Introduction to Finite Element Programming - The FEniCS Tutorial

Hans Petter Langtangen[1,2] (hpl@simula.no)

Anders Logg[3,1] (logg@chalmers.se)

[1]Center for Biomedical Computing, Simula Research Laboratory

[2]Department of Informatics, University of Oslo

[3]Department of Mathematics, Chalmers University of Technology

This document 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 since it actually gives high performance. After having digested the examples in this tutorial, the reader should be able to learn more from the FEniCS documentation, the numerous demo programs that come with the software, and the comprehensive FEniCS book *Automated Solution of Differential Equations by the Finite element Method* [15]. This tutorial is a further development of the opening chapter in [15].

Watch out for shortcomings!

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

[a] `mailto:logg@chalmers.se`

Mar 28, 2016

This document is also available in Sphinx[1] and Bootstrap[2] format.

[1] http://hplgit.github.io/fenics-tutorial/doc/pub/sphinx/
[2] http://hplgit.github.io/fenics-tutorial/doc/pub/ftut.html

Contents

Preface

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 2.6.5 explains briefly how to install the necessary tools.

hpl 1: Rethink how to organize program examples! **hpl 2**: Drop transient and stationary.

All the examples discussed in the following are available as executable Python source code files in a directory tree[3]. File paths reflect the nature of the PDE problem being solved. For example, `poisson/p2D_iter.py` has a descriptive directory path and a very brief Unix-style filename (here `p2D` for Poisson 2D problem, and `_iter` for a version with iterative linear solvers).

The FEniCS version. This document is up-to-date with FEniCS version 1.6. To see which version you have, run the following command in a Unix/Linux terminal window if you run Python version 2.7:

Terminal

```
Terminal> python -c 'import dolfin; print dolfin.__version__'
1.6.0
```

In Python version 3.x you must write

Terminal

```
Terminal> python -c 'import dolfin; print(dolfin.__version__)'
1.6.0
```

The Python version. Python comes in two versions, 2 and 3, and these are not compatible. FEniCS has a code base that runs under both versions. All the programs in this tutorial are also developed such that they can be run under both Python 2 and 3. Programs that need to print must then start with

```
from __future__ import print_function
```

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

Acknowledgments. We thank Johan Hake, Kent-Andre Mardal, and Kristian Valen-Sendstad for promptly answering all questions about FEniCS functionality and for implementing all requests when preparing the first version of this tutorial for the FEniCS book [15]. We are particularly thankful to Professor Douglas Arnold for very valuable feedback on early versions of the text. Øystein Sørensen pointed out a lot of typos and contributed with many helpful comments.

[3] `https://github.com/hplgit/fenics-tutorial/blob/master/src`

Many errors and typos were also reported by Mauricio Angeles, Ida Drøsdal, Miroslav Kuchta, Hans Ekkehard Plesser, Marie Rognes, and Hans Joachim (Achim) Scroll. Ekkehard Ellmann as well as two anonymous reviewers provided a series of suggestions and improvements.

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

0.1 To-do list

Remember: cannot exceed 150 pages (as reported at the end of `ftut.log`). Solutions to exercises will not appear in the printed book. If we run out of pages, we can also remove the exercises by putting if-else constructs around them. There will be one short printed tutorial and then extended e-versions with exercises (and optionally solution) on our github web site.

Regarding layout, we must use `svmono.cls` for the printed book, but are allowed to use gray background in code boxes and lmodern instead of Courier for monospace font. Springer's official ebook has exactly the same layout. For all other versions on our github web site, we can choose whatever layout we want.

- Reorganize sections into chapters. (HPL)
- Rethink how to organize example programs in directories. Find a naming convention. (AL) HPL suggestion: Drop the `stationary` and `transient` directories, have at most separate directories for each PDE.
- Programs are now flat demos. Educate the reader with better software engineering habits: functions, classes, unit tests. Avoid copy-and-edit flat programs implied by today's collection. (HPL)
- Compute maximum error already in the first example so we can create a unit test as early as possible. (Done, HPL)
- Find successful exercises from various tutorials (AL) and add as exercises in the book (HPL/AL). Exercises are key for learning software so having them (in an extended version?) sends an important signal about their relevance.
- Write about installation in a way that does not get outdated. More as a guide to a newcomer: What to choose? (AL)
- Multi-boundary examples do not work with FEniCS 1.6: `dnr*.py`. (AL)
- `membrane2.py` referred to, but it is still in sandbox. (HPL)
- Meshr domains. Set up some common continuum mechanics examples first. (HPL/AL)

- Decide on an elasticity problem.
- Simple Navier-Stokes solver. Do backward facing step and flow around a cylinder.
- According to the `plot` doc string, it should be easy to plot the element a la `plot(u.function_space().ufl_element())` but I did not get this to work. Not crucial, but plotting the element is a nice feature :-)
- Show how the definition of boundaries can be done via strings compiled to C++ (as soon as we have an example with non-trivial boundary segments), cf. Navier-Stokes FEniCS demo.
- Can we change the value of `DOLFIN_EPS`? `imort dolfin; dolfin.DOLFIN_EPS=...` will work, but then all modules in the simulator must do `import dolfin`. Note that its value is very strict, e.g., `10.1+10.2` has rounding $3.5 10^{-15}$, so `DOLFIN_EPS` is strictly for scaled problems only, where all variables are in $[0, 1]$.
- Comment regarding FEniCS demos: The documented demos mention a lot of packages: DOLFIN, FFC, Fiat, ... Make sure the reader of the tutorial does not get lost in the jungle of packages and make sure the names are explained somewhere in the text such that the tutorial is a good background for understanding every demo in every detail.
- Specifications of boundaries (`SubDomain`) can be done by a C++ string as in `DirichletBC(V, (0,0), 'on_boundary && x[0] < DOLFIN_EPS')`. This is an important performance enhancement for large meshes in time dependent problems so it should be documented.

0.1.1 HPL questions

Iterative linear solvers info. We can get this printed out on the screen, but is there any method to extract this text inside the program, such that we can see how many Krylov iterations we do etc.? Any way for Python to capture the standard output stream in C++?

Easy to write a script that post-processes the output, but we have to wait until the simulator has terminated, or we can pipe to a script `process.py` that treats the output in some desired way (could append some into to a file and that is reopened by the simulator):

Terminal

```
Terminal> python mysolver.py | process.py
```

where the simplest `process.py` is

```
import sys, time
t0 = time.clock()
while True:
    line = sys.stdin.readline()
    if not line:
        break
    t1 = time.clock()
    print 'after %g seconds: %s' % (t1-t0, line.rstrip())
print 'END'
```

Chapter 1

The fundamentals

1.1 Mathematical problem formulation

hpl 3: Need a little intro.

1.1.1 The Poisson equation

Let us start with a "Hello, World!" program in the world of PDEs - it must be a program that solves the Laplace or Poisson equation. Our first example regards the following Poisson problem,

$$-\nabla^2 u(\boldsymbol{x}) = f(\boldsymbol{x}), \quad \boldsymbol{x} \text{ in } \Omega, \tag{1.1}$$

$$u(\boldsymbol{x}) = u_0(\boldsymbol{x}), \quad \boldsymbol{x} \text{ on } \partial\Omega\,. \tag{1.2}$$

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

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

$$-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} = f(x, y)\,. \tag{1.3}$$

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

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

Solving a 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 the spatial domain Ω.
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, but getting and reading a proper book on the finite element method in addition is encouraged. Section 2.6.6 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,

$$-\int_\Omega (\nabla^2 u)v \,\mathrm{d}x = \int_\Omega fv \,\mathrm{d}x\,. \tag{1.4}$$

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

$$-\int_\Omega (\nabla^2 u)v \,\mathrm{d}x = \int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x - \int_{\partial\Omega} \frac{\partial u}{\partial n} v \,\mathrm{d}s, \tag{1.5}$$

where $\frac{\partial u}{\partial n}$ is the derivative of u in the outward normal direction at 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 (1.5) therefore vanishes. From (1.4) and (1.5) it follows that

$$\int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x = \int_\Omega f v \,\mathrm{d}x \,. \tag{1.6}$$

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 (1.6) as the *weak form* or *variational form* of the original boundary-value problem (1.1)-(1.2).

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

$$\int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x = \int_\Omega f v \,\mathrm{d}x \quad \forall v \in \hat{V}. \tag{1.7}$$

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

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

In short, $H^1(\Omega)$ is the mathematically well-known Sobolev space containing functions v such that v^2 and $||\nabla v||^2$ have finite integrals over Ω. 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 (1.7), 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 (1.7) to a discrete variational problem. This is done by introducing *finite-dimensional* test and trial spaces, often denoted as $\hat{V}_h \subset \hat{V}$ and $V_h \subset V$. The discrete variational problem reads: Find $u_h \in V_h \subset V$ such that

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

The choice of $\hat{V}_h$ and 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 $\hat{V}_h$ and 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.

What we mean by the notation u and V.

The mathematics literature on variational problems writes u_h for the solution of the discrete problem and u for the solution of the continuous problem. To obtain (almost) a one-to-one relationship between the mathematical formulation of a problem and the corresponding FEniCS program,

we shall 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 $\hat{V}$, without adding any subscript to V or $\hat{V}$. This restriction of V simply 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 the following unified notation for linear weak forms:

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

In the present problem we have that

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

$$L(v) = \int_\Omega fv \,\mathrm{d}x\,. \tag{1.11}$$

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:

- Turn the PDE problem into a discrete variational problem: find $u \in V$ such that $a(u,v) = L(v) \quad \forall v \in \hat{V}$.
- Specify the choice of spaces (V and $\hat{V}$), which means specifying the mesh and type of finite elements.

1.2 A basic Poisson solver

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 $u_{\mathrm{e}}(x, y)$:

$$u_{\mathrm{e}}(x, y) = 1 + x^2 + 2y^2 \tag{1.12}$$

on some 2D domain. By inserting (1.12) in our Poisson problem, we find that $u_{\mathrm{e}}(x, y)$ is a solution if

$$f(x, y) = -6, \quad u_0(x, y) = u_{\mathrm{e}}(x, y) = 1 + x^2 + 2y^2,$$

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 (1.12) 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 (u) with the exact solution (u_{e}). These quantities should be equal to machine precision *at the nodes*.

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

The classical way of testing a program is to compare the numerical solution with an exact analytical solution of the test problem and conclude that the program works if the error is "small enough". Unfortunately, it is impossible to tell if an error 10^{-5} on a 20×20 mesh of P1 elements is just all the numerical approximation errors in the method or if this error also contains the effect of a bug in the code. All we usually know about the numerical error is its *asymptotic properties*, for instance that it goes like h^2 if h is the size of a cell in the mesh. Then we can compare the error on meshes with different h values to see if the asymptotic behavior is correct. This is a very powerful verification technique and is explained in detail in Section 1.5.2. However, if we have a test problem where we know that there are no numerical approximation errors, we know that the analytical solution of the PDE problem should be reproduced to machine precision by the program. That is why we emphasize this kind of test problems throughout this tutorial. Typically, elements of degree d can reproduce polynomials of degree $d + 1$ exactly, so this is the starting point for constructing a solution without numerical approximation errors. Then we fit the data in the problem (like u_0 and f) to this solution.

