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.

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

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

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

Here, $u = u(\boldsymbol{x})$ is the unknown function, $f = f(\boldsymbol{x})$ is a prescribed function, ∇^2 is the Laplace operator (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

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 $v \in \hat{V}$.

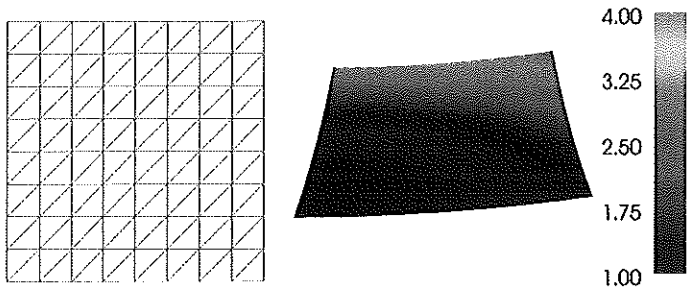

Fig. 2.1 Plot of the solution in the first FEniCS example.

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.

2.3.7 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 $fv \, dx$. 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.

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:

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

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

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

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

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

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

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 $\lambda = \beta$.

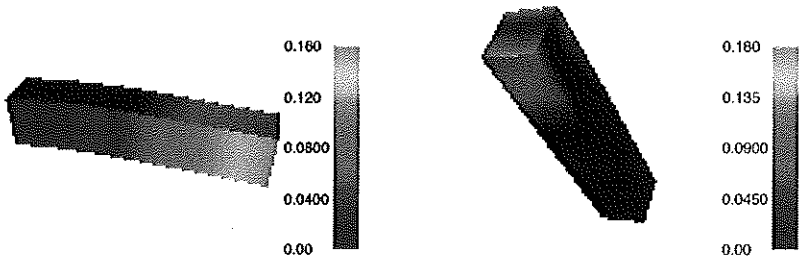

Fig. 3.1 Gravity-induced deformation of a clamped beam: deflection (left) and stress intensity seen from below (right).

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

`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.

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 $\bar{x} = x/H$, $\bar{y} = y/H$, $\bar{z} = z/H$, $\bar{u} = u/U$, $\bar{p} = Hp/(\mu U)$,

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.

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.

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)
```

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

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