` functions to turn them into solvers for d-dimensional problems!

Exercise 5.1: Solve a Poisson problem

Solve the following problem

$$\nabla^2 u = 2e^{-2x}\sin(\pi y)((4-5\pi^2)\sin(2\pi x) - 8\pi\cos(2\pi x)) \text{ in } \Omega = [0,1]\times[0,1] \tag{5.1}$$

$$u = 0 \quad \text{on } \partial\Omega \tag{5.2}$$

The exact solution is given by

$$u(x,y) = 2e^{-2x}\sin(\pi x)\sin(\pi y)\,.$$

Compute the maximum numerical approximation error in a mesh with $2(N_x \times N_y)$ elements and in a mesh with double resolution: $4(N_x \times N_y)$ elements. Show that the doubling the resolution reduces the error by a factor 4 when using Lagrange elements of degree one. Make an illustrative plot of the solution too.

a) Base your implementation on editing the program `ft01_poisson.py`.

Hint 1. In the string for an `Expression` object, `pi` is the value of π. Also note that π^2 must be expressed with syntax `pow(pi,2)` and not (the common Python syntax) `pi**2`.

FEniCS will abort with a compilation error if you type the expressions in a wrong way syntax-wise. Search for *error:* in the `/very/long/path/compile.log` file mentioned in the error message to see what the C++ compiler reported as error in the expressions.

Hint 2. The result that with P1 elements, doubling the resolution reduces the error with a factor of four, is an asymptotic result so it requires a sufficiently fine mesh. Here one may start with $N_x = N_y = 20$.
Filename: `poisson_fsin_flat`.

Solution. Looking at the `ft01_poisson.py` code, we realize that the following edits are required:

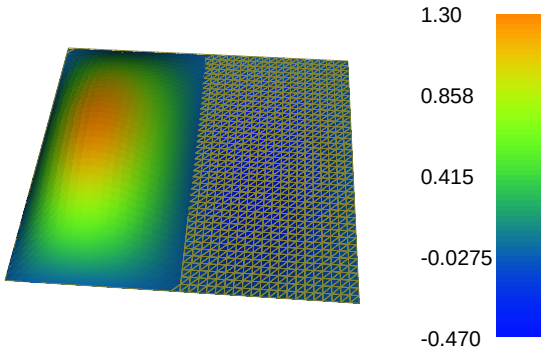
- Modify the `mesh` computation.
- Modify `u_b` and `f`.
- Add expression for the exact solution.
- Modify the computation of the numerical error.
- Insert a loop to enable solving the problem twice.
- Put the error reduction computation and the plot statements after the loop.

Here is the modified code:

```
from fenics import *

Nx = Ny = 20
error = []
for i in range(2):
    Nx *= (i+1)
```

```
    Ny *= (i+1)

    # Create mesh and define function space
    mesh = UnitSquareMesh(Nx, Ny)
    V = FunctionSpace(mesh, 'Lagrange', 1)

    # Define boundary conditions
    u0 = Constant(0)

    def u0_boundary(x, on_boundary):
        return on_boundary

    bc = DirichletBC(V, u0, u0_boundary)

    # Define variational problem
    u = TrialFunction(V)
    v = TestFunction(V)
    f = Expression('-2*exp(-2*x[0])*sin(pi*x[1])*('
                   '(4-5*pow(pi,2))*sin(2*pi*x[0]) '
                   ' - 8*pi*cos(2*pi*x[0]))')
    # Note: no need for pi=DOLFIN_PI in f, pi is valid variable
    a = inner(nabla_grad(u), nabla_grad(v))*dx
    L = f*v*dx

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

    u_e = Expression(
        '2*exp(-2*x[0])*sin(2*pi*x[0])*sin(pi*x[1])')

    u_e_Function = interpolate(u_e, V)         # exact solution
    u_e_array = u_e_Function.vector().array()  # dof values
    max_error = (u_e_array - u.vector().array()).max()
    print('max error:', max_error, '%dx%d mesh' % (Nx, Ny))
    error.append(max_error)

print('Error reduction:', error[1]/error[0])

# Plot solution and mesh
plot(u)

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

# Hold plot
interactive()
```

The number π has the symbol `M_PI` in C and C++, but in C++ strings in `Expression` objects, the symbol `pi` can be used directly (or one can use the less readable `DOLFIN_PI`).



b) Base your implementation on a new file that imports functionality from the module `ft06_poisson_func.py`. Embed the check of the reduction of the numerical approximation error in a unit test. Filename: `poisson_fsin_func`.

Solution. Solving the two problems is a matter of calling `solver` with different sets of arguments. To compute the numerical error, we need code that is close to what we have in `test_solver`.

```
from poisson_func import (
    solver, Expression, Constant, interpolate, File, plot,
    interactive)

def data():
    """Return data for this Poisson problem."""
    u0 = Constant(0)
    u_e = Expression(
        '2*exp(-2*x[0])*sin(2*pi*x[0])*sin(pi*x[1])')
    f = Expression('-2*exp(-2*x[0])*sin(pi*x[1])*('
                   '(4-5*pow(pi,2))*sin(2*pi*x[0]) '
                   ' - 8*pi*cos(2*pi*x[0]))')
    return u0, f, u_e

def test_solver():
    """Check convergence rate of solver."""
    u0, f, u_e = data()
    Nx = 20
    Ny = Nx
    error = []
    # Loop over refined meshes
    for i in range(2):
        Nx *= i+1
        Ny *= i+1
        print('solving on 2(%dx%d) mesh' % (Nx, Ny))
        u = solver(f, u0, Nx, Ny, degree=1)
        # Make a finite element function of the exact u_e
```

```
        V = u.function_space()
        u_e_array = interpolate(u_e, V).vector().array()
        max_error = (u_e_array - u.vector().array()).max()  # Linf norm
        error.append(max_error)
        print('max error:', max_error)
    for i in range(1, len(error)):
        error_reduction = error[i]/error[i-1]
        print('error reduction:', error_reduction)
        assert abs(error_reduction - 0.25) < 0.1

def application():
    """Plot the solution."""
    u0, f, u_e = data()
    Nx = 40
    Ny = Nx
    u = solver(f, u0, Nx, Ny, 1)
    # Dump solution to file in VTK format
    file = File("poisson.pvd")
    file << u
    # Plot solution and mesh
    plot(u)

if __name__ == '__main__':
    test_solver()
    application()
    # Hold plot
    interactive()
```

The unit test is embedded in a proper test function `test_solver` for the pytest or nose testing frameworks. Visualization of the solution is encapsulated in the `application` function. Since we need `u_e`, `u_b`, and `f` in two functions, we place the definitions in a function `data` to avoid copies of these expressions.

