Chapter 1
Preliminaries

1.1 The FEniCS Project

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

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

1.2 What you will learn

The goal of this tutorial is to 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 short and easy with FEniCS.

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

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

Most books on a programming language start with a "Hello, World!" program. That is, one is curious about how a very fundamental task is expressed in the language, and writing a text to the screen can be such a task. In the world of *finite element methods for PDEs*, the most fundamental task must be to solve the Poisson equation. Our counterpart to the classical "Hello, World!" program therefore solves

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

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

Here, $u = u(x)$ is the unknown function, $f = f(x)$ 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 for 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 a similar program in most other software frameworks for PDEs require much more code and more technically difficult programming.

What makes FEniCS attractive?

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

2.1.1 Finite element variational formulation

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

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:

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

2.1.3 Choosing a test problem

The Poisson problem (2.1)–(2.2) has so far featured a general domain Ω and general functions u_{D} for the boundary conditions and f for the right-hand side. For our first implementation we will need to make specific choices for Ω, u_{D}, and f. It will be wise to construct a problem 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

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

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

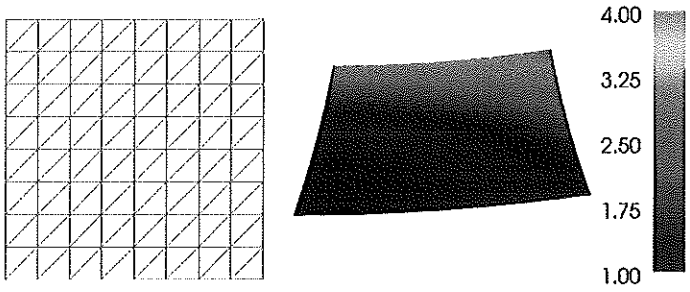


Fig. 2.1 Plot of the solution in the first FEniCS example.

Spyder. Many prefer to work in an integrated development environment that provides an editor for programming, a window for executing code, a window for inspecting objects, etc. 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 press Shift+Enter to execute the cell. The file `ft01_poisson.py` will then be loaded into the notebook. Re-execute the cell (Shift+Enter) to run the program. You may divide the entire program into several cells to examine intermediate results: place the cursor where you want to split the cell and choose **Edit - Split Cell.**

mention %matplotlib_inline for plotting?

2.3 Dissection of the program

We shall now dissect our FEniCS program in detail. The listed FEniCS program defines a finite element mesh, a finite element function space V on this mesh, boundary conditions for u (the function u_{D}), and the bilinear and linear forms $a(u,v)$ and $L(v)$. Thereafter, the 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.

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.

2.3.7 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 $\nabla u \cdot \nabla v dx$ and $fvdx$. This is a key strength of FEniCS: the formulas in the variational formulation translate directly to very similar Python code, a feature that makes it easy to specify and solve complicated PDE problems. The language used to express weak forms is called UFL (Unified Form Language) [1,26] and is an integral part of FEniCS.

Expressing inner products

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

2.3.8 Forming and solving the linear system

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

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

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

2.3.9 Plotting the solution

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.

Plotting in Docker

When running FEniCS from the command-line in a Docker container, plotting is disabled. Visualization of solutions must then be made using external applications such as Paraview running on the host system. Post-processing of solutions is the topic of the following section.

matplotlib plotting?

2.3.10 Exporting and post-processing the solution

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

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

The `solution.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`).