1.2.1 A simple Poisson solver

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

```
from dolfin import *

# Create mesh and define function space
mesh = UnitSquareMesh(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)

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

# Find max error
u0_Function = interpolate(u0, V)         # exact solution
u0_array = u0_Function.vector().array()  # dof values
max_error = (u0_array - u.vector().array()).max()
print('max error:', max_error)

# Hold plot
interactive()
```

The complete code can be found in the file `p2D_plain.py`[1] in the directory `src/poisson`[2].

[1] https://github.com/hplgit/fenics-tutorial/blob/master/src/poisson/p2D_plain.py
[2] https://github.com/hplgit/fenics-tutorial/blob/master/src/poisson

1.2.2 Running the program

To run the program `p2D_plain.py`, open a terminal window, move to the directory containing the program and write

Terminal

```
Terminal> python p2D_plain.py
```

A plot window pops up showing how the solution u looks like as a surface. With the left mouse button you can tilt the figure. Click `m` to bring up the underlying mesh. Click `p` to save to a PNG file `dolfin_plot_0.png` and `P` to save to a PDF file `dolfin_plot_1.pdf`. To kill the plot window and terminate the application, click `Ctrl+q` (hold down the `Ctrl` key and press `q`). Figure 1.1 displays the surface and the mesh below. Since u is a simple quadratic function, constructed for testing our solver, the surface looks quite boring.

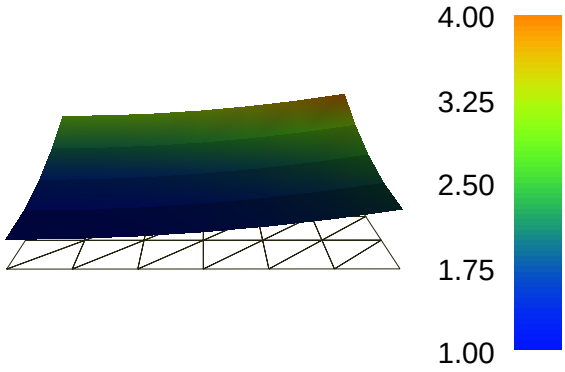


Figure 1.1: Plot of the solution in the first FEniCS example.

1.2.3 Dissection of the program

We shall now dissect this FEniCS program in detail. The program is written in the Python programming language. You may either take a quick look at the official Python tutorial[3] 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. (The requirement of using Python and an abstract mathematical formulation of the finite element problem may seem difficult for

[3] `http://docs.python.org/tutorial/`

those who are unfamiliar with these topics. However, the amount of mathematics and Python that is really demanded to get you productive with FEniCS is quite limited. And Python is an easy-to-learn language that you certainly will love and use far beyond FEniCS programming.) Section 2.6.7 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 compare the numerical and exact solution as well as investigate u visually.

The key import line. The first line in the program,

```
from dolfin import *
```

imports the key classes `UnitSquareMesh`, `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 `from dolfin import *` statement imports other components too, but newcomers to FEniCS programming do not need to care about this.

Generating simple meshes. The statement

```
mesh = UnitSquareMesh(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. The FEniCS program will then read the mesh from file.

Defining a function space corresponding to a mesh. Having a mesh, we can define a discrete function space `V` over this mesh:

```
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" or the P1 element. 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`, which will then generate P2, P3, and so forth, type of elements. Changing the second parameter to `'DG'` creates a function space for discontinuous Galerkin methods.

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

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

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

```
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

```
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. 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]`:

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

Remember that string expressions must have valid C++ syntax.

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

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

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

is implemented as

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

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

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

At any time, parameters can be updated:

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

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

```
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 `True` if `x` is on the physical boundary of the mesh, so in the present case, where we are supposed to return `True` for all points on the boundary, we can just return the supplied value of `on_boundary`. The `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`:

```
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:

```
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
```

Specifying the right-hand side function. Before defining $a(u, v)$ and $L(v)$ we have to specify the f function:

```
f = Expression('-6')
```

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

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

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

```
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 $\nabla u \cdot \nabla v \, \mathrm{d}x$ and $fv \, \mathrm{d}x$. This is a key strength of FEniCS: the formulas in the variational formulation translate directly to very similar Python code, a feature that makes it easy to specify 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) [1, 15] 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 ∇ ("nabla") acts as a vector operator, and therefore this author prefers to always use `nabla_grad`.

Forming and solving the linear system. 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

```
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:

```
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, i.e., 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, 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).

Examining the values of the solution. The present test problem should produce a numerical solution that equals the exact solution to machine precision. That is, there are no approximation errors in our test problem. We can use this property to "prove" that our implementation is correct, a necessary first step before we try to apply our code to more complicated problems. For such verification, we need to compare the computed `u` function to `u0`.

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

$$u = \sum_{j=1}^{N} U_j \phi_j \,. \tag{1.13}$$

By writing `solve(a == L, u, bc)` in the program, a linear system will be formed from a and L, and this system is solved for the $U_1, \ldots, U_N$ values. The $U_1, \ldots, U_N$ values are known as *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 are additional nodes at the facets and maybe also in the interior of cells.

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

```
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:

```
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`. However, `i` corresponds to u at some point in the mesh and the correspondence requires knowledge of the numbering of degrees of freedom and the numbering of vertices in elements in the mesh, see Section 1.3.3 for details.

For now, we want to check that the values in `u_array` are correct: they should equal our `u0` function. The most natural approach is to interpolate our `u0` expression onto our space (i.e., the finite element mesh),

```
u0_Function = interpolate(u0, V)
```

The `interpolate` function returns a `Function` object, whose degrees of freedom values can be obtained by `.vector().array()`. Our goal is to show that the degrees of freedom arrays of `u` and `u0_Function` are equal. One safe of doing this is to compute the maximum error,

```
u0_array = u0_Function.vector().array()  # dof values
max_error = (u0_array - u.vector().array()).max()
print('max error:', max_error)
```

How to check that the error vanishes?

With inexact arithmetics, as we always have on a computer, this maximum error is not zero, but should be a small number. The machine precision is about 10^{-16}, but in finite element calculations, rounding errors of this size may accumulate, so the expected accuracy of `max_error` smaller. Experiments show that increasing the number of elements and increasing the degree of the finite element polynomials increase `max_error`. For a mesh with $2(20 \times 20)$ cubic Lagrange elements (degree 3) `max_error` is about $2 \cdot 10^{-12}$, while for 18 linear elements the maximum error is about $2 \cdot 10^{-15}$.

Plotting the solution. The simplest way of quickly looking at `u` is to say

```
plot(u, interactive=True)
# or
plot(u)
interactive()
```

Clicking on `Help` in the plot windows brings up a list of commands. For example, typing `m` brings up the mesh. With the left, middle, and right mouse buttons you can rotate, translate, and zoom (respectively) the plotted surface to better examine what the solution looks like. You must click `Ctrl+q` to kill the plot window and continue execution beyond the `plot(u, interactive=True)` command or `interactive()`. Figure 1.1 displays the resulting u function.

Plotting both the solution and the mesh is accomplished by

```
plot(u)
plot(mesh)
# Hold plot
interactive()
```

Type `Ctrl+w` to kill all plot windows and continue execution.

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

```
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.

1.2.4 Refactored implementation

Our initial program above is "flat". That is, it is not organized into logical, reusable units in terms of Python functions. Such flat programs are popular for quickly testing out some software, but not well suited for serious problem solving. We shall therefore at once *refactor* the program, meaning that we divide it into functions, but this is just a reordering of the existing statements. During refactoring, we try make functions as reusable as possible in other contexts, but statements specific to a certain problem or task are also encapsulated in (non-reusable) functions. Being able to distinguish reusable code from specialized code is a key issue when refactoring code, and this ability depends on a good mathematical understanding of the problem at hand ("what is general, what is special?"). In a flat program, general and specialized code (and mathematics) is often mixed together.

A general solver function. Some of the code in the previous flat program are needed to solve any Poisson problem $-\nabla^2 u = f$ on $[0,1]\times[0,1]$ with $u = u_0$ on the boundary. Let us collect this code in a reusable function `solver`. Our special test problem will then just be an application of `solver` with some additional statements. We limit the `solver` function to just *compute the numerical solution*. Plotting and comparing the solution with the exact solution are considered to be problem-specific activities to be performed elsewhere.

We parameterize `solver` by f, u_0, and the resolution of the mesh. Since it is so trivial to use higher-order finite element functions by changing the third argument to `FunctionSpace`, we let also the degree of the polynomials in the finite element basis functions be an argument to `solver`. **hpl 4**: The refactoring extends functionality. Should we be strict and keep linear elements? The test is better when it tests the degree parameter as well...

```
from dolfin import *

def solver(f, u0, Nx, Ny, degree=1):
    """
    Solve -Laplace(u)=f on [0,1]x[0,1] with 2*Nx*Ny Lagrange
    elements of specified degree and u=u0 (Expresssion) on
    the boundary.
    """
    # Create mesh and define function space
    mesh = UnitSquareMesh(Nx, Ny)
    V = FunctionSpace(mesh, 'Lagrange', degree)

    def u0_boundary(x, on_boundary):
        return on_boundary

    bc = DirichletBC(V, u0, u0_boundary)

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

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

    return u
```

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

```
def application_test():
```

```
    """Plot the solution in the test problem."""
    u0 = Expression('1 + x[0]*x[0] + 2*x[1]*x[1]')
    f = Constant(-6.0)
    u = solver(f, u0, 6, 4, 1)
    # Dump solution to file in VTK format
    file = File("poisson.pvd")
    file << u
    # Plot solution and mesh
    plot(u)
```

Make a module! The refactored code is put in a file `p2D_func.py`[4]. We should make sure that such a file can be imported (and hence reused) in other programs. Then all statements in the main program should appear with a test `if __name__ == '__main__':`. This test is true if the file is executed as a program, but false if the file is imported. If we want to run this file in the same way as we can run `p2D_func.py`, the main program is simply a call to `application_test()` followed by a call `interactive()` to hold the plot:

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

Verification. The remaining part of our first program is to compare the numerical and the exact solution. Every time we edit the code we must rerun the test and examine that `max_error` is sufficiently small so we know that the code still works. To this end, we shall adopt *unit testing*, meaning that we create a mathematical test and corresponding software that can run all our tests automatically and check that all tests pass. Python has several tools for unit testing. Two very popular ones are pytest and nose. These are almost identical and very easy to use. More classical unit testing with test classes is offered by the built-in tool `unittest`, but here we are going to use pytest (or nose) since it demands shorter and clearer code.