Remarks. This exercise demonstrates that changing a flat program to solve a new problem requires careful editing of statements scattered around in the file, while the solution in b), based on the `solver` function, requires *no modifications* of the `ft06_poisson_func.py` file, just *minimalistic additional new code* in a separate file. The Poisson solver remains in one place (`ft06_poisson_func.py`) while in a) we got two Poisson solvers. If you decide to switch to an iterative solution method for linear systems, you can do so in one place in b), and all applications can take advantage of the extension. Hopefully, with this exercise you realize that embedding PDE solvers in functions (or classes) makes more reusable software than flat programs.

Exercise 5.2: Refactor the code for membrane deflection

The `ft02_membrane.py` program simulates the deflection of a membrane. Refactor this code such that we have a `solver` function as in the `ft06_poisson_func.py`

file. Let the user have the option to choose a direct or iterative solver for the linear system. Also implement a unit test where you have $p = 4$ (constant) and use P2 and P3 elements. In this case, the exact solution is quadratic in x and y and will be "exactly" reproduced by P2 and higher-order elements.

Solution. We can use the `solver` function from `ft06_poisson_func.py` right away. The major difference is that the domain is now a circle and not a square. We change the `solver` function by letting the mesh be an argument `mesh` (instead of `Nx` and `Ny`):

```
def solver(
    f, u_b, mesh, degree=1,
    linear_solver='Krylov', # Alt: 'direct'
    ...):
    V = FunctionSpace(mesh, 'P', degree)
    # code as before
```

The complete code becomes

```
def application(beta, R0, num_elements_radial_dir):
    # Scaled pressure function
    p = Expression(
        '4*exp(-pow(beta,2)*(pow(x[0], 2) + pow(x[1]-R0, 2)))',
        beta=beta, R0=R0)

    # Generate mesh over the unit circle
    domain = Circle(Point(0.0, 0.0), 1.0)
    mesh = generate_mesh(domain, num_elements_radial_dir)

    w = solver(p, Constant(0), mesh, degree=1,
               linear_solver='direct')
    w.rename('w', 'deflection')  # set name and label (description)

    # Plot scaled solution, mesh and pressure
    plot(mesh, title='Mesh over scaled domain')
    plot(w, title='Scaled ' + w.label())
    V = w.function_space()
    p = interpolate(p, V)
    p.rename('p', 'pressure')
    plot(p, title='Scaled ' + p.label())

    # Dump p and w to file in VTK format
    vtkfile1 = File('membrane_deflection.pvd')
    vtkfile1 << w
    vtkfile2 = File('membrane_load.pvd')
    vtkfile2 << p
```

The key function to simulate membrane deflection is named `application`.

For $p = 4$, we have $w = 1 - x^2 - y^2$ as exact solution. The unit test for P2 and P3 goes as follows:

```
def test_membrane():
    """Verification for constant pressure."""
    p = Constant(4)
```

```
    # Generate mesh over the unit circle
    domain = Circle(Point(0.0, 0.0), 1.0)
    for degree in 2, 3:
        print('********* P%d elements:' % degree)
        n = 5
        for i in range(4):  # Run some resolutions
            n *= (i+1)
            mesh = generate_mesh(domain, n)
            #info(mesh)
            w = solver(p, Constant(0), mesh, degree=degree,
                       linear_solver='direct')
            print('max w: %g, w(0,0)=%g, h=%.3E, dofs=%d' %
                  (w.vector().array().max(), w((0,0)),
                   1/np.sqrt(mesh.num_vertices()),
                   w.function_space().dim()))

            w_exact = Expression('1 - x[0]*x[0] - x[1]*x[1]')
            w_e = interpolate(w_exact, w.function_space())
            error = np.abs(w_e.vector().array() -
                           w.vector().array()).max()
            print('error: %.3E' % error)
            assert error < 9.61E-03

def application2(
    beta, R0, num_elements_radial_dir):
    """Explore more built-in visulization features."""
    # Scaled pressure function
    p = Expression(
        '4*exp(-pow(beta,2)*(pow(x[0], 2) + pow(x[1]-R0, 2)))',
        beta=beta, R0=R0)

    # Generate mesh over the unit circle
    domain = Circle(Point(0.0, 0.0), 1.0)
    mesh = generate_mesh(domain, num_elements_radial_dir)

    w = solver(p, Constant(0), mesh, degree=1,
               linear_solver='direct')
    w.rename('w', 'deflection')

    # Plot scaled solution, mesh and pressure
    plot(mesh, title='Mesh over scaled domain')
    viz_w = plot(w,
                 wireframe=False,
                 title='Scaled membrane deflection',
                 axes=False,
                 interactive=False,
                 )
    viz_w.elevate(-10) # adjust (lift) camera from default view
    viz_w.plot(w)      # bring new settings into action
    viz_w.write_png('deflection')
    viz_w.write_pdf('deflection')

    V = w.function_space()
    p = interpolate(p, V)
```

```
    p.rename('p', 'pressure')
    viz_p = plot(p, title='Scaled pressure', interactive=False)
    viz_p.elevate(-10)
    viz_p.plot(p)
    viz_p.write_png('pressure')
    viz_p.write_pdf('pressure')

    # Dump w and p to file in VTK format
    vtkfile1 = File('membrane_deflection.pvd')
    vtkfile1 << w
    vtkfile2 = File('membrane_load.pvd')
    vtkfile2 << p
```

The striking feature is that the solver does not reproduce the solution to an accuracy more than about 0.01 (!), regardless of the resolution and type of element.
Filename: `membrane_func`.

