1 A FEniCS tutorial

By Hans Petter Langtangen

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

1.1 Fundamentals

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

- how to define the PDE problem in terms of a variational problem,

- how to define simple domains,

- how to deal with Dirichlet, Neumann, and Robin conditions,

- how to deal with variable coefficients,

- how to deal with domains built of several materials (subdomains),

- how to compute derived quantities like the flux vector field or a functional of the solution,

- how to quickly visualize the mesh, the solution, the flux, etc.,

- how to solve nonlinear PDEs in various ways,

- how to deal with time-dependent PDEs,

- how to set parameters governing solution methods for linear systems,

- how to create domains of more complex shape.

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

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

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

1.1.1   *The Poisson equation*

Our first example regards the Poisson problem,

$$-\Delta u = f \quad \text{in } \Omega,$$
$$u = u_0 \quad \text{on } \partial\Omega.$$

(1.1)

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

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

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

(1.2)

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

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

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

1. Identify the PDE and its boundary conditions.

2. Reformulate the PDE problem as a variational problem.

3. Make a Python program where the formulas in the variational problem are coded, along with definitions of input data such as f, u_0, and a mesh for Ω in (1.1).

4. Add statements in the program for solving the variational problem, computing derived quantities such as ∇u, and visualizing the results.

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

1.1.2   *Variational formulation*

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

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

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

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

$$- \int_\Omega (\Delta u) v \, dx = \int_\Omega f v \, dx. \tag{1.3}$$

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

$$- \int_\Omega (\Delta u) v \, dx = \int_\Omega \nabla u \cdot \nabla v \, dx - \int_{\partial \Omega} \frac{\partial u}{\partial n} v \, ds, \tag{1.4}$$

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

$$\int_\Omega \nabla u \cdot \nabla v \, dx = \int_\Omega f v \, dx. \tag{1.5}$$

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

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

$$\int_\Omega \nabla u \cdot \nabla v \, dx = \int_\Omega f v \, dx \quad \forall v \in \hat{V}. \tag{1.6}$$

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

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

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

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

$$\int_\Omega \nabla u_h \cdot \nabla v \, dx = \int_\Omega f v \, dx \quad \forall v \in \hat{V}_h \subset \hat{V}. \tag{1.8}$$

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

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

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

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

In the present problem we have that

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

$$L(v) = \int_\Omega f v \, dx. \tag{1.11}$$

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

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

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

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

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

1.1.3   Implementation

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

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

on some 2D domain. By inserting (1.13) in our Poisson problem, we find that $u_e(x,y)$ is a solution if

$$f(x,y) = -6, \quad u_0(x,y) = u_e(x,y) = 1 + x^2 + 2y^2,$$

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

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

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

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

Python code

```python
from dolfin import *

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

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

def u0_boundary(x, on_boundary):
    return on_boundary

bc = DirichletBC(V, u0, u0_boundary)

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

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

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

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

# Hold plot
interactive()
```

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

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

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

The first line in the program,

Python code

```python
from dolfin import *
```

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

The statement

Python code

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

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

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

Python code

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

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

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

Python code

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

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

Python code

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

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

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

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

Python code

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

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

Python code

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

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

Python code

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

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

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

Python code

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

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

Python code

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

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

Python code

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

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

Python code

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

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

Python code

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

In essence, these two lines specify the PDE to be solved. Note the very close correspondence between the Python syntax and the mathematical formulas $\nabla u \cdot \nabla v \, dx$ 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 PDE problems with lots of PDEs and complicated terms in the equations. The language used to express weak forms is called UFL (Unified Form Language) and is an integral part of FEniCS.

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

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

Python code

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

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

Python code

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

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

Figure 1.1: Plot of the solution in the first FEniCS example. (A bounding box around the mesh is added by pressing o in the plot window, and the mouse buttons are then used to rotate and move the plot, see Section 1.1.8.)
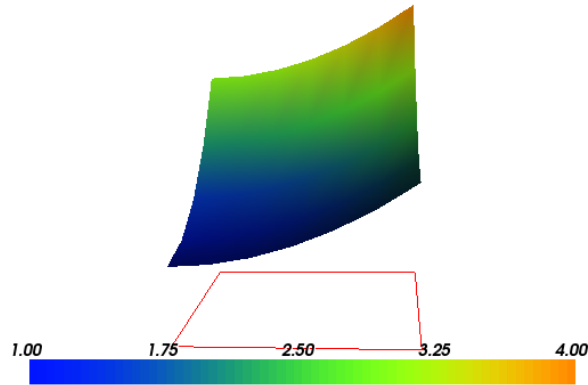
however, `TrialFunction` objects must always be used for the unknowns in the problem specification (the form a), while `Function` objects must be used for quantities that are computed (known).

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

Python code

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

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

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

Python code

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

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

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

1.1.4 Controlling the solution process

Sparse LU decomposition (Gaussian elimination) is used by default to solve linear systems of equations in FEniCS programs. This is a very robust and recommended method for a few thousand unknowns
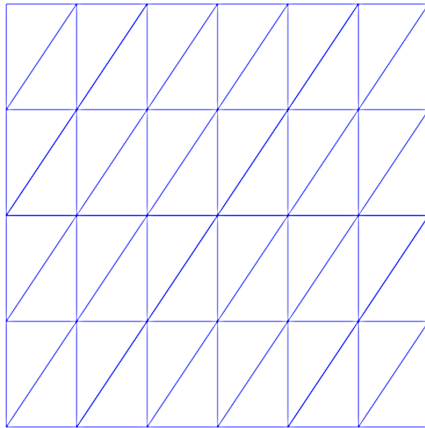
Figure 1.2: Plot of the mesh in the first FEniCS example.