Mathematically, our unit test is that the finite element solution of our problem when $f = -6$ equals the exact solution $u = u_0 = 1 + x^2 + 2y^2$. We have already created code that finds the maximum error in the numerical solution. Because of rounding errors, we cannot demand this maximum error to be zero, but we have to use a tolerance, which depends to the number of elements and the degrees of the polynomials in the finite element basis functions. In Section 1.2.3 we reported some experiments with the size of the maximum error. If we want to test that `solver` works for meshes up to $2(20 \times 20)$ elements and cubic Lagrange elements, 10^{-11} is an appropriate tolerance for testing that the maximum error vanishes.

[4] `https://github.com/hplgit/fenics-tutorial/blob/master/src/poisson/p2D_func.py`

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

- has a name starting with `test_`
- has no arguments
- implements the test as `assert success, msg`

Regarding the last point, `success` is a boolean expression that is `False` if the test fails, and in that case the string `msg` is written to the screen. When the test fails, `assert` raises an `AssertionError` exception in Python, otherwise the statement runs silently. The `msg` string is optional, so `assert success` is the minimal test. In our case, we will do `assert max_error < tol`, where `tol` is the tolerance (10^{-11}) mentioned above.

A proper *test function* for implementing this unit test in the pytest or nose testing frameworks has the following form. Note that we perform the test for different mesh resolutions and degrees of finite elements.

```
def test_solver():
    """Reproduce u=1+x^2+2y^2 to "machine precision"."""
    tol = 1E-11  # This problem's precision
    u0 = Expression('1 + x[0]*x[0] + 2*x[1]*x[1]')
    f = Constant(-6.0)
    for Nx, Ny in [(3,3), (3,5), (5,3), (20,20)]:
        for degree in 1, 2, 3:
            print('solving on 2(%dx%d) mesh with P%d elements'
                  % (Nx, Ny, degree))
            u = solver(f, u0, Nx, Ny, degree)
            # Make a finite element function of the exact u0
            V = u.function_space()
            u0_Function = interpolate(u0, V)  # exact solution
            # Check that dof arrays are equal
            u0_array = u0_Function.vector().array()  # dof values
            max_error = (u0_array - u.vector().array()).max()
            msg = 'max error: %g for 2(%dx%d) mesh and degree=%d' %\
                  (max_error, Nx, Ny, degree)
            assert max_error < tol, msg
```

We can at any time run

Terminal

```
Terminal> py.test -s -v p2D_func.py
```

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

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

Tip: Print messages in test functions.

The `assert` statement runs silently when the test passes so users may become uncertain if all the statements in a test function are really executed. A psychological help is to print out something before `assert` (as we do in the example above) such that it is clear that the test really takes place. (Note that `py.test` needs the `-s` option to show printout from the test functions.)

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 1.4.1. Information in the intermediate sections can be studied on demand.

Exercise 1: Solve a Poisson problem

Solve the following problem

$$\nabla^2 u = 2e^{-2x}\sin(\pi y)((4-5\pi^2)\sin(2\pi x) - 8\pi\cos(2\pi x)), \quad \text{in } \Omega = [0,1]\times[0,1] \tag{1.14}$$

$$u = 0 \quad \text{on } \partial\Omega \tag{1.15}$$

The exact solution is given by

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

Compute the maximum numerical approximation error in a mesh with $2(N_x \times N_y)$ elements and in a mesh with double resolution: $4(N_x \times N_y)$ elements. Show that the doubling the resolution reduces the error by a factor 4 when using Lagrange elements of degree one. (This is a good verification that the implementation is correct, but note that the result requires sufficiently fine mesh - here one may start with $N_x = N_y = 20$.) Make an illustrative plot of the solution too.

a) Base your implementation on editing the program `p2D_plain.py`.

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

FEniCS will abort with a compilation error if you type the expressions in a wrong way syntax-wise. Search for *error:* in the `/very/long/path/compile.log` file mentioned in the error message to see what the C++ compiler reported as error in the expressions.
Filename: `p2D_fsin_plain`.

Solution. Looking at the `p2D_plain.py` code, we realize that the following edits are required:

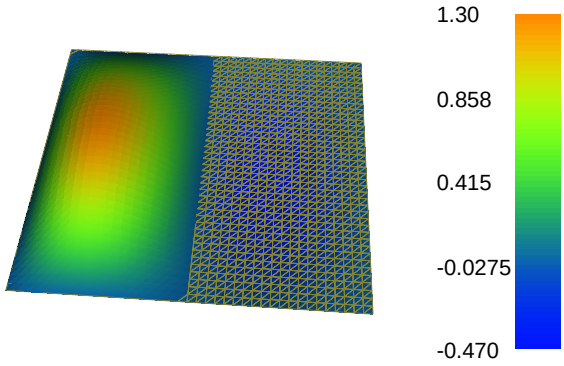
- Modify the `mesh` computation.
- Modify `u0` and `f`.
- Add expression for the exact solution.
- Modify the computation of the numerical error.
- Insert a loop to enable solving the problem twice.
- Put the error reduction computation and the plot statements after the loop.

Here is the modified code:

```
from dolfin import *

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

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

    # Define boundary conditions
    u0 = Constant(0)

    def u0_boundary(x, on_boundary):
        return on_boundary

    bc = DirichletBC(V, u0, u0_boundary)

    # Define variational problem
    u = TrialFunction(V)
    v = TestFunction(V)
```

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

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

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

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

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

# Plot solution and mesh
plot(u)

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

# Hold plot
interactive()
```

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



b) Base your implementation on a new file that imports functionality from the module `p2D_func.py`. Embed the check of the reduction of the numerical approximation error in a unit test. Filename: `p2D_fsin_func`.

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

```
from p2D_func import (
    solver, Expression, Constant, interpolate, File, plot,
    interactive)

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

def test_solver():
    """Check convergence rate of solver."""
    u0, f, u_e = data()
    Nx = 20
    Ny = Nx
    error = []
    # Loop over refined meshes
    for i in range(2):
        Nx *= i+1
        Ny *= i+1
```

```
        print('solving on 2(%dx%d) mesh' % (Nx, Ny))
        u = solver(f, u0, Nx, Ny, degree=1)
        # Make a finite element function of the exact u_e
        V = u.function_space()
        u_e_array = interpolate(u_e, V).vector().array()
        max_error = (u_e_array - u.vector().array()).max()  # Linf norm
        error.append(max_error)
        print('max error:', max_error)
    for i in range(1, len(error)):
        error_reduction = error[i]/error[i-1]
        print('error reduction:', error_reduction)
        assert abs(error_reduction - 0.25) < 0.1

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

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

The unit test is embedded in a proper test function `test_solver` for the pytest or nose testing frameworks. Visualization of the solution is encapsulated in the `application` function. Since we need `u_e`, `u0`, and `f` in two functions, we place the definitions in a function `data` to avoid copies of these expressions.

Remarks. This exercise demonstrates that changing a flat program to solve a new problem requires careful editing of statements scattered around in the file, while the solution in b), based on the `solver` function, requires *no modifications* of the `p2D_func.py` file, just *minimalistic additional new code* in a separate file. The Poisson solver remains in one place (`p2D_func.py`) while in a) we got two Poisson solvers. If you decide to switch to an iterative solution method for linear systems, you can do so in one place in b), and all applications can take advantage of the extension.

1.3 Useful extensions

hpl 3: Need a little intro.

1.3.1 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 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. Consequently, we must tell you already now how you can take advantage of state-of-the-art iterative solution methods in FEniCS.

Setting linear solver parameters. Preconditioned Krylov solvers is a type of popular iterative methods that are easily accessible in FEniCS programs. The Poisson equation results in a symmetric, positive definite coefficient matrix, for which the optimal Krylov solver is the Conjugate Gradient (CG) method. However, the CG method requires boundary conditions to be implemented in a symmetric way. This is not the case by default, so then a Krylov solver for non-symmetric system, such as GMRES, is a better choice. Incomplete LU factorization (ILU) is a popular and robust all-round preconditioner, so let us try the GMRES-ILU pair:

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

Section 2.6.3 lists the most popular choices of Krylov solvers and preconditioners available in FEniCS.

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

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

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

A common platform for FEniCS users is Ubuntu Linux. The FEniCS distribution for Ubuntu contains PETSc, making this package the default linear algebra backend. The default solver is sparse LU decomposition (`'lu'`), and

the actual software that is called is then the sparse LU solver from UMFPACK (which PETSc has an interface to). The available linear algebra backends in a FEniCS installation is listed by

```
list_linear_algebra_backends()
```

The `parameters` database. We will normally like to control the tolerance in the stopping criterion and the maximum number of iterations when running an iterative method. Such parameters can be set by accessing the *global parameter database*, which is called `parameters` and which behaves as a nested dictionary. Write

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

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

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

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

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

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

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

We remark that default values for the global parameter database can be defined in an XML file. To generate such a file from the current set of parameters in a program, run

```
File('dolfin_parameters.xml') << parameters
```

If a `dolfin_parameters.xml` file is found in the directory where a FEniCS program is run, this file is read and used to initialize the `parameters` object. Otherwise, the file `.config/fenics/dolfin_parameters.xml` in the user's home

directory is read, if it exists. Another alternative is to load the XML (with any name) manually in the program:

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

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

An extended solver function. Let us extend the previous solver function from `p2D_func.py` such that it also offers the GMRES+ILU preconditioned Krylov solver.

```
from dolfin import *

def solver(
    f, u0, Nx, Ny, degree=1,
    linear_solver='Krylov', # Alt: 'direct'
    abs_tol=1E-5,           # Absolute tolerance in Krylov solver
    rel_tol=1E-3,           # Relative tolerance in Krylov solver
    max_iter=1000,          # Max no of iterations in Krylov solver
    log_level=PROGRESS,     # Amount of solver output
    dump_parameters=False,  # Write out parameter database?
    ):
    """
    Solve -Laplace(u)=f on [0,1]x[0,1] with 2*Nx*Ny Lagrange
    elements of specified degree and u=u0 (Expresssion) on
    the boundary.
    """
    # Create mesh and define function space
    mesh = UnitSquareMesh(Nx, Ny)
    V = FunctionSpace(mesh, 'Lagrange', degree)

    def u0_boundary(x, on_boundary):
        return on_boundary

    bc = DirichletBC(V, u0, u0_boundary)

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

    # Compute solution
    u = Function(V)

    if linear_solver == 'Krylov':
        prm = parameters['krylov_solver'] # short form
        prm['absolute_tolerance'] = abs_tol
        prm['relative_tolerance'] = rel_tol
        prm['maximum_iterations'] = max_iter
```

```
        print(parameters['linear_algebra_backend'])
        set_log_level(log_level)
        if dump_parameters:
            info(parameters, True)
        solver_parameters = {'linear_solver': 'gmres',
                             'preconditioner': 'ilu'}
    else:
        solver_parameters = {'linear_solver': 'lu'}

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

def solver_objects(
    f, u0, Nx, Ny, degree=1,
    linear_solver='Krylov', # Alt: 'direct'
    abs_tol=1E-5,           # Absolute tolerance in Krylov solver
    rel_tol=1E-3,           # Relative tolerance in Krylov solver
    max_iter=1000,          # Max no of iterations in Krylov solver
    log_level=PROGRESS,     # Amount of solver output
    dump_parameters=False,  # Write out parameter database?
    ):
    """As solver, but use objects for linear variational problem
    and solver."""
    # Create mesh and define function space
    mesh = UnitSquareMesh(Nx, Ny)
    V = FunctionSpace(mesh, 'Lagrange', degree)

    def u0_boundary(x, on_boundary):
        return on_boundary

    bc = DirichletBC(V, u0, u0_boundary)

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

    # Compute solution
    u = Function(V)
    problem = LinearVariationalProblem(a, L, u, bc)
    solver  = LinearVariationalSolver(problem)

    if linear_solver == 'Krylov':
        solver.parameters['linear_solver'] = 'gmres'
        solver.parameters['preconditioner'] = 'ilu'
        prm = solver.parameters['krylov_solver'] # short form
        prm['absolute_tolerance'] = abs_tol
        prm['relative_tolerance'] = rel_tol
        prm['maximum_iterations'] = max_iter
```

```
        print(parameters['linear_algebra_backend'])
        set_log_level(log_level)
        if dump_parameters:
            info(parameters, True)
        solver_parameters = {'linear_solver': 'gmres',
                             'preconditioner': 'ilu'}
    else:
        solver_parameters = {'linear_solver': 'lu'}

    solver.solve()
    return u
```

This new `solver` function, found in the file `p2D_iter.py`, replaces the one in `p2D_func.py`: it has all the functionality of the previous `solver` function, but can also solve the linear system with iterative methods and report the progress of such solvers.

Remark regarding unit tests. Regarding verification of the new `solver` function in terms of unit tests, it turns out that unit testing in a problem where the approximation error vanishes is gets more complicated when we use iterative methods. The problem is to keep the error due to iterative solution smaller than the tolerance used in the verification tests. First of all this means that the tolerances used in the Krylov solvers must be smaller than the tolerance used in the `assert` test, but this is no guarantee to keep the linear solver error this small. For linear elements and small meshes, a tolerance of 10^{-11} works well in the case of Krylov solvers too (using a tolerance 10^{-12} in those solvers. However, as soon as we switch to P2 elements, it is hard to force the linear solver error below 10^{-6}. Consequently, tolerances in tests depend on the numerical methods. The interested reader is referred to the `test_solver` function in `p2D_iter.py` for details: this test function tests the numerical solution for direct and iterative linear solvers, for different meshes, and different degrees of the polynomials in the finite element basis functions.

1.3.2 Linear variational problem and solver objects

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

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

Many objects have an attribute `parameters` corresponding to a parameter set in the global `parameters` database, but local to the object. Here,

`solver.parameters` play that role. Setting the CG method with ILU preconditioning as solution method and specifying solver-specific parameters can be done like this:

```
solver.parameters['linear_solver'] = 'gmres'
solver.parameters['preconditioner'] = 'ilu'
prm = solver.parameters['krylov_solver'] # short form
prm['absolute_tolerance'] = 1E-7
prm['relative_tolerance'] = 1E-4
prm['maximum_iterations'] = 1000
```

Settings in the global `parameters` database are propagated to parameter sets in individual objects, with the possibility of being overwritten as done above.

The linear variational problem and solver objects as outlined above are incorporated in an alternative solver function, named `solver_objects`, in `p2D_iter.py`. Otherwise, this function is parallel to the previously shown `solver` function.

1.3.3 Writing out the discrete solution

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

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

The elements in `u_array` correspond to function values of `u` at nodes in the mesh. Now, a fundamental question is: What are the coordinates of node `i` whose value is `u_array[i]`? To answer this question, we need to understand how to get our hands on the coordinates, and in particular, the numbering of degrees of freedom and the numbering of vertices in the mesh. We start with P1 (1st order Lagrange) elements where all the nodes are vertices in the mesh.

The function `mesh.coordinates()` returns the coordinates of the vertices as a `numpy` array with shape (M, d), M being the number of vertices in the mesh and d being the number of space dimensions:

```
>>> from dolfin import *
>>>
>>> mesh = UnitSquareMesh(2, 2)
>>> coor = mesh.coordinates()
>>> coor
array([[ 0. ,  0. ],
       [ 0.5,  0. ],
       [ 1. ,  0. ],
       [ 0. ,  0.5],
       [ 0.5,  0.5],
       [ 1. ,  0.5],
       [ 0. ,  1. ],
       [ 0.5,  1. ],
       [ 1. ,  1. ]])
```

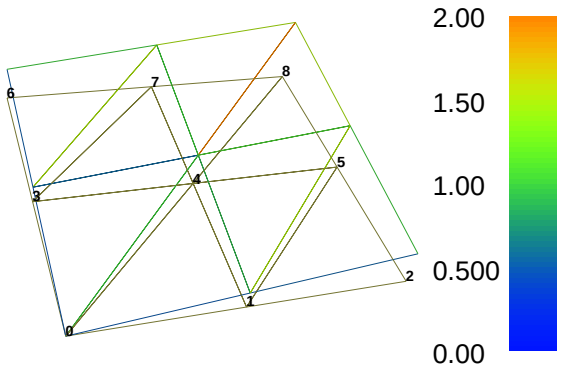
We see from this output that vertices are first numbered along $y = 0$ with increasing x coordinate, then along $y = 0.5$, and so on.

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

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

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

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



The vertex values of a `Function` object can be extracted by `u.compute_vertex_values()`, which returns an array where element `i` is the value of `u` at vertex `i`:

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

```
vertex 5: u_at_vertices[5]=1.5  u([ 1.   0.5])=1.5
vertex 6: u_at_vertices[6]=1    u([ 0.  1.])=1
vertex 7: u_at_vertices[7]=1.5  u([ 0.5  1. ])=1.5
vertex 8: u_at_vertices[8]=2    u([ 1.  1.])=2
```

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

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

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

For Lagrange elements of degree larger than 1, there are degrees of freedom (nodes) that do not correspond to vertices. **hpl 6**: Anders, is the following true? There is no simple way of getting the coordinates associated with the non-vertex degrees of freedom, so if we want to write out the values of a finite element solution, the following code snippet does the task at the vertices, and this will work for all kinds of Lagrange elements.

```
def compare_exact_and_numerical_solution(Nx, Ny, degree=1):
    u0 = Expression('1 + x[0]*x[0] + 2*x[1]*x[1]')
    f = Constant(-6.0)
    u = solver(f, u0, Nx, Ny, degree, linear_solver='direct')
    # Grab exact and numerical solution at the vertices and compare
    V = u.function_space()
    u0_Function = interpolate(u0, V)
    u0_at_vertices = u0_Function.compute_vertex_values()
    u_at_vertices = u.compute_vertex_values()
    coor = V.mesh().coordinates()
    for i, x in enumerate(coor):
        print('vertex %2d (%9g,%9g): error=%g'
              % (i, x[0], x[1],
                 u0_at_vertices[i] - u_at_vertices[i]))
        # Could compute u0(x) - u_at_vertices[i] but this
        # is much more expensive and gives more rounding errors
    center = (0.5, 0.5)
    error = u0(center) - u(center)
    print('numerical error at %s: %g' % (center, error))
```

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

Cheap vs expensive function evaluation.

Given a `Function` object `u`, we can evaluate its values in various ways:

1. `u(x)` for an arbitrary point `x`
2. `u.vector().array()[i]` for degree of freedom number `i`
3. `u.compute_vertex_values()[i]` at vertex number `i`

The first method, though very flexible, is in general very expensive while the other two are very efficient (but limited to certain points).

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

```
center = (0.5, 0.5)
error = u0(center) - u(center)
print('numerical error at %s: %g' % (center, error)
```

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

```
numerical error at (0.5, 0.5): -0.0833333
```

The discrepancy is due to the fact that the center point is not a node in this particular mesh, but a point in the interior of a cell, and `u` varies linearly over the cell while `u0` is a quadratic function. When the center point is a node, as in a $2(t \times 2)$ or $2(4 \times 4)$ mesh, the error is of the order 10^{-15}.

We have seen how to extract the nodal values in a `numpy` array. If desired, we can adjust the nodal values too. Say we want to normalize the solution such that $\max_j U_j = 1$. Then we must divide all U_j values by $\max_j U_j$. The following function performs the task:

```
def normalize_solution(u):
    """Normalize u: return u divided by max(u)."""
    u_array = u.vector().array()
    u_max = u_array.max()
    u_array /= u_max
    u.vector()[:] = u_array
    u.vector().set_local(u_array)  # alternative
    return u
```

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

Be careful when manipulating degrees of freedom.

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

All the code in this subsection can be found in the file `p2D_iter.py` in the `poisson` directory.

1.3.4 Parameterizing the number of space dimensions

FEniCS makes it is easy to write a unified simulation code that can operate in 1D, 2D, and 3D. We will conveniently make use of this feature in forthcoming examples. As an appetizer, go back to the introductory programs `p2D_plain.py` or `p2D_func.py` in the `poisson` directory and change the mesh construction from `UnitSquareMesh(6, 4)` to `UnitCubeMesh(6, 4, 5)`. Now the domain is the unit cube partitioned into $6 \times 4 \times 5$ boxes, and each box is divided into six tetrahedra-shaped finite elements for computations. Run the program and observe that we can solve a 3D problem without any other modifications (!). The visualization allows you to rotate the cube and observe the function values as colors on the boundary.

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

```
def unit_hypercube(divisions, degree):
    mesh_classes = [UnitIntervalMesh, UnitSquareMesh, UnitCubeMesh]
    d = len(divisions)
    mesh = mesh_classes[d-1](*divisions)
    V = FunctionSpace(mesh, 'Lagrange', degree)
    return V, mesh
```

The construction `mesh_class[d-1]` will pick the right name of the object used to define the domain and generate the mesh. Moreover, the argument `*divisions` sends all the component of the list `divisions` as separate arguments. For example, in a 2D problem where `divisions` has two elements, the statement

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

is equivalent to

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

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

1.3.5 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

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

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, using the same elements for the components of w as we used for u. This process is known as *projection*.