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 recommended method for a few thousand unknowns in the equation system, and may hence be the method of choice in many 2D and smaller 3D problems. However, sparse LU decomposition becomes slow and memory demanding in large problems. This fact forces the use of iterative methods, which are faster and require much less memory. The forthcoming text tells you how to take advantage of state-of-the-art iterative solution methods in FEniCS.

5.2.1 Controlling the solution process

Setting linear solver parameters. Preconditioned Krylov solvers is a type of popular iterative methods that are easily accessible in FEniCS programs. The Poisson equation results in a symmetric, positive definite coefficient matrix, for which the optimal Krylov solver is the Conjugate Gradient (CG) method. However, the CG method requires boundary conditions to be implemented in a symmetric way. This is not the case by default, so then a Krylov solver for non-symmetric system, 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 5.2.2 lists the most popular choices of Krylov solvers and preconditioners available in FEniCS.

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

```
parameters['linear_algebra_backend'] = backendname
```

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

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

```
list_linear_algebra_backends()
```

The `parameters` database. We will normally like to control the tolerance in the stopping criterion and the maximum number of iterations when running an iterative method. Such parameters can be set by accessing the *global parameter database*, which is called `parameters` and which 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 the norm of the residual, which must be smaller than the absolute tolerance parameter *or* smaller than the relative tolerance parameter times the initial residual.

To get a printout of the number of actual iterations to reach the stopping criterion, we can insert

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

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

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('fenics_parameters.xml') << parameters
```

If a `fenics_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/fenics_parameters.xml` in the user's home directory is read, if it exists. Another alternative is to load the XML (with any name) manually in the program:

```
File('fenics_parameters.xml') >> parameters
```

The XML file can also be in gzip'ed form with the extension `.xml.gz`.

An extended solver function. We may extend the previous solver function from `ft06_poisson_func.py` in Section 5.1.1 such that it also offers the GMRES+ILU preconditioned Krylov solver:

```
def solver(
    f, u_b, Nx, Ny, degree=1,
    linear_solver='Krylov', # Alt: 'direct'
    abs_tol=1E-5,           # Absolute tolerance in Krylov solver
    rel_tol=1E-3,           # Relative tolerance in Krylov solver
    max_iter=1000,          # Max no of iterations in Krylov solver
    log_level=PROGRESS,     # Amount of solver output
    dump_parameters=False,  # Write out parameter database?
    ):
    ...
    # Set up variational problem: a, L, declare u, etc.

    if linear_solver == 'Krylov':
        prm = parameters['krylov_solver'] # short form
        prm['absolute_tolerance'] = abs_tol
        prm['relative_tolerance'] = rel_tol
        prm['maximum_iterations'] = max_iter
        print(parameters['linear_algebra_backend'])
        set_log_level(log_level)
        if dump_parameters:
            info(parameters, True)
        solver_parameters = {'linear_solver': 'gmres',
```

```
                                'preconditioner': 'ilu'}
    else:
        solver_parameters = {'linear_solver': 'lu'}

    solve(a == L, u, bc, solver_parameters=solver_parameters)
    return u
```

This new `solver` function, found in the file `ft07_poisson_iter.py`, replaces the one in `ft06_poisson_func.py`: it has all the functionality of the previous `solver` function, but can also solve the linear system with iterative methods and report the progress of such solvers.

Remark regarding unit tests. Regarding verification of the new `solver` function in terms of unit tests, it turns out that unit testing in a problem where the approximation error vanishes is 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. However, as soon as we switch to P2 elements, it is hard to force the linear solver error below 10^{-6}. Consequently, tolerances in tests depend on the numerical methods. The interested reader is referred to the `test_solver` function in `ft07_poisson_iter.py` for details: this test 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.

5.2.2 Linear solvers and preconditioners

The following solution methods for linear systems can be accessed in FEniCS programs:

Name	Method
`'lu'`	sparse LU factorization (Gaussian elim.)
`'cholesky'`	sparse Cholesky factorization
`'cg'`	Conjugate gradient method
`'gmres'`	Generalized minimal residual method
`'bicgstab'`	Biconjugate gradient stabilized method
`'minres'`	Minimal residual method
`'tfqmr'`	Transpose-free quasi-minimal residual method
`'richardson'`	Richardson method

Possible choices of preconditioners include

Name	Method
'none'	No preconditioner
'ilu'	Incomplete LU factorization
'icc'	Incomplete Cholesky factorization
'jacobi'	Jacobi iteration
'bjacobi'	Block Jacobi iteration
'sor'	Successive over-relaxation
'amg'	Algebraic multigrid (BoomerAMG or ML)
'additive_schwarz'	Additive Schwarz
'hypre_amg'	Hypre algebraic multigrid (BoomerAMG)
'hypre_euclid'	Hypre parallel incomplete LU factorization
'hypre_parasails'	Hypre parallel sparse approximate inverse
'ml_amg'	ML algebraic multigrid

Many of the choices listed above are only offered by a specific backend, so setting the backend appropriately is necessary for being able to choose a desired linear solver or preconditioner. You can also use constructions like

```
prec = 'amg' if has_krylov_solver_preconditioner('amg') \
       else 'default'
```

An up-to-date list of the available solvers and preconditioners in FEniCS can be produced by