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

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

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

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

With numpy arrays we can write MATLAB-like code to analyze the data. Indexing is done with square brackets: array_u[j], where the index j always starts at 0. If the solution is computed with piecewise linear Lagrange elements (P_1), then the size of the array array_u is equal to the number of vertices, and each array_u[j] is the value at some vertex in the mesh. However, the degrees of freedom are not necessarily numbered in the same way as the vertices of the mesh, see Section 5.2.6 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:

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

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

2.4 Deflection of a membrane

Our first FEniCS program for the Poisson equation targeted a simple test problem where we could easily verify the implementation. 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 equation

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, 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 this particular case 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

with $\beta = 0$

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

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 scaled solution w, the physical deflection can be computed by

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

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

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

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

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

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

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

We use a finite element method to solve (3.7) and either of the equations (3.8) or (3.9). This requires turning the equations into weak forms. As usual, we multiply by a test function $v \in \hat{V}$ and integrate second-derivatives by parts. Introducing the symbol u for u^{n+1} (which is natural in the program), the resulting weak form arising from formulation (3.8) can be conveniently written in the standard notation:

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

where

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

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

The alternative form (3.9) has an abstract formulation

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

where

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

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. The most common choice is `project`, which computes an approximation to u_0, but in some applications where we want to verify the code by reproducing exact solutions, one must use `interpolate` (and we use such a test problem!).

In summary, we thus need to solve the following sequence of variational problems to compute the finite element solution to the heat equation: find $u^0 \in V$ such that $a_0(u^0, v) = L_0(v)$ holds for all $v \in \hat{V}$, and then find $u^{n+1} \in V$ such that $a(u^{n+1}, v) = L_{n+1}(v)$ for all $v \in \hat{V}$, or alternatively, $F(u^{n+1}, v) = 0$ for all $v \in \hat{V}$, for $n = 0, 1, 2, \ldots$.

3.1.3 FEniCS implementation

Our program needs to implement the time-stepping manually, but can rely on FEniCS to easily compute a_0, L_0, 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 $u_{\mathrm{D}}(x, y, t) = 1 + x^2 + \alpha y^2 + \beta t$ and the initial value is $u_0(x, y) = 1 + x^2 + \alpha y^2$.

FEniCS implementation. A new programming issue is how to deal with functions that vary in space *and time*, such as the boundary condition $u_{\mathrm{D}}(x, y, t) = 1 + x^2 + \alpha y^2 + \beta t$. A natural solution is to use a FEniCS `Expression` with time t as a parameter, in addition to the parameters α and β:

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

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

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.

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

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

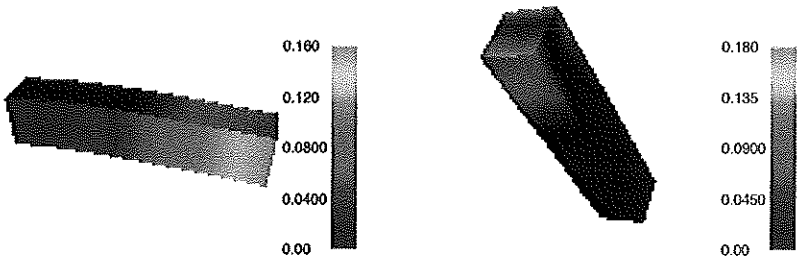


Fig. 3.1 Gravity-induced deformation of a clamped beam: deflection (left) and stress intensity seen from below (right).

3.4 The Navier–Stokes equations

As our next 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. We shall touch on a number of FEniCS topics, many of them quite advanced. But you will see that even a relatively complex algorithm such as

a second-order splitting method for the incompressible Navier–Stokes equations, can be implemented with relative ease in FEniCS.

3.4.1 PDE problem

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

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

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

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

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

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

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

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

3.4.2 Variational formulation

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

Instead, we will apply a simpler and often very efficient approach, known as a *splitting method*. The idea is to consider the two equations (3.29) and (3.30) separately. There exist many splitting strategies for the incompress-

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

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

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

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

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: $(\nabla u)_{i,j} = \partial u_j/\partial x_i = (\mathrm{grad}\, u)^\mathsf{T}$. 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)`.

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

ible Navier–Stokes equations. One of the oldest is the method proposed by Chorin [6] and Temam [31], often referred to as *Chorin's method*. We will use a modified version of Chorin's method, the so-called incremental pressure correction scheme (IPCS) due to [13] which gives improved accuracy compared to the original scheme at little extra cost.