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

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

grad_u = project(grad(u), W)
```

Figure 1.2 shows example of how such a smoothed `gradu(u)` vector field is visualized.

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.

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

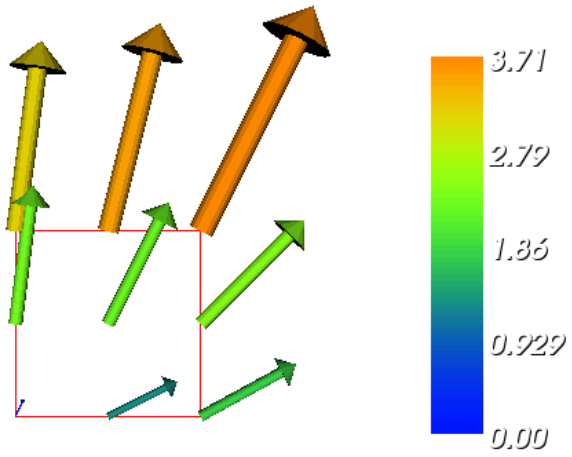


Figure 1.2: Example of visualizing the vector field ∇u by arrows at the nodes.

```
grad_u_x, grad_u_y = grad_u.split(deepcopy=True)
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

```
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

```
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. This is less convenient to work with.

The function `gradient(u)` in `p2D_iter.py` returns a projected (smoothed) ∇u vector field, given some finite element function `u`:

```
def gradient(u):
    """Return grad(u) projected onto same space as u."""
```

```
    V = u.function_space()
    mesh = V.mesh()
    V_g = VectorFunctionSpace(mesh, 'Lagrange', 1)
    grad_u = project(grad(u), V_g)
    grad_u.rename('grad(u)', 'continuous gradient field')
    return grad_u
```

Examining the arrays with vertex values 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 (i.e., as soon as we are one element away from the boundary). The `application_test_gradient` function in `p2D_iter.py` performs some experiments.

Detour: Manual projection.

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

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

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

$$a(w, v) = L(v) \quad \forall v \in \hat{V^{(\mathrm{g})}}, \tag{1.16}$$

where

$$a(w, v) = \int_\Omega w \cdot v \,\mathrm{d}x, \tag{1.17}$$

$$L(v) = \int_\Omega \nabla u \cdot v \,\mathrm{d}x\,. \tag{1.18}$$

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

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

```
V_g = VectorFunctionSpace(mesh, '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.

1.3.6 A variable-coefficient Poisson problem

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

$$\begin{aligned} -\nabla \cdot [p(x,y)\nabla u(x,y)] &= f(x,y) \quad \text{in } \Omega, \\ u(x,y) &= u_0(x,y) \quad \text{on } \partial\Omega\,. \end{aligned} \tag{1.19}$$

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

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

Modifications of the PDE solver. What are the modifications we need to do in the previously shown codes to incorporate the variable coefficient p? from Section 1.3.3?

- `solver` must take `p` as argument,
- `f` in our test problem must be an `Expression` since it is no longer a constant,
- a new `Expression p` must be defined for the variable coefficient,
- the formula for $a(u, v)$ in the variational problem is slightly changed.

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

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

The function spaces for u and v are the same as in 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

$$a(u,v) = \int_\Omega p\nabla u\cdot\nabla v\,\mathrm{d}x, \tag{1.20}$$

$$L(v) = \int_\Omega fv\,\mathrm{d}x\,. \tag{1.21}$$

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

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

by

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

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

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

No additional modifications are necessary. The file `p2D_vc.py` (variable-coefficient Poisson problem in 2D) is a copy of `p2D_iter.py` with the mentioned changes incorporated. Observe that $p=1$ recovers the original problem in `p2D_iter.py`.

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

Modifications of the flux computations. 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 1.3.5, 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 $w\in V^{(\mathrm{g})}$ such that

$$a(w,v) = L(v) \quad \forall v\in \hat{V^{(\mathrm{g})}}, \tag{1.22}$$

where

$$a(w, v) = \int_\Omega w \cdot v \, \mathrm{d}x, \tag{1.23}$$

$$L(v) = \int_\Omega (-p\nabla u) \cdot v \, \mathrm{d}x \, . \tag{1.24}$$

This problem is identical to the one in Section 1.3.5, except that p enters the integral in L.

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

```
flux = project(-p*grad(u),
               VectorFunctionSpace(mesh, 'Lagrange', degreee))
```

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

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

Plotting the flux vector field is naturally as easy as plotting the gradient (see Section 1.3.5):

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

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

hpl 7: The following is not done properly in the revised version. The function `application_test_gradient` in the program `p2D_vc.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 1.4.4.

1.3.7 Creating the linear system explicitly

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

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

which is nothing but a linear system,

$$AU = b,$$

where the entries in A and b are given by

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

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

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

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

The matrix A and vector b are first assembled without incorporating essential (Dirichlet) boundary conditions. Thereafter, the call `bc.apply(A, b)` performs the necessary modifications of the linear system such that `u` is guaranteed to equal the prescribed boundary values. When we have multiple Dirichlet conditions stored in a list `bcs`, as explained in Section 1.6.2, we must apply each condition in `bcs` to the system:

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

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

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

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

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

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

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

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

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

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

Matrix processing in Python or MATLAB/Octave is only feasible for small PDE problems since the `numpy` arrays or matrices in MATLAB file format are dense matrices. 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).

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

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

Appropriate names of solvers and preconditioners are found in Section 2.6.3.

To control tolerances in the stopping criterion and the maximum number of iterations, one can explicitly form a `KrylovSolver` object and set items in its `parameters` attribute (see also Section 1.3.2):

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

The function `solver_linalg` in the program file `p2D_vc.py` implements a solver function where the user can choose between different types of assembly: the variational (`solve(a == L, u, bc)`), assembling the matrix and right-hand side separately, and assembling the system such that the coefficient matrix preserves symmetry. The function `application_linalg` runs a test problem on sequence of meshes and solves the problem with symmetric and non-symmetric modification of the coefficient matrix. One can monitor the number of Krylov method iteration and realize that with a symmetric coefficient matrix, the Conjugate Gradient method requires slightly fewer iterations than GMRES in the non-symmetric case. Taking into account that the Conjugate Gradient method has less work per iteration, there is some efficiency to be gained by using `assemble_system`.

hpl 8: Running `application_linalg`, the results are strange: Why does the `solve(a==L,...)` method need many more iterations than `solve(A, U, b, ...)` when we use the same Krylov parameter settings? Something wrong with the settings?

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

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

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

Creating the linear system explicitly in a program can have some advantages in more advanced problem settings. For example, A may be constant throughout a time-dependent simulation, so we can avoid recalculating A at every time level and save a significant amount of simulation time. Sections 2.1.2 and 2.1.3 deal with this topic in detail.

1.4 Visualization

Perhaps you are not particularly amazed by viewing the simple surface of u in the test problem used in the previous sections. 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. The present chapter starts with the solver for a membrane deflection, where the geometry is a circle instead of the unit square, and where the involved functions have more exciting shapes. We then go on with explaining how the membrane deflection and the pressure load can be visualized, using both the built-in FEniCS visualization tool and the powerful application ParaView. Finally, we return to box-shaped domains with uniform partition and show how the unstructured data in FEniCS finite element solvers can be transformed to structured mesh representations like those commonly used for finite difference methods. The structured mesh representation, whenever possible, gives greater flexibility with respect to visualization and data analysis.

1.4.1 Deflection of a circular membrane

The problem. 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

$$-T\nabla^2 D = p(x, y) \quad \text{in } \Omega = \{(x, y) \,|\, x^2 + y^2 \leq R\}, \tag{1.25}$$

with

$$p(x, y) = \frac{A}{2\pi\sigma} \exp\left(-\frac{1}{2}\left(\frac{x - x_0}{\sigma}\right)^2 - \frac{1}{2}\left(\frac{y - y_0}{\sigma}\right)^2\right). \tag{1.26}$$

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.

Scaling. The localization of the pressure, (x_0, y_0), is for simplicity set to $(0, R_0)$. There are many physical parameters in this problem, and we can benefit from grouping them by means of scaling. Let us introduce dimensionless

coordinates $\bar{x} = x/R$, $\bar{y} = y/R$, and a dimensionless deflection $w = D/D_c$, where D_c is a characteristic size of the deflection. Introducing $\bar{R}_0 = R_0/R$, we get

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

where

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

With an appropriate scaling, $\bar{w}$ and its derivatives are of size unity, so the left-hand side of the scaled PDE is about unity in size, while the right-hand side has α as its characteristic size. This suggest choosing α to be unity, or around unit. We shall in particular choose $\alpha = 4$. With this value, the solution is $w(\bar{x}, \bar{y}) = 1 - \bar{x}^2 - \bar{y}^2$. (One can also find the analytical solution in scaled coordinates and show that the maximum deflection $D(0, 0)$ is D_c if we choose $\alpha = 4$ to determine D_c.) With $D_c = AR^2/(8\pi\sigma T)$ and dropping the bars we get the scaled problem

$$\nabla^2 w = 4 \exp\left(-\beta^2(x^2 + (y - R_0)^2)\right), \tag{1.27}$$

to be solved over the unit circle with $w = 0$ on the boundary. Now there are only two parameters to vary: the dimensionless extent of the pressure, β, and the localization of the pressure peak, $R_0 \in [0, 1]$. As $\beta \to 0$, we have a special case with solution $w = 1 - x^2 - y^2$.

Given a computed w, the physical deflection is given by

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

Implementation. Very few modifications of the software in `p2D_iter.py` are required. Actually, the `solver` function can be reused, except that the domain is now a circle and not a square. We change the `solver` function by letting the mesh be an argument `mesh` (instead of `Nx` and `Ny`):

```
def solver(
    f, u0, mesh, degree=1,
    linear_solver='Krylov', # Alt: 'direct'
    ...):
    V = FunctionSpace(mesh, 'Lagrange', degree)
    ...
```

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

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

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

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

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

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

```
f.beta = 12
f.R0 = 0.3
```

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

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

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

We need some evidence that the program works, and to this end we may use the analytical solution listed above for the case $\beta = 0$.

The final program is found in the file `membrane.py`, located in the `poisson` directory. The key function to simulate membrane deflection is named `application`.

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

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

    w = solver(p, Constant(0), mesh, degree=1,
               linear_solver='direct')
```

```
    w.rename('w', 'deflection')  # set name and label (description)

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

    # Dump p and w to file in VTK format
    vtkfile = File('membrane.pvd')
    vtkfile << w
    vtkfile << p
```

: Tip: Clean up compilation files.

Running FEniCS programs usually implies compilation of some generated C++ code. If you run into compilation errors, the first action to perform is to clean up compilation files by the command

Terminal

```
Terminal> instant-clean
```

Then recompile, and if the error persist, open the `.log` file referred to in the error message, search for the word *error*, and see if the error message from the C++ compiler makes sense. As a user, your compilation errors will in most cases arise from wrong (C++) syntax in `Expression` objects.