```
list_linear_solver_methods()
list_krylov_solver_preconditioners()
```

5.2.3 Linear variational problem and solver objects

The `solve(a == L, u, bc)` call is just a compact syntax alternative to a slightly more comprehensive specification of the variational equation and the solution of the associated linear system. This alternative syntax is used in a lot of FEniCS applications and will also be used later in this tutorial, so we show it already now:

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

Many objects have an attribute `parameters` corresponding to a parameter set in the global `parameters` database, but local to the object. Here, `solver.parameters` play that role. Setting the CG method with ILU preconditioning as 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 done above.

The linear variational problem and solver objects as outlined above are incorporated in an alternative solver function, named `solver_objects`, in `ft07_poisson_iter.py`. Otherwise, this function is parallel to the previously shown `solver` function.

5.2.4 Creating the linear system explicitly

hpl 41: This is already explained for Navier–Stokes now!

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

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

which is nothing but a linear system,

$$AU = b,$$

where the entries in 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 side 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-side vector b, and then solve the linear system $AU = b$ with respect to the U vector. Instead of `solve(a == L, u, b)` we now write

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

The variables `a` and `L` are as before. That is, `a` refers to the bilinear form involving a `TrialFunction` object (e.g., `u`) and a `TestFunction` object (`v`), and `L` involves a `TestFunction` object (`v`). From `a` and `L`, the `assemble` function can compute A and b.

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`, as explained in Section 4.1.2, we must apply each condition in `bcs` to the system:

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

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

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

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

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

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

```
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 a preferred tool for computing the eigenvalues of large, sparse matrices of the type encountered in PDE problems (see `demo/la/eigenvalue` in the FEniCS source code tree for a demo).

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

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

Appropriate names of solvers and preconditioners are found in Section 5.2.2.

To control tolerances in the stopping criterion and the maximum number of iterations, one can explicitly form a `KrylovSolver` object and set items in its `parameters` attribute (see also Section 5.2.3):

```
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()
set_log_level(DEBUG)
solver.solve(A, U, b)
```

The function `solver_linalg` in the program file `ft08_poisson_vc.py` implements a solver function where the user can choose between different types of assembly: the variational (`solve(a == L, u, bc)`), assembling the matrix and right-hand side separately, and assembling the system such that the coefficient matrix preserves symmetry. The function `application_linalg` runs a test problem on sequence of meshes and solves the problem with symmetric and non-symmetric modification of the coefficient matrix. One can monitor the number of Krylov method iteration and realize that with a symmetric coefficient matrix, the Conjugate Gradient method requires slightly fewer iterations than GMRES in the non-symmetric case. Taking into account that the Conjugate Gradient method has less work per iteration, there is some efficiency to be gained by using `assemble_system`.

hpl 42: Running `application_linalg`, the results are strange: Why does the `solve(a==L,...)` method need many more iterations than `solve(A, U, b, ...)` when we use the same Krylov parameter settings? Something wrong with the settings?

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

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

Note that we must turn off the default behavior of setting the start vector ("initial guess") to zero, and then the provided value of `U` is used as start vector.

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.

5.3 A variable-coefficient Poisson problem

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

$$\begin{aligned} -\nabla\cdot[p(x,y)\nabla u(x,y)] &= f(x,y) \quad \text{in } \Omega, \\ u(x,y) &= u_{\mathrm{b}}(x,y) \quad \text{on } \partial\Omega\,. \end{aligned} \tag{5.3}$$

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

Test problem. Let us continue to use our favorite solution $u(x,y) = 1+x^2+2y^2$ and then prescribe $p(x,y) = x+y$. It follows that $u_{\mathrm{b}}(x,y) = 1+x^2+2y^2$ and $f(x,y) = -8x-10y$.

5.3.1 Modifications of the PDE solver

What are the modifications we need to do in the previously shown codes to incorporate the variable coefficient p? from Section 5.1.3?

- `solver` must take `p` as argument,
- `f` in our test problem must be an `Expression` since it is no longer a constant,
- a new `Expression p` must be defined for the variable coefficient,

- the formula for $a(u,v)$ in the variational problem is slightly changed.

First we address the modified variational problem. Multiplying the PDE by a test function v and integrating by parts now results in

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

The function spaces for u and v are the same as in the problem with $p=1$, implying that the boundary integral vanishes since $v=0$ on $\partial\Omega$ where we have Dirichlet conditions. The weak form $a(u,v)=L(v)$ then has

$$a(u,v)=\int_\Omega p\nabla u\cdot\nabla v\,\mathrm{d}x,\quad L(v)=\int_\Omega fv\,\mathrm{d}x\,. \tag{5.4}$$

In the code for solving $-\nabla^2 u=f$ we must replace

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

by

```
a = p*dot(grad(u), grad(v))*dx
```

to solve $-\nabla\cdot(p\nabla u)=f$. Moreover, the definitions of `p` and `f` in the test problem read

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

No additional modifications are necessary. The file `ft08_poisson_vc.py` (variable-coefficient Poisson problem in 2D) is a copy of `ft07_poisson_iter.py` with the mentioned changes incorporated. Observe that $p=1$ recovers the original problem in `ft07_poisson_iter.py`.

You can run it and confirm that it recovers the exact u at the nodes.

5.3.2 Flux computations

The flux $-p\nabla u$ is often of interest to compute. The formula is simple: since $u=\sum_{j=1}^N U_j\phi_j$, we have that

$$-p\nabla u = -p\sum_{j=1}^N U_j\nabla\phi_j\,.$$

Given the solution variable `u` in the program, its gradient is obtained by `grad(u)` or `grad(u)`. However, the gradient of a piecewise continuous finite element scalar field is a discontinuous vector field since the ϕ_j has discontinuous derivatives at the boundaries of the cells. For example, using Lagrange elements of degree 1, u is linear over each cell, and the numerical

∇u becomes a piecewise constant vector field. On the contrary, the exact gradient is continuous. For visualization and data analysis purposes we often want the computed gradient to be a continuous vector field. Typically, we want each component of ∇u to be represented in the same way as u itself. To this end, we can project the components of ∇u onto the same function space as we used for u. This means that we solve $w = \nabla u$ approximately by a finite element method, using the same elements for the components of w as we used for u. This process is known as *projection*.