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

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

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

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

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

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

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

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

where $T = \sigma \cdot n$ is the boundary traction. If we solve a problem with a free boundary, we can take $T = 0$ on the boundary. However, if we compute the flow through a channel or a pipe and want to model flow that continues into an "imaginary channel" at the outflow, we need to treat this term with some care. The assumption we then make is that the derivative of the velocity in the direction of the channel is zero at the outflow, corresponding to a flow that is "fully developed" or doesn't change significantly downstream of the outflow. Doing so, the remaining boundary term at the outflow becomes $pn - \mu \nabla u^T \cdot n$, which is the term appearing in the variational problem (3.32).

($pn - \mu \nabla u^T n = 0$ is consistent with the code using nabla_grad(u)*n)

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

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

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

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

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

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

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

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

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

3.4.3 FEniCS implementation

Test problem 1: Channel flow. As a first test problem, we compute the flow between two infinite plates, so-called channel or Poiseuille flow, 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 on diffusion accross the channel: $t_c = H^2/\nu$. We let U, some characteristic inflow velocity, be the velocity scale and H the spatial scale. The pressure scale is taken as the characteristic shear stress, $\mu U/H$, since this is a primary example of shear flow. Inserting $\bar{x} = x/H$, $\bar{y} = y/H$, $\bar{z} = z/H$, $\bar{u} = u/U$, $\bar{p} = Hp/(\mu U)$,

and $\bar{t} = H^2/\nu$ in the equations results in the scaled Navier–Stokes equations (dropping bars after the scaling):

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

Here, $\mathrm{Re} = \varrho U H/\mu$ is the Reynolds number. Because of the time and pressure scale, which are different from convection-dominated fluid flow, the Reynolds number is associated with the convective term and not the viscosity term. 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 $\partial^2 p/\partial^2 x = 0$ so $\partial p/\partial x$ is a constant, here called $-\beta$. This is the driving force of the flow and can be specified as a known parameter in the problem. Integrating $u_x''(y) = -\beta$ over the width of the channel, $[0,1]$, and requiring $u = 0$ 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 $y = 1/2$, 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. Mathematically, the pressure must be prescribed at a point, but since p does not depend on y, we can set p to a known value, e.g. zero, along the outlet boundary $x = 1$. The result is $p(x) = 8(1-x)$ and $u_x = 4y(1-y)$.

The boundary conditions can be set as $p = 1$ at $x = 0$, $p = 0$ at $x = 1$ 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. Note that it is only meaningful to solve the Navier–Stokes equations in 2D or 3D geometries, although the underlying mathematical problem collapses to two 1D problems, one for $u_x(y)$ and one for $p(x)$.

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

For a specific engineering problem one wants to simulate a specific fluid and set corresponding parameters. A general solver is most naturally implemented with dimensions and the original physical parameters. However, the scaled problem simplifies numerical simulations a lot. First of all, it tells that

all fluids flow in the same way: it does not matter whether we have oil, gas, or water flowing between two plates, and it does not matter how fast the flow is (up to some criticial value of the Reynolds number where the flow becomes unstable and goes over to a complicated turbulent flow of totally different nature). This means that one simulation is enough to cover all types of channel flows! In other applications scaling tells us that it might be necessary to set just the fraction of some parameters (dimensionless numbers) rather than the parameters themselves. This simplifies exploring the input parameter space which is often the purpose of simulation. Frequently, the scaled problem is run by setting some of the input parameters with dimension to fixed values (often unity).