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

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

Python code

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

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

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

Python code

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

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

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

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

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

Python code

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

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

Python code

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

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

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

Python code

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

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

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

Python code

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

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

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

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

1.1.5 Linear variational problem and solver objects

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

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

Python code

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

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

Python code

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

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

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

1.1.6 Examining the discrete solution

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

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

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

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

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

Python code

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

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

<div align="right">*Python code*</div>

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

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

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

- `mesh.coordinates()` returns the coordinates of the vertices as an $M \times d$ numpy array, M being the number of vertices in the mesh and d being the number of space dimensions,

- `mesh.num_cells()` returns the number of cells (triangles) in the mesh,

- `mesh.num_vertices()` returns the number of vertices in the mesh (with our choice of linear Lagrange elements this equals the number of nodes),

- `str(mesh)` returns a short "pretty print" description of the mesh, e.g.,

<div align="right">*Output*</div>

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

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

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

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

<div align="right">*Python code*</div>

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

The beginning of the output looks like this:

<div align="right">*Output*</div>

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

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

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

---

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

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

Python code

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

The maximum error can now be computed as

Python code

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

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

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

Python code

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

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

Output

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

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

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

Python code

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

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

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

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

1.1.7   *Solving a real physical problem*

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

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

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

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

with

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

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

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

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

with $w = 0$ on the boundary and with

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

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

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

1. Initialize T, A, R, x_0, y_0, and σ,

2. create a mesh over the unit circle,

3. make an expression object for the scaled pressure function f,

4. define the `a` and `L` formulas in the variational problem for w and compute the solution,

5. plot the mesh, w, and f,

6. write out the maximum real deflection D,

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

Python code

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

A mesh over the unit circle can be created by

Python code

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

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

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

Python code

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

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

Python code

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

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

Python code

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

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

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

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

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

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

Python code

```python
from dolfin import *

# Set pressure function:
```

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

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

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

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

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

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

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

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

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

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

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

1.1.8   Quick visualization with VTK

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

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

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

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

Python code

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

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

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

Python code

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

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

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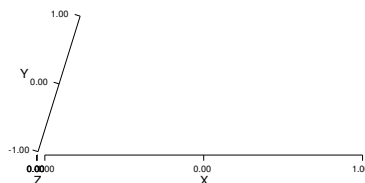
The `format` argument in the latter line can also take the values `"ps"` for a standard PostScript file and `"pdf"` for a PDF file. Note the necessity of the `viz_w.update(w)` call – without it we will not see the effects of tilting the camera and changing the color scale. Figure 1.3 shows the resulting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

1.1.9 Computing derivatives

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

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

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

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

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

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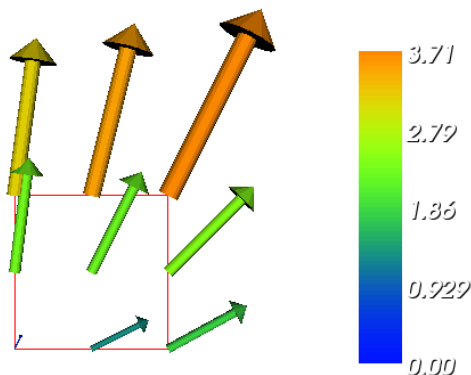
where

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

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

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

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

Python code

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

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

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

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

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

Python code

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

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

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

Python code

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

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

Python code

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

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

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

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

Python code

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

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

1.1.10 A variable-coefficient Poisson problem

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

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

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

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

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

1. `f` must be an `Expression` since it is no longer a constant,

2. a new `Expression` `p` must be defined for the variable coefficient,

3. the variational problem is slightly changed.

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

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

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

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

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

In the code from Section 1.1.3 we must replace

Python code

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

by

Python code

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

The definitions of `p` and `f` read

Python code

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

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

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

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

where

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

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

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

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

Python code

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

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

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

Python code

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

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

Python code

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

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

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

Python code

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

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

1.1.11 Computing functionals

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

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

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

$$||u_e - u|| = \left(\int_\Omega (u_e - u)^2 \, dx \right)^{1/2}, \tag{1.31}$$

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

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

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

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

Python code

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

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

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

in the membrane problem from Section 1.1.7.

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

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

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

Python code

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

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

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

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

The computation of (1.31) can be done by

Python code

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

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

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

Python code

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

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

Python code

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

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

Python code

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

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

Python code

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

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

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

Python code

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

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

Python code

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

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

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

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

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

The procedure above can easily be turned into Python code:

Python code

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

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

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

Output

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

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

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

Output

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

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

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

Output

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

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

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

<div align="right">Python code</div>

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

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

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

1.1.12   *Visualization of structured mesh data*

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

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

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

<div align="right">Python code</div>

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

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

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

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

Python code

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

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

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

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

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

Python code

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

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

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

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

Python code

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

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

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

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

Python code

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

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

Python code

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

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

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

Python code

```python
ev.figure()
flux2_x = flux_x if flux_x.ufl_element().degree() == 1 else \
            interpolate(flux_x, FunctionSpace(mesh, "Lagrange", 1))
flux_x_box = scitools.BoxField.dolfin_function2BoxField(
            flux2_x, mesh, (nx,ny), uniform_mesh=True)
x, fluxval, y_fixed, snapped = \
            flux_x_box.gridline(start, direction=X)
y = y_fixed
flux_x_exact = -(x + y)*2*x
ev.plot(x, fluxval, "r-",
        x, flux_x_exact, "b-",
        legend=("numerical (projected) flux", "exact flux"),
        title="Flux in x-direction (at y=%g)" % y_fixed,
        savefig="flux.eps")
```
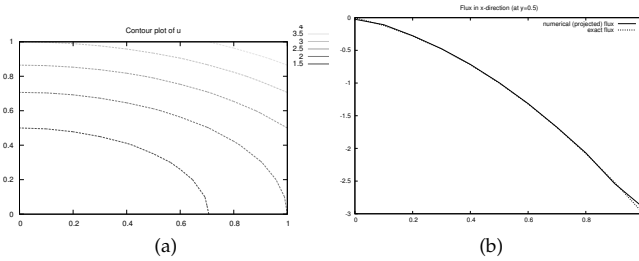
Figure 1.5: Examples of plots created by transforming the finite element field to a field on a uniform, structured 2D grid: (a) contour plot of the solution; (b) curve plot of the exact flux $-p\partial u/\partial x$ against the corresponding projected numerical flux.
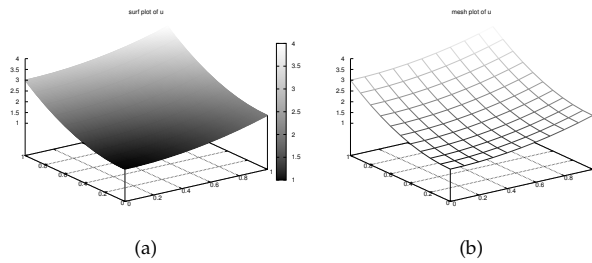

Figure 1.6: Examples of plots created by transforming the finite element field to a field on a uniform, structured 2D grid: (a) a surface plot of the solution; (b) lifted mesh plot of the solution.

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

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

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

1.1.13 *Combining Dirichlet and Neumann conditions*

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

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

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

Let Γ_D and Γ_N denote the parts of $\partial\Omega$ where the Dirichlet and Neumann conditions apply,

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

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

$$u = u_0 \text{ on } \Gamma_D, \tag{1.38}$$

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

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

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

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

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

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

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

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

$$-\int_\Omega (\Delta u) v \, dx = \int_\Omega \nabla u \cdot \nabla v \, dx - \int_{\partial\Omega} \frac{\partial u}{\partial n} v \, ds, \tag{1.44}$$

and since $v = 0$ on Γ_D,

$$-\int_{\partial\Omega} \frac{\partial u}{\partial n} v \, ds = -\int_{\Gamma_N} \frac{\partial u}{\partial n} v \, ds = \int_{\Gamma_N} g v \, ds, \tag{1.45}$$

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

$$\int_\Omega \nabla u \cdot \nabla v \, dx + \int_{\Gamma_N} g v \, ds = \int_\Omega f v \, dx. \tag{1.46}$$

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

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

$$L(v) = \int_\Omega f v \, dx - \int_{\Gamma_N} g v \, ds. \tag{1.48}$$

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

1. The function describing the boundary where Dirichlet conditions apply must be modified.

2. The new boundary term must be added to the expression in `L`.

Step 1 can be coded as

Python code

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

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

A more compact implementation reads

Python code

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

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

Python code

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

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

Python code

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

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

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

1.1.14 Multiple Dirichlet conditions

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

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

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

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

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

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

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

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

Python code

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

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

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

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

Python code

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

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

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

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

Python code

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

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

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

1.1.15 A linear algebra formulation

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

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

which is nothing but a linear system,

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

where the entries in A and b are given by

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

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

Python code

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

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

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

Python code

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

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

Python code

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

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

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

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

Python code

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

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

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

Python code

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

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

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

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

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

Python code

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

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

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

Python code

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

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

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

Python code

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

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

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

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

1.1.16 *Parameterizing the number of space dimensions*

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

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

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

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

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

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

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

Python code

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

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

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

Python code

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

is equivalent to

Python code

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

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

Python code

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

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

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

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

Python code

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

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

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

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

1.2 *Nonlinear problems*

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

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

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

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

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

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

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

where

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

and

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

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

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

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

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

1.2.1 *Picard iteration*

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

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

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

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

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

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

with

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

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

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

with

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

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

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

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

Python code

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

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

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

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

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

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

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

1.2.2   *A Newton method at the algebraic level*

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

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

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

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

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

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

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

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

The following results were used to obtain (1.73):

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

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

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

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

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

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

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

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

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

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

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

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

Python code

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

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

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

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

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

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

Python code

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

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

Python code

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

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

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

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

Python code

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

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

Python code

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

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

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

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

1.2.3   A Newton method at the PDE level

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

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

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

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

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

and dropping other nonlinear terms in δu, we get

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.2.4 *Solving the nonlinear variational problem directly*

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

Python code

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

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

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

Python code

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

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

Python code

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

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

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

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

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

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

and obtain

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

which leads to

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

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

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

Python code

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

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

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

Python code

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

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

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

Python code

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

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

or

Python code

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

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

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

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

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

Python code

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

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

solver.solve()
```

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

1.3   *Time-dependent problems*

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

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

1.3.1 A diffusion problem and its discretization

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

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

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

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

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

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

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

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

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

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

where dt is the time discretization parameter. Inserting (1.94) in (1.93) yields

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

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

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

$$u^k - dt\Delta u^k = u^{k-1} + dt f^k, \quad k = 1, 2, \dots \tag{1.97}$$

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

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

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

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

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

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

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

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

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

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

1.3.2 *Implementation*

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

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

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

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

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

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

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

Python code

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

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

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

Python code

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

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

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

Python code

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

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

The definition of a and L goes as follows:

Python code