Not surprisingly, projection is a so common operation in finite element programs that FEniCS has a function for doing the task: `project(q, W)`, which returns the projection of some `Function` or `Expression` object named `q` onto the `FunctionSpace` (if `q` is scalar) or `VectorFunctionSpace` (if `q` is vector-valued) named `W`. Specifically, in our case where `u` is computed and we want to project the vector-valued ∇u and $-p\nabla u$ onto the `VectorFunctionSpace` where each component has the same `Function` space as `u`:

```
V = u.function_space()
degree = u.ufl_element().degree()
W = VectorFunctionSpace(V.mesh(), 'P', degree)

grad_u = project(grad(u), W)
flux_u = project(-p*grad(u), W)
```

An appropriate function for computing the flux based on `u` and `p` is

```
def flux(u, p):
    """Return p*grad(u) projected onto same space as u."""
    V = u.function_space()
    mesh = V.mesh()
    degree = u.ufl_element().degree()
    V_g = VectorFunctionSpace(mesh, 'P', degree)
    flux_u = project(-p*grad(u), V_g)
    flux_u.rename('flux(u)', 'continuous flux field')
    return flux_u
```

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, title='flux field')

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

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 (demands a `deepcopy=True` in the split of `flux`):

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

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

```
flux_u_array = flux_u.vector().array()
```

but this is a flat `numpy` array where the degrees of freedom for the x component of the flux is stored in the first part, then the degrees of freedom of the y component, and so on. This is less convenient to work with.

The function `application_test_flux` in the program `ft08_poisson_vc.py` demonstrates the computations described above.

hpl 43: Not sure if we should use paper space on the box below. Could be in extended material only.

Detour: Manual projection.

Although you will always use `project` to project a finite element function, it can be constructive this point in the tutorial to formulate the projection mathematically and implement its steps manually in FEniCS.

Looking at the component $\partial u/\partial x$ of the gradient, we project the (discrete) derivative $\sum_j U_j \partial\phi_j/\partial x$ onto a function space with basis $\phi_1, \phi_2, \ldots$ such that the derivative in this space is expressed by the standard sum $\sum_j \bar{U}_j \phi_j$, for suitable (new) coefficients $\bar{U}_j$.

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

$$a(w, v) = L(v) \quad \forall v \in \hat{V^{(\mathrm{g})}}, \tag{5.5}$$

where

$$a(w, v) = \int_\Omega w \cdot v \, \mathrm{d}x, \tag{5.6}$$

$$L(v) = \int_\Omega \nabla u \cdot v \, \mathrm{d}x \,. \tag{5.7}$$

The function spaces $V^{(\mathrm{g})}$ and $\hat{V^{(\mathrm{g})}}$ (with the superscript g denoting "gradient") are vector versions of the function space for u, with boundary conditions removed (if V is the space we used for u, with no restrictions on boundary values, $V^{(\mathrm{g})} = \hat{V^{(\mathrm{g})}} = [V]^d$, where d is the number

of space dimensions). For example, if we used piecewise linear functions on the mesh to approximate u, the variational problem for w corresponds to approximating each component field of w by piecewise linear functions.

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

```
V_g = VectorFunctionSpace(mesh, 'P', 1)
w = TrialFunction(V_g)
v = TestFunction(V_g)

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

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

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

5.3.3 Taking advantage of structured mesh data

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

A necessary first step is to transform our `mesh` object to an object representing a rectangle with equally-shaped *rectangular* cells. The 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 `src/modules`, associated with this booklet, we have included a Python module `BoxField` that can take a finite element function `u` computed by a FEniCS software and represent it on a structured box-shaped mesh and assign or extract values by multi-dimensional indexing: `[i]` in 1D, `[i,j]` in 2D, and `[i,j,k]` in 3D. Given a finite element function `u`, the following function returns a `BoxField` object that represents `u` on a structured mesh:

```
def structured_mesh(u, divisions):
    """Represent u on a structured mesh."""
    # u must have P1 elements, otherwise interpolate to P1 elements
    u2 = u if u.ufl_element().degree() == 1 else \
         interpolate(u, FunctionSpace(mesh, 'P', 1))
    mesh = u.function_space().mesh()
    from BoxField import fenics_function2BoxField
    u_box = fenics_function2BoxField(
        u2, mesh, divisions, uniform_mesh=True)
    return u_box
```

Note that we can only turn functions on meshes with P1 elements into `BoxField` objects, so if `u` is based on another element type, we first interpolate the scalar field onto a mesh with P1 elements. Also note that to use the function, we need to know the divisions into cells in the various spatial directions (`divisions`).

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]` (for vectorized computations or surface plotting)
- `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], u_box.grid.coor[Y])`

Iterating over points and values. Let us go back to the `solver` function in the `ft08_poisson_vc.py` code from Section 5.3, compute `u`, map it onto a `BoxField` object for a structured mesh representation, and write out 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       # numpy array
X = 0;  Y = 1           # for indexing in x and y direction

# Iterate over 2D mesh points (i,j)
```

```
print('u_ is defined on a structured mesh with %s points' %
      str(u_.shape))
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[X][j],
               u_[i,j]))
```

Finite difference approximations. Note that with `u_`, we can easily express finite difference approximation 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
```

Surface plot. The ability to access a finite element field in the way one can access a finite difference-type of field is handy in many occasions, including visualization and data analysis. With Matplotlib we can create a surface plot, see Figure 5.1 (upper left):

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
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_`.

Contour plot. A contour plot can also be made by Matplotlib:

```
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).