Choosing a very peak-formed pressure with large β (e.g., $\beta \geq 20$) and a location R_0 toward the circular boundary (e.g., $R_0 = 0.5$), may produce an exciting visual demonstrations of 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 p. You are strongly encouraged to play around with the plots and different mesh resolutions:

Terminal

```
Terminal> python -c 'import membrane as m; m.application()' \
          membrane.py
```

1.4.2 Quick built-in visualization

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

We have made a program `membrane.py` for the membrane deflection problem in Section 1.4.1 and added various demonstrations of plotting capabilities. You are encouraged to play around with `membrane.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:

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

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

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

- pressing `m` turns visualization of the mesh on and off,
- pressing `b` turns on and off a bounding box,
- pressing `p` dumps the plot to a PNG file,
- pressing `P` dumps the plot to a PDF file,
- pressing 'Ctrl +' stretches the surface in the z direction,
- pressing 'Ctrl -' shrinks++ the surface in the z direction,
- pressing 'Ctrl w' closes the plot window,
- pressing 'Ctrl q' closes all plot windows.

The plots created by pressing `p` or `P` are stored in files with names `dolfin_plot_X.png` or `dolfin_plot_X.pdf`, where `X` is an integer that is increased by one from the last plot that was made. The file stem `dolfin_plot_` can be set to something

more suitable through the `hardcopy_prefix` keyword argument to the `plot` function, for instance, `plot(f, hardcopy_prefix='pressure')`.

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

Terminal

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

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

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

The commands above appear in the `application2` function in the `membrane.py` file.

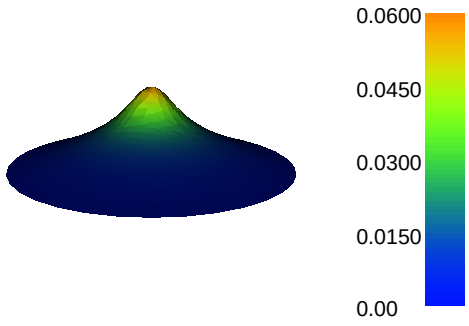


Figure 1.3: Plot of the deflection of a membrane.

1.4.3 ParaView

We strongly recommend FEniCS users to visualize multi-dimensional scalar and vector fields with ParaView[5].

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

The `application` function in the `membrane.py` file writes w and p, to as finite element functions, to file. The default filenames are `membrane000000.vtu` for the first field, w, and `membrane000001.vtu` for the second field, p. These files are in VTK format and their data can be visualized in ParaView.

1. Start the ParaView application.
2. Open a file with **File - Open...**. You will see a list of `.vtu` files, more specifically you see `mambrane..vtu`. Click to the left of that name to expand the collection of `membrane*.vtu` files. Choose the first one, `membrane000000.vtu`
3. Click on **Apply** to the left (*Properties* pane) in the GUI, and ParaView will visualize the contents of the file, here as a color image.
4. To get rid of the axis in the lower left corner of the plot area and axis cross in the middle of the circle, find the *Show Orientation Axis* and *Show Center* buttons to the right in the second row of buttons at the top of the GUI. Click on these buttons to toggle axis information on/off.
5. If you want a color bar to explain the mapping between w values and colors, go to the *Color Map Editor* in the right of the GUI and use the *Show/hide color legend* button. Alternatively, find *Coloring* in the lower left part of the GUI, and toggle the *Show* button.
6. The color map, by default going from blue (low values) to red (high values), can easily be changed. Find the *Coloring* menu in the left part of the GUI, click *Edit*, then in the *Color Map Editor* double click at the left end of the color spectrum and choose another color, say yellow, then double click at the right and of the spectrum and choose pink, scroll down to the bottom of the dialog and click *Update*. The color map now goes from yellow to pink.
7. To save the plot to file, click on **File - Export Scene...**, choose a file type, fill in a filename, and save. See Figure 1.4 (middle).
8. To change the background color of plots, choose **Edit - Settings...**, **Color** tab, click on **Background Color**, and choose it to be, e.g., white. Then choose **Foreground Color** to be something different.
9. To plot the mesh with colors reflecting the size of w, find the *Representation* drop down menu in the left part of the GUI, and replace *Surface* by *Wireframe*.
10. To overlay a surface plot with a wireframe plot, load w and plot as surface, then load w again and plot as wireframe. Make sure both icons in the *Pipeline Browser* in the left part of the GUI are *on* for the two `membrane000000.vtu` files. See Figure 1.4 (left).

11. Redo the surface plot. Then we can add some contour lines. Press the semi-sphere icon in the third row of buttons at the top of the GUI (the so-called *filters*). A set of contour values can now be specified at in a dialog box in the left part of the GUI. Remove the default contour (0.578808) and add 0.01, 0.02, 0.03, 0.04, 0.05. Click *Apply* and see an overlay of white contour lines. In the *Pipeline Browser* you can click on the icons to turn a filter on or off.

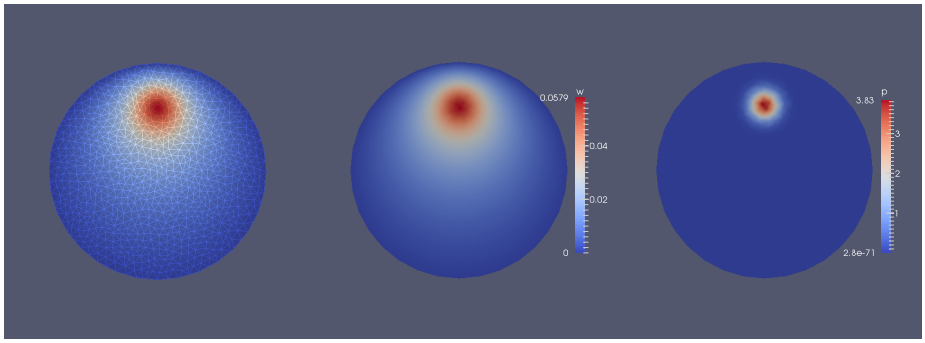


Figure 1.4: Default visualizations in ParaView: deflection (left, middle) and pressure load (right).

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

For more information, we refer to The ParaView Guide [23] (free PDF available) and to the ParaView tutorial[6].

1.4.4 Taking advantage of structured mesh data

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

A necessary first step is to transform our `mesh` object to an object representing a rectangle with equally-shaped *rectangular* cells. The second step is to transform

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

the one-dimensional array of nodal values to a two-dimensional array holding the values at the corners of the cells in the structured mesh. We want to access a value by its i and j indices, i counting cells in the x direction, and j counting cells in the y direction. This transformation is in principle straightforward, yet it frequently leads to obscure indexing errors, so using software tools to ease the work is advantageous.

In the directory `src/modules`, associated with this booklet, we have included a Python module `BoxField` that can take a finite element function `u` computed by a FEniCS software and represent it on a structured box-shaped mesh and assign or extract values by multi-dimensional indexing: `[i]` in 1D, `[i,j]` in 2D, and `[i,j,k]` in 3D. Given a finite element function `u`, the following function returns a `BoxField` object that represents `u` on a structured mesh:

```
def structured_mesh(u, divisions):
    """Represent u on a structured mesh."""
    # u must have P1 elements, otherwise interpolate to P1 elements
    u2 = u if u.ufl_element().degree() == 1 else \
         interpolate(u, FunctionSpace(mesh, 'Lagrange', 1))
    mesh = u.function_space().mesh()
    from BoxField import dolfin_function2BoxField
    u_box = dolfin_function2BoxField(
        u2, mesh, divisions, uniform_mesh=True)
    return u_box
```

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

The `u_box` object contains several useful data structures:

- `u_box.grid`: object for the structured mesh
- `u_box.grid.coor[X]`: grid coordinates in `X=0` direction
- `u_box.grid.coor[Y]`: grid coordinates in `Y=1` direction
- `u_box.grid.coor[Z]`: grid coordinates in `Z=2` direction
- `u_box.grid.coorv[X]`: vectorized version of `u_box.grid.coor[X]` (for vectorized computations or surface plotting)
- `u_box.grid.coorv[Y]`: vectorized version of `u_box.grid.coor[Y]`
- `u_box.grid.coorv[Z]`: vectorized version of `u_box.grid.coor[Z]`
- `u_box.values`: `numpy` array holding the `u` values; `u_box.values[i,j]` holds `u` at the mesh point with coordinates `(u_box.grid.coor[X], u_box.grid.coor[Y])`

Iterating over points and values. Let us go back to the `solver` function in the `p2D_vc.py` code from Section 1.3.6, compute `u`, map it onto a `BoxField` object for a structured mesh representation, and write out the coordinates and function values at all mesh points:

```
u = solver(p, f, u0, nx, ny, 1, linear_solver='direct')
u_box = structured_mesh(u, (nx, ny))
u_ = u_box.values       # numpy array
X = 0;  Y = 1           # for indexing in x and y direction

# Iterate over 2D mesh points (i,j)
print('u_ is defined on a structured mesh with %s points' %
      str(u_.shape))
for j in range(u_.shape[1]):
    for i in range(u_.shape[0]):
        print('u[%d,%d]=u(%g,%g)=%g' %
              (i, j,
               u_box.grid.coor[X][i], u_box.grid.coor[X][j],
               u_[i,j]))
```

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

```
x = u_box.grid.coor[X]
dx = x[1] - x[0]
u_xx = (u_[i-1,j] - 2*u_[i,j] + u_[i+1,j])/dx**2
```

Surface plot. The ability to access a finite element field in the way one can access a finite difference-type of field is handy in many occasions, including visualization and data analysis. With Matplotlib we can create a surface plot, see Figure 1.5 (upper left):