```python
dt = 0.3       # time step

u = TrialFunction(V)
```

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

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

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

Finally, we perform the time stepping in a loop:

Python code

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

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

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

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

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

Python code

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

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

Python code

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

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

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

1.3.3   *Avoiding assembly*

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

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

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

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

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

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

which is nothing but two matrix-vector products,

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

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

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

and

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

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

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

That is, no assembly is necessary to compute b.

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

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

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

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

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

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

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

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

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

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

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

1. Define separate forms a_M and a_K

2. Assemble a_M to M and a_K to K

3. Compute $A = M + \mathrm{d}t\,K$

4. Define f as an `Expression`

5. Interpolate the formula for f to a finite element function F^k

6. Compute $b = MU^{k-1} + \mathrm{d}tMF^k$

The relevant code segments become

Python code

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

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

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

# 5. and 6.
while t <= T:
    f_k = interpolate(f, V)
    F_k = fk.vector()
    b = M*u_1.vector() + dt*M*F_k
```
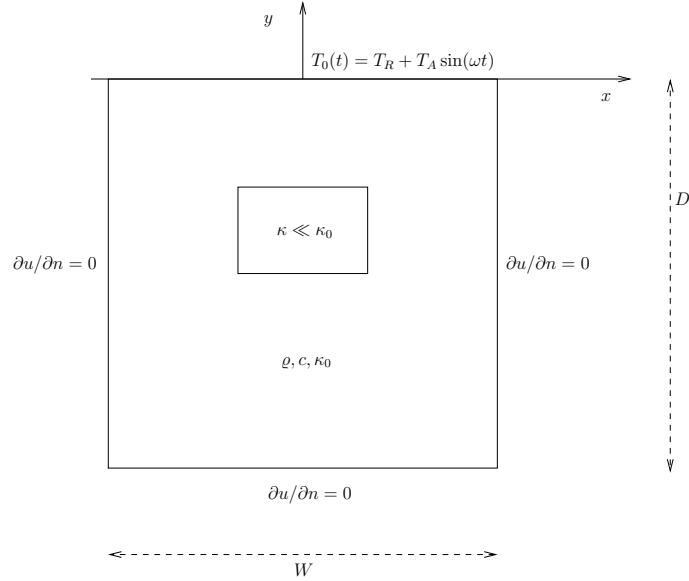
The complete program appears in the file `d2_d2D.py`.

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

1.3.4 A physical example

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

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

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

The initial-boundary value problem for this problem reads

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Python code

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

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

Setting Dirichlet conditions at the upper boundary can be done by

Python code

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

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

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

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

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

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

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

Python code

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

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

Python code

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

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

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

Python code

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

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

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

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

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

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

Python code

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

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

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

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

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

Python code

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

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

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

Python code

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

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

1. $T_R = 0$, $T_A = 1$, $\kappa_0 = \kappa_1 = 0.2$, $\varrho = c = 1$, $\omega = 2\pi$

2. $T_R = 0$, $T_A = 1$, $\kappa_0 = 0.2$, $\kappa_1 = 0.01$, $\varrho = c = 1$, $\omega = 2\pi$

3. $T_R = 0$, $T_A = 1$, $\kappa_0 = 0.2$, $\kappa_1 = 0.001$, $\varrho = c = 1$, $\omega = 2\pi$

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

5. As above, but $\kappa_0 = 12.3$ K^{-1}Ns^{-1} and $\kappa_1 = 10^4$ K^{-1}Ns^{-1}

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

1.4 Creating more complex domains

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

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

1.4.1 Built-in mesh generation tools

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

Python code

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

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

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

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

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

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

1.4.2 *Transforming mesh coordinates*

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

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

toward $x = a$, or

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

toward $x = b$

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

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

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

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

Python code

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

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

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

x_bar, y_bar = denser(x, y)
xy_bar_coor = numpy.array([x_bar, y_bar]).transpose()
mesh.coordinates()[:] = xy_bar_coor
```
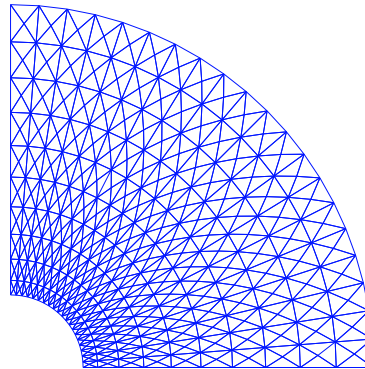
Figure 1.8: A hollow cylinder generated by mapping a rectangular mesh, stretched toward the left side.


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

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

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

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

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

1.5 Handling domains with different materials

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

1.5.1 Working with two subdomains

Suppose we want to solve

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

in a domain Ω consisting of two subdomains where k takes on a different value in each subdomain. For simplicity, yet without loss of generality, we choose for the current implementation the domain
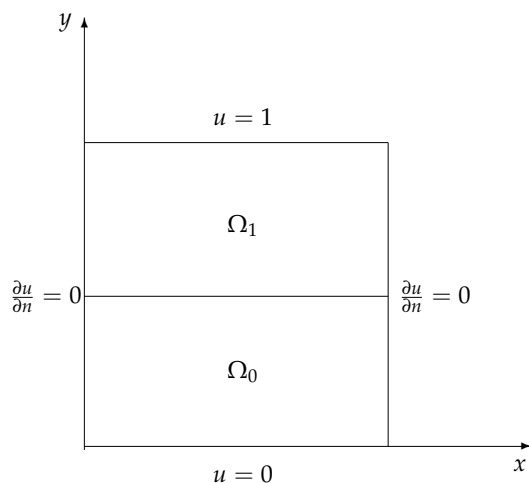
Figure 1.9: Sketch of a Poisson problem with a variable coefficient that is constant in each of the two subdomains Ω_0 and Ω_1.

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

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

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

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

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

Physically, the present problem may correspond to heat conduction, where the heat conduction in Ω_1 is ten times more efficient than in Ω_0. An alternative interpretation is flow in porous media with two geological layers, where the layers' ability to transport the fluid differs by a factor of 10.

1.5.2 Implementation

The new functionality in this subsection regards how to define the subdomains Ω_0 and Ω_1. For this purpose we need to use subclasses of class SubDomain, not only plain functions as we have used so far for specifying boundaries. Consider the boundary function

Python code

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

for defining the boundary $x = 0$. Instead of using such a stand-alone function, we can create an instance[2] of a subclass of SubDomain, which implements the inside method as an alternative to the

---

[2]The term *instance* means a Python object of a particular type (such as SubDomain, Function, FunctionSpace, etc.). Many use *instance* and *object* as interchangeable terms. In other computer programming languages one may also use the term *variable* for

`boundary` function:

Python code

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

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

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

Python code