Curve plot through the mesh. A handy feature of `BoxField` objects is the ability to give a start point in the grid and a direction, and then extract the field and corresponding coordinates along the nearest line of mesh points. In 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`, are extracted by

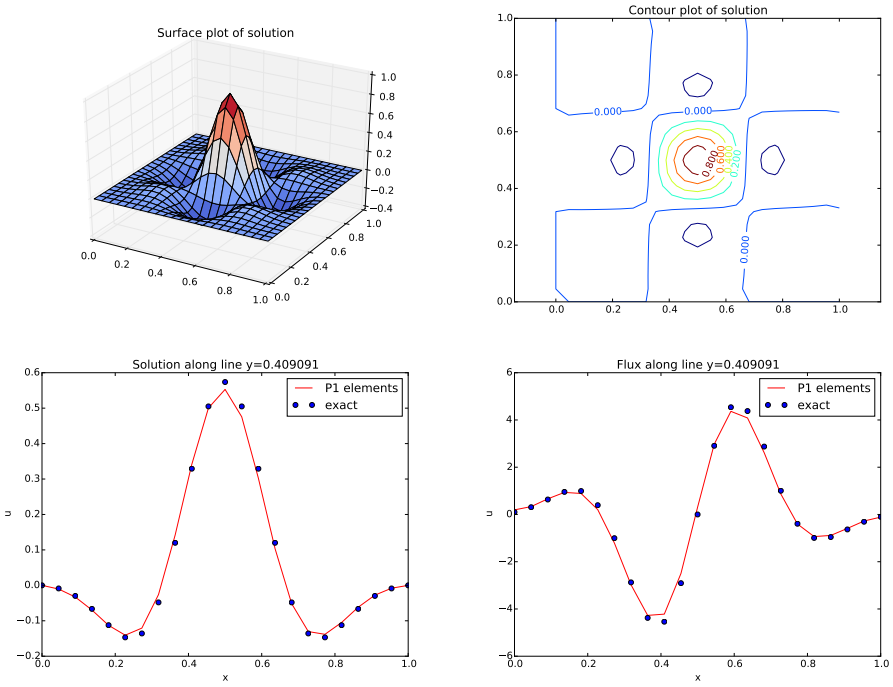


Fig. 5.1 Various plots of the solution on a structured mesh.

```
start = (0, 0.4)
X = 0
x, u_val, y_fixed, snapped = u_box.gridline(start, direction=X)
```

The variable `snapped` is true if the line had to be snapped onto a gridline and in that case `y_fixed` holds the snapped (altered) y value. To avoid interpolation in the structured mesh, `snapped` is in fact *always* true.

A comparison of the numerical and exact solution along the line $y = 0.5$ (snapped from $y = 0.4$) is made by the following code:

```
start = (0, 0.4)
x, u_val, y_fixed, snapped = u_box.gridline(start, direction=X)
u_e_val = [u_b((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='upper left')
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.

Curve plot of the flux. Let us also compare the numerical and exact flux $-p\partial u/\partial x$ along the same line as above:

```
flux_u = flux(u, p)
flux_u_x, flux_u_y = flux_u.split(deepcopy=True)

# Plot the numerical and exact flux along the same line
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 = structured_mesh(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='upper right')
plt.title('Flux along line y=%g' % y_fixed)
plt.xlabel('x');  plt.ylabel('u')
```

The second `plt.plot` command requires a Python function `flux_u_x_exact(x,y)` to be available for the exact flux expression.

Note that Matplotlib is one choice of plotting package. With the unified interface in the SciTools package 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

$$H(x) = e^{-16(x-\frac{1}{2})^2}\sin(3\pi x)\,.$$

We just fit a function $f(x,y)$ in the PDE (can choose $p=1$), and notice that $u=0$ along the boundary of the unit square. 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_c = sym.printing.ccode(u)
```

Printing out `u_c` 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_c = u_c.replace('M_PI', 'pi')
u_b = Expression(u_c)
```

Thereafter, we can progress with the computation of $f = -\nabla\cdot(p\nabla u)$:

```
p = 1
f = sym.diff(-p*sym.diff(u, x), x) + sym.diff(-p*sym.diff(u, y), y)
f = sym.simplify(f)
f_c = sym.printing.ccode(f)
f_c = f_c.replace('M_PI', 'pi')
f = Expression(f_c)
```

We also need a Python function for the exact flux $-p\partial u/\partial x$:

```
flux_u_x_exact = sym.lambdify([x, y], -p*sym.diff(u, x),
                              modules='numpy')
```

It remains to define `p = Constant(1)` and set `nx` and `ny` before calling `solver` to compute the finite element solution of this problem.

5.4 Postprocessing computations

hpl 28: Need a little intro.

5.4.1 Computing functionals

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

$$\frac{1}{2}||\nabla u||^2 \equiv \frac{1}{2}\int_\Omega \nabla u\cdot\nabla u\,\mathrm{d}x, \tag{5.8}$$

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

$$||u_\mathrm{e} - u|| = \left(\int_\Omega (u_\mathrm{e}-u)^2\,\mathrm{d}x\right)^{1/2}, \tag{5.9}$$

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

$$F = -\int_\Gamma p\nabla u\cdot\boldsymbol{n}\,\mathrm{d}s, \tag{5.10}$$

where $\boldsymbol{n}$ is an outward unit normal at Γ and p is a coefficient (see the problem in Section 5.3 for a specific example). All these functionals are easy to compute with FEniCS, and this section describes how it can be done.

Energy functional. The integrand of the energy functional (5.8) 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 `assemble` call performs the integration. It is possible to restrict the integration to subdomains, or parts of the boundary, by using a mesh function to mark the subdomains as explained in Section 4.1.4.

Error functional. Computation of (5.9) is typically done by

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

The exact solution u_e is here in a `Function` or `Expression` object `u_exact`, while `u` is the finite element approximation. (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 demonstrate in Section 5.4.2, the integration of `(u - u_exact)**2*dx` can result in too optimistic convergence rates unless one is careful how `u_exact` is transferred onto a mesh. The general recommendation for reliable error computation is to use the `errornorm` function (see `pydoc fenics.errornorm` and Section 5.4.2 for more information):

```
E = errornorm(u_exact, u)
```

Flux Functionals. To compute flux integrals like $F = -\int_\Gamma p\nabla u\cdot\boldsymbol{n}\,\mathrm{d}s$ we need to define the $\boldsymbol{n}$ vector, referred to as *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 = -p*dot(grad(u), n)*ds
total_flux = assemble(flux)
```

Although `grad(u)` and `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 Laplace/Poisson PDE. With `grad(u)` we must in that case write `dot(n, grad(u))`.