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
fig = plt.figure()
ax = fig.gca(projection='3d')
cv = u_box.grid.coorv  # vectorized mesh coordinates
ax.plot_surface(cv[X], cv[Y], u_, cmap=cm.coolwarm,
                rstride=1, cstride=1)
plt.title('Surface plot of solution')
```

The key issue is to know that the coordinates needed for the surface plot is in `u_box.grid.coorv` and that the values are in `u_`.

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

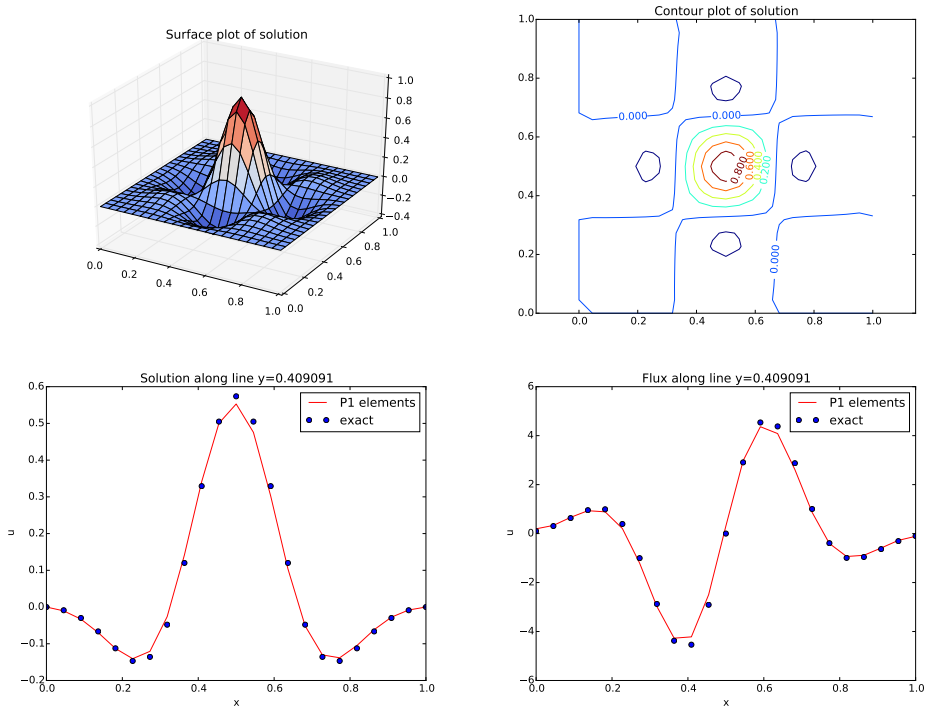


Figure 1.5: Various plots of the solution on a structured mesh.

```
fig = plt.figure()
ax = fig.gca()
levels = [1.5, 2.0, 2.5, 3.5]
cs = ax.contour(cv[X], cv[Y], u_, levels=levels)
plt.clabel(cs)  # add labels to contour lines
plt.axis('equal')
plt.title('Contour plot of solution')
```

The result appears in Figure 1.5 (upper right).

Curve plot through the mesh. A handy feature of `BoxField` objects is the ability to give a start point in the grid and a direction, and then extract the field and corresponding coordinates along the nearest line of mesh points. In 3D fields one can also extract data in a plane. Say we want to plot u along the line $y = 0.4$. The mesh points, `x`, and the u values along this line, `u_val`, are extracted by

```
start = (0, 0.4)
X = 0
x, u_val, y_fixed, snapped = u_box.gridline(start, direction=X)
```

The variable `snapped` is true if the line had to be snapped onto a gridline and in that case `y_fixed` holds the snapped (altered) y value. To avoid interpolation in the structured mesh, `snapped` is in fact *always* true.

A comparison of the numerical and exact solution along the line $y = 0.5$ (snapped from $y = 0.4$) is made by the following code:

```
start = (0, 0.4)
x, u_val, y_fixed, snapped = u_box.gridline(start, direction=X)
u_e_val = [u0((x_, y_fixed)) for x_ in x]

plt.figure()
plt.plot(x, u_val, 'r-')
plt.plot(x, u_e_val, 'bo')
plt.legend(['P1 elements', 'exact'], loc='upper left')
plt.title('Solution along line y=%g' % y_fixed)
plt.xlabel('x');  plt.ylabel('u')
```

See Figure 1.5 (lower left) for the resulting curve plot.

Curve plot of the flux. Let us also compare the numerical and exact flux $-p\partial u/\partial x$ along the same line as above:

```
flux_u = flux(u, p)
flux_u_x, flux_u_y = flux_u.split(deepcopy=True)

# Plot the numerical and exact flux along the same line
flux2_x = flux_u_x if flux_u_x.ufl_element().degree() == 1 \
          else interpolate(flux_x,
               FunctionSpace(u.function_space().mesh(),
                             'Lagrange', 1))
flux_u_x_box = structured_mesh(flux_u_x, (nx,ny))
x, flux_u_val, y_fixed, snapped = \
   flux_u_x_box.gridline(start, direction=X)
y = y_fixed

plt.figure()
plt.plot(x, flux_u_val, 'r-')
plt.plot(x, flux_u_x_exact(x, y_fixed), 'bo')
plt.legend(['P1 elements', 'exact'], loc='upper right')
plt.title('Flux along line y=%g' % y_fixed)
plt.xlabel('x');  plt.ylabel('u')
```

The second `plt.plot` command requires a Python function `flux_u_x_exact(x,y)` to be available for the exact flux expression.

Note that Matplotlib is one choice of plotting package. With the unified interface in the SciTools package[7] one can access Matplotlib, Gnuplot, MATLAB, OpenDX, VisIt, and other plotting engines through the same API.

[7] `https://github.com/hplgit/scitools`

Test problem. The graphics referred to in Figure 1.5 correspond to a test problem with prescribed solution $u_e = H(x)H(y)$, where

$$H(x) = e^{-16(x-\frac{1}{2})^2} \sin(3\pi x)\,.$$

We just fit a function $f(x, y)$ in the PDE (can choose $p = 1$), and notice that $u = 0$ along the boundary of the unit square. Although it is easy to carry out the differentiation of f by hand and hardcode the resulting expressions in an `Expression` object, a more reliable habit is to use Python's symbolic computing engine, `sympy`, to perform mathematics and automatically turn formulas into C++ syntax for `Expression` objects. The following text assumes some familiarity with `sympy` and illustrates how FEniCS programmers may take advantage of symbolic computing.

We start out with defining the exact solution in `sympy`:

```
from sympy import exp, sin, pi  # for use in math formulas
import sympy as sym
H = lambda x: exp(-16*(x-0.5)**2)*sin(3*pi*x)
x, y = sym.symbols('x[0], x[1]')
u = H(x)*H(y)
```

Define symbolic coordinates as required in `Expression` objects.

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

Turning the expression for `u` into C or C++ syntax for `Expression` objects needs two steps. First we ask for the C code of the expression,

```
u_c = sym.printing.ccode(u)
```

Then we do some editing of `u_c` to match the required syntax of `Expression` objects. Printing `u_c` gives (here manually broken up as two lines)

```
-exp(-16*pow(x[0] - 0.5, 2) - 16*pow(x[1] - 0.5, 2))*
sin(3*M_PI*x[0])*sin(3*M_PI*x[1])
```

The necessary syntax adjustment is replacing the symbol `M_PI` for π in C/C++ by `DOLFIN_PI`:

```
u_c = u_c.replace('M_PI', 'DOLFIN_PI')
u0 = Expression(u_c)
```

Thereafter, we can progress with the computation of $f = -\nabla \cdot (p\nabla u)$:

```
p = 1
f = sym.diff(-p*sym.diff(u, x), x) + sym.diff(-p*sym.diff(u, y), y)
f = sym.simplify(f)
f_c = sym.printing.ccode(f)
f_c = f_c.replace('M_PI', 'DOLFIN_PI')
f = Expression(f_c)
```

We also need a Python function for the exact flux $-p\partial u/\partial x$:

```
flux_u_x_exact = sym.lambdify([x, y], -p*sym.diff(u, x),
                              modules='numpy')
```

It remains to define `p = Constant(1)` and set `nx` and `ny` before calling `solver` to compute the finite element solution of this problem.

1.5 Postprocessing computations

hpl 3: Need a little intro.

1.5.1 Computing functionals

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

$$\frac{1}{2}||\nabla u||^2 \equiv \frac{1}{2}\int_\Omega \nabla u\cdot\nabla u\,\mathrm{d}x, \tag{1.28}$$

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

$$||u_\mathrm{e} - u|| = \left(\int_\Omega (u_\mathrm{e}-u)^2\,\mathrm{d}x\right)^{1/2}, \tag{1.29}$$

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

$$F = -\int_\Gamma p\nabla u\cdot\boldsymbol{n}\,\mathrm{d}s, \tag{1.30}$$

where $\boldsymbol{n}$ is an outward unit normal at Γ and p is a coefficient (see the problem in Section 1.3.6 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 (1.28) is described in the UFL language in the same manner as we describe weak forms:

```
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 as explained in Section 1.6.4.

Error functional. Computation of (1.29) is typically done by

```
error = (u - u_exact)**2*dx
E = sqrt(abs(assemble(error)))
```

The exact solution u_{e} is here in a `Function` or `Expression` object `u_exact`, while `u` is the finite element approximation. (Sometimes, for very small error values, the result of `assemble(error)` can be a (very small) negative number, so we have used `abs` in the expression for `E` above to ensure a positive value for the `sqrt` function.)

As will be explained and demonstrate in Section 1.5.2, the integration of `(u - u_exact)**2*dx` can result in too optimistic convergence rates unless one is careful how `u_exact` is transferred onto a mesh. The general recommendation for reliable error computation is to use the `errornorm` function (see `pydoc dolfin.errornorm` and Section 1.5.2 for more information):

```
E = errornorm(u_exact, u)
```

Flux Functionals. To compute flux integrals like $F = -\int_\Gamma p\nabla u\cdot\boldsymbol{n}\,\mathrm{d}s$ we need to define the $\boldsymbol{n}$ vector, referred to as *facet normal* in FEniCS. If the surface domain Γ in the flux integral is the complete boundary we can perform the flux computation by

```
n = FacetNormal(mesh)
flux = -p*dot(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 1.6.4. Assuming that the part corresponds to subdomain number `i`, the relevant syntax for the variational formulation of the flux is `-p*inner(grad(u), n)*ds(i)`.

1.5.2 Computing convergence rates

To illustrate error computations and convergence of finite element solutions, we have included a function `convergence_rate` in the `p2D_vc.py` program. This is a tool that is very handy when verifying finite element codes and will therefore be explained in detail here.

The L^2 norm of the error in a finite element approximation u, u_{e} being the exact solution, is given by

Various ways of computing the error.

$$E = \left(\int_\Omega (u_e - u)^2 \, \mathrm{d}x \right)^{1/2},$$

and implemented in FEniCS by

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

Sometimes, for very small error values, the result of `assemble(error)` can be a (very small) negative number, so we have used `abs` in the expression for `E` above to ensure a positive value for the `sqrt` function.

We remark that `u_e` will, in the expression above, be interpolated onto the function space `V` before `assemble` can perform the integration over the domain. This implies that the exact solution used in the integral will vary linearly over the cells, and not as a sine function, if `V` corresponds to linear Lagrange elements. This situation may yield a smaller error `u - u_e` than what is actually true. More accurate representation of the exact solution is easily achieved by interpolating the formula onto a space defined by higher-order elements, say of third degree:

```
Ve = FunctionSpace(mesh, '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` to the same space:

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

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

```
E = errornorm(u_exact, u, normtype='L2', degree=3)
```