```python
class Omega0(SubDomain):
    def inside(self, x, on_boundary):
        return True if x[1] <= 0.5 else False

class Omega1(SubDomain):
    def inside(self, x, on_boundary):
        return True if x[1] >= 0.5 else False
```

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

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

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

Since we need to define subdomains of Ω in the present example, we must make use of a `MeshFunction` over cells. The `MeshFunction` constructor is fed with three arguments: 1) the type of value: `"int"` for integers, `"uint"` for positive (unsigned) integers, `"double"` for real numbers, and `"bool"` for logical values; 2) a `Mesh` object, and 3) the topological dimension of the mesh entity in question: cells have topological dimension equal to the number of space dimensions in the PDE problem, and facets have one dimension lower. Alternatively, the constructor can take just a filename and initialize the `MeshFunction` from data in a file.

We start with creating a `MeshFunction` whose values are non-negative integers (`"uint"`) for numbering the subdomains. The mesh entities of interest are the cells, which have dimension 2 in a two-dimensional problem (1 in 1D, 3 in 3D). The appropriate code for defining the `MeshFunction` for two subdomains then reads

Python code

```python
subdomains = MeshFunction("uint", mesh, 2)
# Mark subdomains with numbers 0 and 1
subdomain0 = Omega0()
subdomain0.mark(subdomains, 0)
subdomain1 = Omega1()
subdomain1.mark(subdomains, 1)
```

the same thing. We mostly use the well-known term *object* in this text.

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

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

Python code

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

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

Python code

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

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

Python code

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

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

Having the `k` function ready for finite element computations, we can proceed in the normal manner with defining essential boundary conditions, as in Section 1.1.14, and the $a(u, v)$ and $L(v)$ forms, as in Section 1.1.10. All the details can be found in the file `mat2_p2D.py`.

1.5.3 *Multiple Neumann, Robin, and Dirichlet conditions*

Let us go back to the model problem from Section 1.1.14 where we had both Dirichlet and Neumann conditions. The term `v*g*ds` in the expression for `L` implies a boundary integral over the complete boundary, or in FEniCS terms, an integral over all exterior facets. However, the contributions from the parts of the boundary where we have Dirichlet conditions are erased when the linear system is modified by the Dirichlet conditions. We would like, from an efficiency point of view, to integrate `v*g*ds` only over the parts of the boundary where we actually have Neumann conditions. And more importantly, in other problems one may have different Neumann conditions or other conditions like the Robin type condition. With the mesh function concept we can mark different parts of the boundary and integrate over specific parts. The same concept can also be used to treat multiple Dirichlet conditions. The forthcoming text illustrates how this is done.

Essentially, we still stick to the model problem from Section 1.1.14, but replace the Neumann condition at $y = 0$ by a *Robin condition*[3]:

$$-\frac{\partial u}{\partial n} = p(u - q), \tag{1.130}$$

where p and q are specified functions. Since we have prescribed a simple solution in our model problem, $u = 1 + x^2 + 2y^2$, we adjust p and q such that the condition holds at $y = 0$. This implies that $q = 1 + x^2 + 2y^2$ and p can be arbitrary (the normal derivative at $y = 0$: $\partial u / \partial n = -\partial u / \partial y = -4y = 0$).

Now we have four parts of the boundary: Γ_N which corresponds to the upper side $y = 1$, Γ_R which corresponds to the lower part $y = 0$, Γ_0 which corresponds to the left part $x = 0$, and Γ_1 which corresponds to the right part $x = 1$. The complete boundary-value problem reads

$$-\Delta u = -6 \text{ in } \Omega, \tag{1.131}$$

$$u = u_L \text{ on } \Gamma_0, \tag{1.132}$$

$$u = u_R \text{ on } \Gamma_1, \tag{1.133}$$

$$-\frac{\partial u}{\partial n} = p(u - q) \text{ on } \Gamma_R, \tag{1.134}$$

$$-\frac{\partial u}{\partial n} = g \text{ on } \Gamma_N. \tag{1.135}$$

The involved prescribed functions are $u_L = 1 + 2y^2$, $u_R = 2 + 2y^2$, $q = 1 + x^2 + 2y^2$, p is arbitrary, and $g = -4y$.

Integration by parts of $-\int_\Omega v \Delta u \, \mathrm{d}x$ becomes as usual

$$-\int_\Omega v \Delta u \, \mathrm{d}x = \int_\Omega \nabla u \cdot \nabla v \, \mathrm{d}x - \int_{\partial \Omega} \frac{\partial u}{\partial n} v \, \mathrm{d}s. \tag{1.136}$$

The boundary integral vanishes on $\Gamma_0 \cup \Gamma_1$, and we split the parts over Γ_N and Γ_R since we have different conditions at those parts:

$$-\int_{\partial \Omega} v \frac{\partial u}{\partial n} \, \mathrm{d}s = -\int_{\Gamma_N} v \frac{\partial u}{\partial n} \, \mathrm{d}s - \int_{\Gamma_R} v \frac{\partial u}{\partial n} \, \mathrm{d}s = \int_{\Gamma_N} vg \, \mathrm{d}s + \int_{\Gamma_R} vp(u - q) \, \mathrm{d}s. \tag{1.137}$$

The weak form then becomes

$$\int_\Omega \nabla u \cdot \nabla v \, \mathrm{d}x + \int_{\Gamma_N} gv \, \mathrm{d}s + \int_{\Gamma_R} p(u - q)v \, \mathrm{d}s = \int_\Omega fv \, \mathrm{d}x, \tag{1.138}$$

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

$$\int_{\Gamma_R} p(u - q)v \, \mathrm{d}s = \int_{\Gamma_R} puv \, \mathrm{d}s - \int_{\Gamma_R} pqv \, \mathrm{d}s. \tag{1.139}$$

---

[3]The Robin condition is most often used to model heat transfer to the surroundings and arise naturally from Newton's cooling law.

We then have

$$a(u, v) = \int_\Omega \nabla u \cdot \nabla v \, dx + \int_{\Gamma_R} p u v \, ds, \tag{1.140}$$

$$L(v) = \int_\Omega f v \, dx - \int_{\Gamma_N} g v \, ds + \int_{\Gamma_R} p q v \, ds. \tag{1.141}$$

A natural starting point for implementation is the file `stationary/poisson/dn2_p2D.py`. The new aspects are

1. definition of a mesh function over the boundary,

2. marking each side as a subdomain, using the mesh function,

3. splitting a boundary integral into parts.

Task 1 makes use of the `MeshFunction` object, but contrary to Section 1.5.2, this is not a function over cells, but a function over cell facets. The topological dimension of cell facets is one lower than the cell interiors, so in a two-dimensional problem the dimension becomes 1. In general, the facet dimension is given as `mesh.topology().dim()-1`, which we use in the code for ease of direct reuse in other problems. The construction of a `MeshFunction` object to mark boundary parts now reads

Python code