It is possible to restrict the integration to a part of the boundary using a mesh function to mark the relevant part, as explained in Section 4.1.4. Assuming that the part corresponds to subdomain number `i`, the relevant syntax for the variational formulation of the flux is `-p*dot(grad(u), n)*ds(i)`.

5.4.2 Computing convergence rates

hpl 45: Newer FEniCS examples have `dx(degree)`. Should explain that syntax. Also `Expression(string, degree)`.

To illustrate error computations and convergence of finite element solutions, we have included a function `convergence_rate` in the `ft08_poisson_vc.py` program. This is a tool that is very handy when verifying finite element codes and will therefore be explained in detail here.

The L^2 norm of the error in a finite element approximation u, u_{e} being the exact solution, is given by

Various ways of computing the error.

$$E = \left(\int_\Omega (u_e - u)^2 \, \mathrm{d}x \right)^{1/2},$$

and implemented in FEniCS by

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

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.

We remark that `u_e` will, in the expression above, be interpolated onto the function space `V` before `assemble` can perform the integration over the domain. This implies that the exact solution used in the integral will vary linearly over the cells, and not as a sine function, if `V` corresponds to linear Lagrange elements. This situation may yield a smaller error `u - u_e` than what is actually true. More accurate representation of the exact solution is easily achieved by interpolating the formula onto a space defined by higher-order elements, say of third degree:

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

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

```
u_Ve = interpolate(u, Ve)
```

```
error = (u_Ve - u_e_Ve)**2*dx
```

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

```
E = errornorm(u_exact, u, normtype='L2', degree=3)
```

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

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

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

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

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

All the various types of error computations here are placed in a function `compute_errors` in `ft08_poisson_vc.py`: **hpl 46**: Necessary to repeat code? New info is essentiall the return dict. **hpl 47**: Anders, I (in 2010...) ran into problems with `fenics.errornorm`, see comments in the code below, and made the version below. We should check out these problems again and adjust `fenics.errornorm` if necessary.

```
def compute_errors(u, u_exact):
    """Compute various measures of the error u - u_exact, where
    u is a finite element Function and u_exact is an Expression."""

    # Compute error norm (for very small errors, the value can be
    # negative so we run abs(assemble(error)) to avoid failure in sqrt

    V = u.function_space()

    # Function - Expression
    error = (u - u_exact)**2*dx
    E1 = sqrt(abs(assemble(error)))

    # Explicit interpolation of u_e onto the same space as u:
    u_e = interpolate(u_exact, V)
```

```
    error = (u - u_e)**2*dx
    E2 = sqrt(abs(assemble(error)))

    # Explicit interpolation of u_exact to higher-order elements,
    # u will also be interpolated to the space Ve before integration
    Ve = FunctionSpace(V.mesh(), 'P', 5)
    u_e = interpolate(u_exact, Ve)
    error = (u - u_e)**2*dx
    E3 = sqrt(abs(assemble(error)))

    # fenics.errornorm interpolates u and u_e to a space with
    # given degree, and creates the error field by subtracting
    # the degrees of freedom, then the error field is integrated
    # TEMPORARY BUG - doesn't accept Expression for u_e
    #E4 = errornorm(u_e, u, normtype='l2', degree=3)
    # Manual implementation errornorm to get around the bug:
    def errornorm(u_exact, u, Ve):
        u_Ve = interpolate(u, Ve)
        u_e_Ve = interpolate(u_exact, Ve)
        e_Ve = Function(Ve)
        # Subtract degrees of freedom for the error field
        e_Ve.vector()[:] = u_e_Ve.vector().array() - u_Ve.vector().array()
        # More efficient computation (avoids the rhs array result above)
        #e_Ve.assign(u_e_Ve)                      # e_Ve = u_e_Ve
        #e_Ve.vector().axpy(-1.0, u_Ve.vector())  # e_Ve += -1.0*u_Ve
        error = e_Ve**2*dx(Ve.mesh())
        return sqrt(abs(assemble(error))), e_Ve
    E4, e_Ve = errornorm(u_exact, u, Ve)

    # Infinity norm based on nodal values
    u_e = interpolate(u_exact, V)
    E5 = abs(u_e.vector().array() - u.vector().array()).max()

    # H1 seminorm
    error = dot(grad(e_Ve), grad(e_Ve))*dx
    E6 = sqrt(abs(assemble(error)))

    # Collect error measures in a dictionary with self-explanatory keys
    errors = {'u - u_exact': E1,
              'u - interpolate(u_exact,V)': E2,
              'interpolate(u,Ve) - interpolate(u_exact,Ve)': E3,
              'errornorm': E4,
              'infinity norm (of dofs)': E5,
              'grad(error) H1 seminorm': E6}

    return errors
```

Computing convergence rates empirically. Calling the `solver` function for finer and finer meshes enables us to study the convergence rate. Define the element size $h = 1/n$, where n is the number of cell divisions in x and y direction (`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_3 and so forth. Assuming

$E_i = Ch_i^r$ for unknown constants C and r, we can compare two consecutive experiments, $E_i = Ch_i^r$ and $E_{i-1} = Ch_{i-1}^r$, and solve for r:

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

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

The procedure above can easily be turned into Python code. Here we run through a different types of elements (P1, P2, P3, and P4), perform experiments over a series of refined meshes, and for each experiment report the six error types as returned by `compute_errors`:

```
def convergence_rate(u_exact, f, u_b, p, degrees,
                     n=[2**(k+3) for k in range(5)]):
    """
    Compute convergence rates for various error norms for a
    sequence of meshes with Nx=Ny=b and P1, P2, ...,
    Pdegrees elements. Return rates for two consecutive meshes:
    rates[degree][error_type] = r0, r1, r2, ...
    """

    h = {}  # Discretization parameter, h[degree][experiment]
    E = {}  # Error measure(s), E[degree][experiment][error_type]
    P_degrees = 1,2,3,4
    num_meshes = 5

    # Perform experiments with meshes and element types
    for degree in P_degrees:
        n = 4   # Coarsest mesh division
        h[degree] = []
        E[degree] = []
        for i in range(num_meshes):
            n *= 2
            h[degree].append(1.0/n)
            u = solver(p, f, u_b, n, n, degree,
                       linear_solver='direct')
            errors = compute_errors(u, u_exact)
            E[degree].append(errors)
            print('2*(%dx%d) P%d mesh, %d unknowns, E1=%g' %
                  (n, n, degree, u.function_space().dim(),
                   errors['u - u_exact']))
    # Convergence rates
    from math import log as ln  # log is a fenics name too
    error_types = list(E[1][0].keys())
    rates = {}
    for degree in P_degrees:
        rates[degree] = {}
        for error_type in sorted(error_types):
            rates[degree][error_type] = []
            for i in range(num_meshes):
                Ei   = E[degree][i][error_type]
                Eim1 = E[degree][i-1][error_type]
```

```
                r = ln(Ei/Eim1)/ln(h[degree][i]/h[degree][i-1])
                rates[degree][error_type].append(round(r,2))
    return rates
```

Test problem. Section 5.3.2 specifies a more complicated solution,

$$u(x,y) = \sin(\omega\pi x)\sin(\omega\pi y)$$

on the unit square. This choice implies $f(x,y) = 2\omega^2\pi^2 u(x,y)$. With ω restricted to an integer it follows that $u_\mathrm{b} = 0$.

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(0.0), boundary)

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

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

Experiments. The function `convergence_rate_sin()` in `ft08_poisson_vc.py` implements the test problem above and applies the `convergence_rate` function to estimate convergence rates. We achieve some interesting results. Using the error measure `E5` based on the infinity norm of the difference of the degrees of freedom, we have

element	$n=8$	$n=16$	$n=32$	$n=64$	$n=128$
P1	1.99	1.97	1.99	2.0	2.0
P2	3.99	3.96	3.99	4.0	3.99
P3	3.96	3.89	3.96	3.99	4.0
P4	3.75	4.99	5.0	5.0	

The computations with P4 elements on a $2(128\times 128)$ mesh with a direct solver (UMFPACK) on a small laptop broke down. Otherwise we achieve expected results: the error goes like h^{d+1} for elements of degree d. Also L^2 norms based on the `errornorm` gives the expected h^{d+1} rate for u and h^d for ∇u.

However, using `(u - u_exact)**2` for the error computation, which implies interpolating `u_exact` onto the same space as `u`, results in h^4 convergence for P2 elements.

element	$n = 8$	$n = 16$	$n = 32$	$n = 64$	$n = 128$
P1	1.98	1.94	1.98	2.0	2.0
P2	3.98	3.95	3.99	3.99	3.99
P3	3.69	4.03	4.01	3.95	2.77

This is an example where it is important to interpolate `u_exact` to a higher-order space (polynomials of degree 3 are sufficient here) to avoid computing a too optimistic convergence rate.

Checking convergence rates is the next best method for verifying PDE codes (the best being a numerical solution without approximation errors as in Section 5.1.3 and many other places in this tutorial).

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] Dietrich Braess. *Finite Elements*. Cambridge University Press, Cambridge, third edition, 2007.

[5] Susanne C. Brenner and L. Ridgway 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] Philippe 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] Katuhiko 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] J. Kiusalaas. *Numerical Methods in Engineering With Python*. Cambridge University Press, 2005.

[17] R. H. Landau, M. J. Paez, and C. C. Bordeianu. *Computational Physics: Problem Solving with Python*. Wiley, third edition, 2015.

[18] H. P. Langtangen. *Python Scripting for Computational Science*. Springer, third edition, 2009.

[19] H. P. Langtangen. *A Primer on Scientific Programming With Python*. Texts in Computational Science and Engineering. Springer, fifth edition, 2016.

[20] H. P. Langtangen and L. R. Hellevik. Brief tutorials on scientific Python. *http://hplgit.github.io/bumpy/doc/web/index.html*.

[21] H. P. Langtangen and A. Logg. *The Advanced FEniCS Tutorial - Writing State-of-the-art Finite Element Solvers in Hours*. Springer, 2016.

[22] M. G. Larson and F. Bengzon. *The Finite Element Method: Theory, Implementation, and Applications*. Texts in Computational Science and Engineering. Springer, 2013.

[23] A. Logg, K.-A. Mardal, and G. N. Wells. *Automated Solution of Partial Differential Equations by the Finite Element Method*. Springer, 2012.

[24] M. Pilgrim. *Dive into Python*. Apress, 2004. *http://www.diveintopython.net*.

[25] A. Quarteroni and A. Valli. *Numerical Approximation of Partial Differential Equations*. Springer Series in Computational Mathematics. Springer, 1994.

[26] A. Henderson Squillacote. *The Paraview Guide*. Kitware, 2007.

[27] R. Temam. Sur l'approximation de la solution des équations de navier-stokes. *Arc. Ration. Mech. Anal.*, 32:377–385, 1969.

Index