FEniCS implementation. Our previous examples have all started out with the creation of a mesh and then the definition of a `FunctionSpace` on the mesh. For the 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 `Q` is a scalar-valued function space for the pressure. We use piecewise quadratic elements for the velocity and piecewise linear elements for the pressure. When creating a `VectorFunctionSpace` in FEniCS, the value-dimension (the length of the vectors) will be set equal to the geometric dimension of the finite element mesh. One can easily create vector-valued function spaces with other dimensions in FEniCS by adding the keyword parameter `dim`:

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

Stable finite element spaces for the Navier–Stokes equations

It is well-known that certain finite element spaces are not *stable* for the Navier–Stokes equations, or even for the simpler Stokes equations. The prime example of an unstable pair of finite element spaces is to use first degree continuous piecewise polynomials for both the velocity and the pressure. Using an unstable pair of spaces typically results in a solution with *spurious* (unwanted, non-physical) oscillations in the pressure solution. The simple remedy is to use 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. The `near` function comes from FEniCS and performs a test with tolerance: `abs(x[0]-0) < 3E-16` so we do not run into rounding troubles. Alternatively, we may give the boundary definition as a string of C++ code, much like we have previously defined expressions such as `u0 = Expression('1 + x[0]*x[0] + 2*x[1]*x[1]', degree=2)`. The above definition of the boundary in terms of a Python function may thus be replaced by a simple C++ string:

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

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

For the current example, we will set three different boundary conditions. First, we will set $u = 0$ at the walls of the channel; that is, at $y = 0$ and $y = 1$. Second, we will set $p = 1$ 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

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:

ILU, like LU, does not scale well

```
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.

Choosing a linear algebra backend. The actual GMRES and ILU implementations that are brought into action depend on the choice of linear algebra package. FEniCS interfaces several linear algebra packages, called *linear algebra backends* in FEniCS terminology. PETSc is the default choice if FEniCS is compiled with PETSc. If PETSc is not available, then FEniCS falls back to using the Eigen backend. The linear algebra backend in FEniCS can be set using the following command:

```
parameters.linear_algebra_backend = backendname
```

where `backendname` is a string. To see which linear algebra backends are available, you can call the FEniCS function `list_linear_algebra_backends()`. Similarly, one may check which linear algebra backend is currently being used by the following command:

```
print parameters.linear_algebra_backend

# Alternative syntax for Python 3
print(parameters.linear_algebra_backend)
```

Setting solver parameters. We will normally want to control the tolerance in the stopping criterion and the maximum number of iterations when running an iterative method. Such parameters can be controlled at both a *global* and a *local* level. We will start by looking at how to set global parameters. For more advanced programs, one may want to use a number of different linear solvers and set different tolerances and other parameters. Then it becomes important to control the parameters at a *local* level. We will return to this issue in Section 5.2.3.

Changing a parameter in the global FEniCS parameter database affects all linear solvers (created *after* the parameter has been set). The global FEniCS parameter database is simply called `parameters` and it behaves as a nested dictionary. Write