```python
boundary_parts = \
  MeshFunction("uint", mesh, mesh.topology().dim()-1)
```

As in Section 1.5.2 we use a subclass of `SubDomain` to identify the various parts of the mesh function. Problems with domains of more complicated geometries may set the mesh function for marking boundaries as part of the mesh generation. In our case, the $y = 0$ boundary can be marked by

Python code

```python
class LowerRobinBoundary(SubDomain):
    def inside(self, x, on_boundary):
        tol = 1E-14   # tolerance for coordinate comparisons
        return on_boundary and abs(x[1]) < tol

Gamma_R = LowerRobinBoundary()
Gamma_R.mark(boundary_parts, 0)
```

The code for the $y = 1$ boundary is similar and is seen in `dnr_p2D`.

The Dirichlet boundaries are marked similarly, using subdomain number 2 for Γ_0 and 3 for Γ_1:

Python code

```python
class LeftBoundary(SubDomain):
    def inside(self, x, on_boundary):
        tol = 1E-14   # tolerance for coordinate comparisons
        return on_boundary and abs(x[0]) < tol

Gamma_0 = LeftBoundary()
Gamma_0.mark(boundary_parts, 2)

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

Gamma_1 = RightBoundary()
Gamma_1.mark(boundary_parts, 3)
```

Specifying the `DirichletBC` objects may now make use of the mesh function (instead of a `SubDomain` subclass object) and an indicator for which subdomain each condition should be applied to:

Python code

```
u_L = Expression("1 + 2*x[1]*x[1]")
u_R = Expression("2 + 2*x[1]*x[1]")
bcs = [DirichletBC(V, u_L, boundary_parts, 2),
       DirichletBC(V, u_R, boundary_parts, 3)]
```

Some functions need to be defined before we can go on with the `a` and `L` of the variational problem:

Python code

```
g = Expression("-4*x[1]")
q = Expression("1 + x[0]*x[0] + 2*x[1]*x[1]")
p = Constant(100)  # arbitrary function can go here
u = TrialFunction(V)
v = TestFunction(V)
f = Constant(-6.0)
```

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

The variational problem can be defined as

Python code

```
a = inner(nabla_grad(u), nabla_grad(v))*dx + p*u*v*ds(0)
L = f*v*dx - g*v*ds(1) + p*q*v*ds(0)
```

For the `ds(0)` and `ds(1)` symbols to work we must obviously connect them (or `a` and `L`) to the mesh function marking parts of the boundary. This is done by a certain keyword argument to the `assemble` function:

Python code

```
A = assemble(a, exterior_facet_domains=boundary_parts)
b = assemble(L, exterior_facet_domains=boundary_parts)
```

Then essential boundary conditions are enforced, and the system can be solved in the usual way:

Python code

```
for bc in bcs: bc.apply(A, b)
u = Function(V)
U = u.vector()
solve(A, U, b)
```

The complete code is in the `dnr_p2D.py` file in the `stationary/poisson` directory.

1.6   More examples

Many more topics could be treated in a FEniCS tutorial, e.g., how to solve systems of PDEs, how to work with mixed finite element methods, how to create more complicated meshes and mark boundaries, and how to create more advanced visualizations. However, to limit the size of this tutorial,

the examples end here. There are, fortunately, a rich set of FEniCS demos. The FEniCS documentation explains a collection of PDE solvers in detail: the Poisson equation, the mixed formulation for the Poisson equation, the Biharmonic equation, the equations of hyperelasticity, the Cahn-Hilliard equation, and the incompressible Navier–Stokes equations. Both Python and C++ versions of these solvers are explained. An eigenvalue solver is also documented. In the `dolfin/demo` directory of the DOLFIN source code tree you can find programs for these and many other examples, including the advection-diffusion equation, the equations of elastodynamics, a reaction-diffusion equation, various finite element methods for the Stokes problem, discontinuous Galerkin methods for the Poisson and advection-diffusion equations, and an eigenvalue problem arising from electromagnetic waveguide problem with Nédélec elements. There are also numerous demos on how to apply various functionality in FEniCS, e.g., mesh refinement and error control, moving meshes (for ALE methods), computing functionals over subsets of the mesh (such as lift and drag on bodies in flow), and creating separate subdomain meshes from a parent mesh.

The project CBC.Solve (`https://launchpad.net/cbc.solve`) offers more complete PDE solvers for the Navier–Stokes equations (Chapter 29), the equations of hyperelasticity (Chapter 27), fluid–structure interaction (Chapter 29), viscous mantle flow (Chapter 31), and the bidomain model of electrophysiology. Another project, CBC.RANS (`https://launchpad.net/cbc.rans`), offers an environment for very flexible and easy implementation of Navier–Stokes solvers and turbulence (Mortensen et al., 2011b,a). For example, CBC.RANS contains an elliptic relaxation model for turbulent flow involving 18 nonlinear PDEs. FEniCS proved to be an ideal environment for implementing such complicated PDE models. The easy construction of systems of nonlinear PDEs in CBC.RANS has been further generalized to simplify the implementation of large systems of nonlinear PDEs in general. The functionality is found in the CBC.PDESys package (`https://launchpad.net/cbcpdesys`).

1.7 Miscellaneous topics

1.7.1 Glossary

Below we explain some key terms used in this tutorial.

FEniCS: name of a software suite composed of many individual software components (see `fenicsproject.org`). Some components are DOLFIN and Viper, explicitly referred to in this tutorial. Others are FFC and FIAT, heavily used by the programs appearing in this tutorial, but never explicitly used from the programs.

DOLFIN: a FEniCS component, more precisely a C++ library, with a Python interface, for performing important actions in finite element programs. DOLFIN makes use of many other FEniCS components and many external software packages.

Viper: a FEniCS component for quick visualization of finite element meshes and solutions.

UFL: a FEniCS component implementing the *unified form language* for specifying finite element forms in FEniCS programs. The definition of the forms, typically called a and L in this tutorial, must have legal UFL syntax. The same applies to the definition of functionals (see Section 1.1.11).

Class (Python): a programming construction for creating objects containing a set of variables and functions. Most types of FEniCS objects are defined through the class concept.

Instance (Python): an object of a particular type, where the type is implemented as a class. For instance, `mesh = UnitInterval(10)` creates an instance of class `UnitInterval`, which is reached by the name `mesh`. (Class `UnitInterval` is actually just an interface to a corresponding C++ class in the DOLFIN C++ library.)

Class method (Python): a function in a class, reached by dot notation: `instance_name.method_name`.

`self` parameter (Python): required first parameter in class methods, representing a particular object of the class. Used in method definitions, but never in calls to a method. For example, if `method(self, x)` is the definition of `method` in a class `Y`, `method` is called as `y.method(x)`, where `y` is an instance of class `Y`. In a call like `y.method(x)`, `method` is invoked with `self=y`.

Class attribute (Python): a variable in a class, reached by dot notation: `instance_name.attribute_name`.

1.7.2 Overview of objects and functions

Most classes in FEniCS have an explanation of the purpose and usage that can be seen by using the general documentation command `pydoc` for Python objects. You can type

Output

```
pydoc dolfin.X
```

to look up documentation of a Python class `X` from the DOLFIN library (`X` can be `UnitSquare`, `Function`, `FunctionSpace`, etc.). Below is an overview of the most important classes and functions in FEniCS programs, in the order they typically appear within programs.

`UnitSquare(nx, ny)`: generate mesh over the unit square $[0,1] \times [0,1]$ using `nx` divisions in x direction and `ny` divisions in y direction. Each of the `nx*ny` squares are divided into two cells of triangular shape.

`UnitInterval`, `UnitCube`, `UnitCircle`, `UnitSphere`, `Interval`, `Rectangle`, and `Box`: generate mesh over domains of simple geometric shape, see Section 1.4.

`FunctionSpace(mesh, element_type, degree)`: a function space defined over a mesh, with a given element type (e.g., `"Lagrange"` or `"DG"`), with basis functions as polynomials of a specified degree.

`Expression(formula, p1=v1, p2=v2, ...)`: a scalar- or vector-valued function, given as a mathematical expression `formula` (string) written in C++ syntax. The spatial coordinates in the expression are named `x[0]`, `x[1]`, and `x[2]`, while time and other physical parameters can be represented as symbols `p1`, `p2`, etc., with corresponding values `v1`, `v2`, etc., initialized through keyword arguments. These parameters become attributes, whose values can be modified when desired.

`Function(V)`: a scalar- or vector-valued finite element field in the function space `V`. If `V` is a `Function Space` object, `Function(V)` becomes a scalar field, and with `V` as a `VectorFunctionSpace` object, `Function(V)` becomes a vector field.

`SubDomain`: class for defining a subdomain, either a part of the boundary, an internal boundary, or a part of the domain. The programmer must subclass `SubDomain` and implement the `inside(self, x, on_boundary)` function (see Section 1.1.3) for telling whether a point `x` is inside the subdomain or not.

`Mesh`: class for representing a finite element mesh, consisting of cells, vertices, and optionally faces, edges, and facets.

`MeshFunction`: tool for marking parts of the domain or the boundary. Used for variable coefficients ("material properties", see Section 1.5.1) or for boundary conditions (see Section 1.5.3).

`DirichletBC(V, value, where)`: specification of Dirichlet (essential) boundary conditions via a function space `V`, a function `value(x)` for computing the value of the condition at a point `x`, and a specification `where` of the boundary, either as a `SubDomain` subclass instance, a plain function, or as a `MeshFunction` instance. In the latter case, a 4th argument is provided to describe which subdomain number that describes the relevant boundary.

`TrialFunction(V)`: define a trial function on a space `V` to be used in a variational form to represent the unknown in a finite element problem.

`TestFunction(V)`: define a test function on a space `V` to be used in a variational form.

`assemble(X)`: assemble a matrix, a right-hand side, or a functional, given a from `X` written with UFL syntax.

`assemble_system(a, L, bcs)`: assemble the matrix and the right-hand side from a bilinear (a) and linear (L) form written with UFL syntax. The `bcs` parameter holds one or more `DirichletBC` objects.

`LinearVariationalProblem(a, L, u, bcs)`: define a variational problem, given a bilinear (a) and linear (L) form, written with UFL syntax, and one or more `DirichletBC` objects stored in `bcs`.

`LinearVariationalSolver(problem)`: create solver object for a a linear variational problem object (`problem`).

`solve(A, U, b)`: solve a linear system with `A` as coefficient matrix (`Matrix` object), `U` as unknown (`Vector` object), and `b` as right-hand side (`Vector` object). Usually, `U = u.vector()`, where `u` is a `Function` object representing the unknown finite element function of the problem, while `A` and `b` are computed by calls to `assemble` or `assemble_system`.

`plot(q)`: quick visualization of a mesh, function, or mesh function `q`, using the Viper component in FEniCS.

`interpolate(func, V)`: interpolate a formula or finite element function `func` onto the function space `V`.

`project(func, V)`: project a formula or finite element function `func` onto the function space `V`.

1.7.3 *User-defined functions*

When defining a function in terms of a mathematical expression inside a string formula, e.g.,

Python code