```
info(parameters, verbose=True)
```

```
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
```

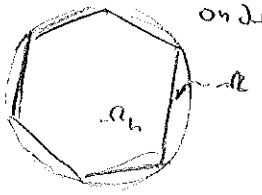
The striking feature is that the solver does not reproduce the solution to an accuracy more than about 0.01 (!), regardless of the resolution and type of element.
Filename: `membrane_func`.

But convergence should be linear ?

5.2 Working with linear solvers

Sparse LU decomposition (Gaussian elimination) is used by default to solve linear systems of equations in FEniCS programs. This is a very robust and simple method. It is the recommended method for systems with up to a few thousand unknowns and may hence be the method of choice for many 2D and smaller 3D problems. However, sparse LU decomposition becomes slow and one quickly runs out of memory for larger problems. For large problems, we instead need to use *iterative methods* which are faster and require much less memory. We will now look at how to take advantage of state-of-the-art iterative solution methods in FEniCS.

$-\Delta u = -\Delta \tilde{u} = 4$
$\tilde{u} = 0$ on $\partial\Omega$, $\tilde{u} = 0$ on $\partial\Omega_h$



$\|u - u_h\|_{\infty,\Omega_h}$
$\leq \|u - \tilde{u}\| + \|\tilde{u} - u_h\|$
$\to 0$
$= \|u\|_{\infty,\partial\Omega_h}$
(by maximum principle)
$\sim O(h)$

5.2.1 Controlling the solution process

Choosing a linear solver and preconditioner. Preconditioned Krylov solvers is a type of popular iterative methods that are easily accessible in FEniCS programs. The Poisson equation results in a symmetric, positive definite system matrix, for which the optimal Krylov solver is the Conjugate Gradient (CG) method. However, the CG method requires boundary conditions to be implemented in a symmetric way. This is not the case by default, so

Formally, this is true.
In practice, it is usually not necessary
(with a symmetric preconditioner, "cg" and "ilu" may fail.)

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.

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 file (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 `ft12_poisson_solver.py` in Section 5.1.1 such that it also offers the GMRES+ILU preconditioned Krylov solver:

This new `solver` function, found in the file `ft10_poisson_extended.py`, replaces the one in `ft12_poisson_solver.py`: it has all the functionality of the previous `solver` function, but can also solve the linear system with iterative methods.

A remark regarding unit tests. Regarding verification of the new `solver` function in terms of unit tests, it turns out that unit testing for a problem where the approximation error vanishes gets more complicated when we use iterative methods. The problem is to keep the error due to iterative solution smaller than the tolerance used in the verification tests. First of all, this means that the tolerances used in the Krylov solvers must be smaller than the tolerance used in the `assert` test, but this is no guarantee to keep the linear solver error this small. For linear elements and small meshes, a tolerance of 10^{-11} works well in the case of Krylov solvers too (using a tolerance 10^{-12} in those solvers). However, as soon as we switch to P_2 elements, it is hard to force the linear solver error below 10^{-6}. Consequently, tolerances in tests depend on the numerical method being used. The interested reader is referred to the `demo_solvers` function in `ft10_poisson_extended.py` for details: this

function tests the numerical solution for direct and iterative linear solvers, for different meshes, and different degrees of the polynomials in the finite element basis functions.

5.2.2 List of linear solver methods and preconditioners

Which linear solvers and preconditioners that are available in FEniCS depends on how FEniCS has been configured and which linear algebra backend is currently active. The following table shows an example of which linear solvers that can be available through FEniCS when the PETSc backend is active:

Name	Method
`'bicgstab'`	Biconjugate gradient stabilized method
`'cg'`	Conjugate gradient method
`'gmres'`	Generalized minimal residual method
`'minres'`	Minimal residual method
`'petsc'`	PETSc built in LU solver
`'richardson'`	Richardson method
`'superlu_dist'`	Parallel SuperLU
`'tfqmr'`	Transpose-free quasi-minimal residual method
`'umfpack'`	UMFPACK

The set of available preconditioners also depends on configuration and linear algebra backend. The following table shows an example of which preconditioners may be available:

Name	Method
`'icc'`	Incomplete Cholesky factorization
`'ilu'`	Incomplete LU factorization
`'petsc_amg'`	PETSc algebraic multigrid
`'sor'`	Successive over-relaxation

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

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

5.3 Postprocessing computations

As the final theme in this chapter, we will look at how to *postprocess computations*; that is, how to compute various derived quantities from the computed solution of a PDE. The solution u itself may be of interest for visualizing general features of the solution, but sometimes one is interested in computing the solution of a PDE to compute a specific quantity that derives from the solution, such as, e.g., the flux, a point-value, or some average of the solution.

5.3.1 A variable-coefficient Poisson problem

As a test problem, we will extend the Poisson problem from Chapter 2 with a variable coefficient $\kappa(x,y)$ in the Laplace operator:

$$-\nabla\cdot[\kappa(x,y)\nabla u(x,y)] = f(x,y) \quad \text{in } \Omega, \tag{5.3}$$

$$u(x,y) = u_{\mathrm{D}}(x,y) \quad \text{on } \partial\Omega. \tag{5.4}$$

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

We shall quickly demonstrate that this simple extension of our model problem only requires an equally simple extension of the FEniCS program. The following simple changes must be made to the previously shown codes:

- the `solver` function must take `k` (κ) as an argument,
- a new `Expression` `k` must be defined for the variable coefficient,
- the right-hand side `f` must be an `Expression` since it is no longer a constant,
- the formula for $a(u,v)$ in the variational problem must be updated.

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

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

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

$$a(u,v) = \int_\Omega \kappa\nabla u\cdot\nabla v\,\mathrm{d}x, \quad L(v) = \int_\Omega fv\,\mathrm{d}x. \tag{5.5}$$

We must thus replace

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

by

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

Moreover, the definitions of `k` and `f` in the test problem read

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

No additional modifications are necessary.

5.3.2 Flux computations

It is often of interest to compute the flux $Q = -\kappa\nabla u$. Since $u = \sum_{j=1}^{N} U_j\phi_j$, it follows that

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

However, the gradient of a piecewise continuous finite element scalar field is a discontinuous vector field since the basis functions $\{\phi_j\}$ have discontinuous derivatives at the boundaries of the cells. For example, using Lagrange elements of degree 1, u is linear over each cell, and the gradient becomes a piecewise constant vector field. On the contrary, the exact gradient is continuous. For visualization and data analysis purposes, we often want the computed gradient to be a continuous vector field. Typically, we want each component of ∇u to be represented in the same way as u itself. To this end, we can project the components of ∇u onto the same function space as we used for u. This means that we solve $w = \nabla u$ approximately by a finite element method, using the same elements for the components of w as we used for u. This process is known as *projection*.

Projection is a common operation in finite element analysis and FEniCS has a function for easily performing the projection: `project(expression, W)`, which returns the projection of some expression into the space `W`.

In our case, the flux $Q = -\kappa\nabla u$ is vector-valued and we need to pick `W` as the vector-valued function space of the same degree as the space `V` where `u` resides:

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

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

mesh is not callable error

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

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

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

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.

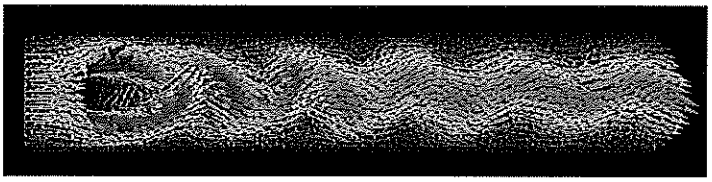


Fig. 3.4 Plot of the velocity for the cylinder test problem at final time.

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
```

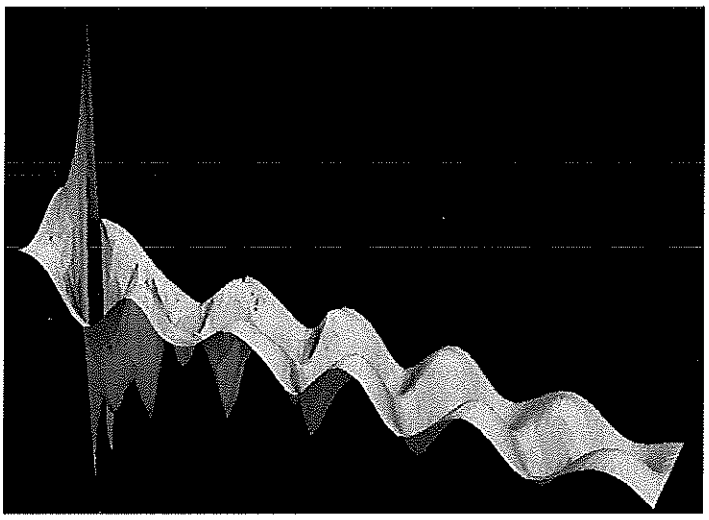


Fig. 3.5 Plot of the pressure for the cylinder test problem at final time.

```
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]
```