```python
myfunc = Expression("sin(x[0])*cos(x[1])")
```

the expression contained in the first argument will be turned into a C++ function and compiled to gain efficiency. Therefore, the syntax used in the expression must be valid C++ syntax. Most Python syntax for mathematical expressions are also valid C++ syntax, but power expressions make an exception: `p**a` must be written as `pow(p,a)` in C++ (this is also an alternative Python syntax). The following mathematical functions can be used directly in C++ expressions when defining `Expression` objects: `cos`, `sin`, `tan`, `acos`, `asin`, `atan`, `atan2`, `cosh`, `sinh`, `tanh`, `exp`, `frexp`, `ldexp`, `log`, `log10`, `modf`, `pow`, `sqrt`, `ceil`, `fabs`, `floor`, and `fmod`. Moreover, the number π is available as the symbol `pi`. All the listed functions are taken from the `cmath` C++ header file, and one may hence consult documentation of `cmath` for more information on the various functions.

1.7.4 *Linear solvers and preconditioners*

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

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

Possible choices of preconditioners include

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

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

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

Python code

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

1.7.5 Installing FEniCS

The FEniCS software components are available for Linux, Windows and Mac OS X platforms. Detailed information on how to get FEniCS running on such machines are available at the `fenicsproject.org` website. Here are just some quick descriptions and recommendations by the author.

To make the installation of FEniCS as painless and reliable as possible, the reader is strongly recommended to use Ubuntu Linux[4]. Any standard PC can easily be equipped with Ubuntu Linux, which may live side by side with either Windows or Mac OS X or another Linux installation. Basically, you download Ubuntu from `http://www.ubuntu.com/getubuntu/download`, burn the file on a CD or copy it to a memory stick, reboot the machine with the CD or memory stick, and answer some usually straightforward questions (if necessary). On Windows, Wubi is a tool that automatically installs Ubuntu on the machine. Just give a user name and password for the Ubuntu installation, and Wubi performs the rest. The graphical user interface (GUI) of Ubuntu is quite similar to both Windows 7 and Mac OS X, but to be efficient when doing science with FEniCS this author recommends to run programs in a terminal window and write them in a text editor like Emacs or Vim. You can employ integrated development environment such as Eclipse, but intensive FEniCS developers and users tend to find terminal windows and plain text editors more user friendly.

Instead of making it possible to boot your machine with the Linux Ubuntu operating system, you can run Ubuntu in a separate window in your existing operation system. There are several solutions to chose among: the free *VirtualBox* and *VMWare Player*, or the commercial tools *VMWare Fusion* and *Parallels* (just search for the names to download the programs).

Once the Ubuntu window is up and running, FEniCS is painlessly installed by

---

[4]Even though Mac users now can get FEniCS by a one-click install, I recommend using Ubuntu on Mac, unless you have high Unix competence and much experience with compiling and linking C++ libraries on Mac OS X.

```
sudo apt-get install fenics
```

Sometimes the FEniCS software in a standard Ubuntu installation lacks some recent features and bug fixes. Visiting the detailed download page on `fenicsproject.org` and copying a few Unix commands is all you have to do to install a newer version of the software.

1.7.6   Books on the finite element method

There are a large number of books on the finite element method. The books typically fall in either of two categories: the abstract mathematical version of the method and the engineering "structural analysis" formulation. FEniCS builds heavily on concepts in the abstract mathematical exposition. An easy-to-read book, which provides a good general background for using FEniCS, is Gockenbach (2006). The book Donea and Huerta (2003) has a similar style, but aims at readers with interest in fluid flow problems. Hughes (1987) is also highly recommended, especially for those interested in solid mechanics and heat transfer applications.

Readers with background in the engineering "structural analysis" version of the finite element method may find Bickford (1994) as an attractive bridge over to the abstract mathematical formulation that FEniCS builds upon. Those who have a weak background in differential equations in general should consult a more fundamental book, and Eriksson et al. (1996) is a very good choice. On the other hand, FEniCS users with a strong background in mathematics and interest in the mathematical properties of the finite element method, will appreciate the texts Brenner and Scott (2008), Braess (2007), Ern and Guermond (2004), Quarteroni and Valli (2008), or Ciarlet (2002).

1.7.7   Books on Python

Two very popular introductory books on Python are "Learning Python" (Lutz, 2007) and "Practical Python" (Hetland, 2002). More advanced and comprehensive books include "Programming Python" (Lutz, 2006), and "Python Cookbook" (Martelli and Ascher, 2005) and "Python in a Nutshell" (Martelli, 2006). The web page `http://wiki.python.org/moin/PythonBooks` lists numerous additional books. Very few texts teach Python in a mathematical and numerical context, but the references Langtangen (2008, 2011); Kiusalaas (2009) are exceptions.