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

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

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

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

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

All the various types of error computations here are placed in a function `compute_errors` in `p2D_vc.py`: **hpl 10**: Necessary to repeat code? New info is essentiall the return dict. **hpl 11**: Anders, I (in 2010...) ran into problems with `dolfin.errornorm`, see comments in the code below, and made the version below. We should check out these problems again and adjust `dolfin.errornorm` if necessary.

```
def compute_errors(u, u_exact):
    """Compute various measures of the error u - u_exact, where
    u is a finite element Function and u_exact is an Expression."""

    # Compute error norm (for very small errors, the value can be
    # negative so we run abs(assemble(error)) to avoid failure in sqrt

    V = u.function_space()

    # Function - Expression
    error = (u - u_exact)**2*dx
    E1 = sqrt(abs(assemble(error)))

    # Explicit interpolation of u_e onto the same space as u:
    u_e = interpolate(u_exact, V)
    error = (u - u_e)**2*dx
    E2 = sqrt(abs(assemble(error)))

    # Explicit interpolation of u_exact to higher-order elements,
    # u will also be interpolated to the space Ve before integration
    Ve = FunctionSpace(V.mesh(), 'Lagrange', 5)  # mesh here: BUG, module mesh in dolfin...make
    u_e = interpolate(u_exact, Ve)
    error = (u - u_e)**2*dx
```

```
    E3 = sqrt(abs(assemble(error)))

    # dolfin.errornorm interpolates u and u_e to a space with
    # given degree, and creates the error field by subtracting
    # the degrees of freedom, then the error field is integrated
    # TEMPORARY BUG - doesn't accept Expression for u_e
    #E4 = errornorm(u_e, u, normtype='l2', degree=3)
    # Manual implementation errornorm to get around the bug:
    def errornorm(u_exact, u, Ve):
        u_Ve = interpolate(u, Ve)
        u_e_Ve = interpolate(u_exact, Ve)
        e_Ve = Function(Ve)
        # Subtract degrees of freedom for the error field
        e_Ve.vector()[:] = u_e_Ve.vector().array() - u_Ve.vector().array()
        # More efficient computation (avoids the rhs array result above)
        #e_Ve.assign(u_e_Ve)                      # e_Ve = u_e_Ve
        #e_Ve.vector().axpy(-1.0, u_Ve.vector())  # e_Ve += -1.0*u_Ve
        error = e_Ve**2*dx(Ve.mesh())
        return sqrt(abs(assemble(error))), e_Ve
    E4, e_Ve = errornorm(u_exact, u, Ve)

    # Infinity norm based on nodal values
    u_e = interpolate(u_exact, V)
    E5 = abs(u_e.vector().array() - u.vector().array()).max()

    # H1 seminorm
    error = inner(grad(e_Ve), grad(e_Ve))*dx
    E6 = sqrt(abs(assemble(error)))

    # Collect error measures in a dictionary with self-explanatory keys
    errors = {'u - u_exact': E1,
              'u - interpolate(u_exact,V)': E2,
              'interpolate(u,Ve) - interpolate(u_exact,Ve)': E3,
              'errornorm': E4,
              'infinity norm (of dofs)': E5,
              'grad(error) H1 seminorm': E6}

    return errors
```

Computing convergence rates empirically. Calling the `solver` function for finer and finer meshes enables us to study the convergence rate. Define the element size $h = 1/n$, where n is the number of cell divisions in x and y direction (`n=Nx=Ny` in the code). We perform experiments with $h_0 > h_1 > h_2 \cdots$ and compute the corresponding errors E_0, E_1, E_3 and so forth. Assuming $E_i = Ch_i^r$ for unknown constants C and r, we can compare two consecutive experiments, $E_i = Ch_i^r$ and $E_{i-1} = Ch_{i-1}^r$, and solve for r:

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

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

The procedure above can easily be turned into Python code. Here we run through a different types of elements (P1, P2, P3, and P4), perform experiments over a series of refined meshes, and for each experiment report the six error types as returned by `compute_errors`:

```
def convergence_rate(u_exact, f, u0, p, degrees,
                     n=[2**(k+3) for k in range(5)]):
    """
    Compute convergence rates for various error norms for a
    sequence of meshes with Nx=Ny=b and P1, P2, ...,
    Pdegrees elements. Return rates for two consecutive meshes:
    rates[degree][error_type] = r0, r1, r2, ...
    """

    h = {}  # Discretization parameter, h[degree][experiment]
    E = {}  # Error measure(s), E[degree][experiment][error_type]
    P_degrees = 1,2,3,4
    num_meshes = 5

    # Perform experiments with meshes and element types
    for degree in P_degrees:
        n = 4   # Coarsest mesh division
        h[degree] = []
        E[degree] = []
        for i in range(num_meshes):
            n *= 2
            h[degree].append(1.0/n)
            u = solver(p, f, u0, n, n, degree,
                       linear_solver='direct')
            errors = compute_errors(u, u_exact)
            E[degree].append(errors)
            print('2*(%dx%d) P%d mesh, %d unknowns, E1=%g' %
                  (n, n, degree, u.function_space().dim(),
                   errors['u - u_exact']))
    # Convergence rates
    from math import log as ln  # log is a dolfin name too
    error_types = list(E[1][0].keys())
    rates = {}
    for degree in P_degrees:
        rates[degree] = {}
        for error_type in sorted(error_types):
            rates[degree][error_type] = []
            for i in range(num_meshes):
                Ei   = E[degree][i][error_type]
```

```
                Eim1 = E[degree][i-1][error_type]
                r = ln(Ei/Eim1)/ln(h[degree][i]/h[degree][i-1])
                rates[degree][error_type].append(round(r,2))
    return rates

def convergence_rate_sin():
    """Compute convergence rates for u=sin(x)*sin(y) solution."""
    omega = 1.0
    u_exact = Expression('sin(omega*pi*x[0])*sin(omega*pi*x[1])',
                         omega=omega)
    f = 2*omega**2*pi**2*u_exact
    u0 = Constant(0)
    p = Constant(1)
    # Note: P4 for n>=128 seems to break down
    rates = convergence_rates(u_exact, f, u0, p, degrees=4,
                              n=[2**(k+3) for k in range(5)])
    # Print rates
    print('\n\n')
    for error_type in error_types:
        print(error_type)
        for degree in P_degrees:
            print('P%d: %s' %
                  (degree, str(rates[degree][error_type])[1:-1]))
```

Note how we make a complete general function `convergence_rate`, aimed at any 2D Poisson problem in the class we now can solve, and then call this general function in `convergence_rate_sin` for a special test case.

Test problem. Section 1.3.5 specifies a more complicated solution,

$$u(x,y) = \sin(\omega\pi x)\sin(\omega\pi y)$$

on the unit square. This choice implies $f(x,y) = 2\omega^2\pi^2 u(x,y)$. With ω restricted to an integer it follows that $u_0 = 0$.

We need to define the appropriate boundary conditions, the exact solution, and the f function in the code:

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

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

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

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

Experiments. Calling `convergence_rate_sin()` gives some interesting results. Using the error measure `E5` based on the infinity norm of the difference of the degrees of freedom, we have

element	$n = 8$	$n = 16$	$n = 32$	$n = 64$	$n = 128$
P1	1.99	1.97	1.99	2.0	2.0
P2	3.99	3.96	3.99	4.0	3.99
P3	3.96	3.89	3.96	3.99	4.0
P4	3.75	4.99	5.0	5.0	

The computations with P4 elements on a 128×128 with a direct solver (UMFPACK) on a small laptop broke down. Otherwise we achieve expected results: the error goes like h^{d+1} for elements of degree d. Also L^2 norms based on the `errornorm` gives the expected h^{d+1} rate for u and h^d for ∇u.

However, using `(u - u_exact)**2` for the error computation, which implies interpolating `u_exact` onto the same space as `u`, results in h^4 convergence for P2 elements.

element	$n = 8$	$n = 16$	$n = 32$	$n = 64$	$n = 128$
P1	1.98	1.94	1.98	2.0	2.0
P2	3.98	3.95	3.99	3.99	3.99
P3	3.69	4.03	4.01	3.95	2.77

This is an example where it is important to interpolate `u_exact` to a higher-order space (polynomials of degree 3 are sufficient here) to avoid computing a too optimistic convergence rate.

Checking convergence rates is the next best method for verifying PDE codes (the best being a numerical solution without approximation errors as in Section 1.3.3 and many other places in this tutorial).

1.6 Multiple domain and boundaries

hpl 3: Need a little intro.

1.6.1 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

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

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

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

$$-\nabla^2 u = f \text{ in } \Omega, \tag{1.31}$$

$$u = u_0 \text{ on } \Gamma_D, \tag{1.32}$$

$$-\frac{\partial u}{\partial n} = g \text{ on } \Gamma_N\,. \tag{1.33}$$

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

$$\begin{aligned} f &= -6, \\ g &= \begin{cases} -4, & y = 1 \\ 0, & y = 0 \end{cases} \\ u_0 &= 1 + x^2 + 2y^2\,. \end{aligned}$$

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

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

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

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

and since $v = 0$ on Γ_D,

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

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

$$\int_\Omega \nabla u\cdot\nabla v\,\mathrm{d}x + \int_{\Gamma_N} gv\,\mathrm{d}s = \int_\Omega fv\,\mathrm{d}x\,. \tag{1.34}$$

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

$$a(u, v) = \int_\Omega \nabla u\cdot\nabla v\,\mathrm{d}x, \tag{1.35}$$

$$L(v) = \int_\Omega fv\,\mathrm{d}x - \int_{\Gamma_N} gv\,\mathrm{d}s\,. \tag{1.36}$$

Implementation. How does the Neumann condition impact the implementation? Let us go back to the very simplest file, `p2D_plain.py`, from Section 1.2.1, we realize that the statements remain almost the same. Only two adjustments are necessary:

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

The first adjustment can be coded as

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

A more compact implementation reads

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

Never use == for comparing real numbers!

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

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

Comparison of real numbers need to use tolerances! The values of the tolerances depend on the size of the numbers involved in arithmetic operations:

```
>>> abs(0.1+0.2 - 0.3)
5.551115123125783e-17
>>> abs(1.1+1.2 - 2.3)
0.0
>>> abs(10.1+10.2 - 20.3)
3.552713678800501e-15
>>> abs(100.1+100.2 - 200.3)
```

```
0.0
>>> abs(1000.1+1000.2 - 2000.3)
2.2737367544323206e-13
>>> abs(10000.1+10000.2 - 20000.3)
3.637978807091713e-12
```

For numbers around unity, tolerances as low as $3 \cdot 10^{-16}$ can be used (in fact, this tolerance is known as `DOLFIN_EPS` in the `dolfin` package), otherwise an appropriate tolerance must be found.

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

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

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

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

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

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

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

1.6.2 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 1.6.1 and define two separate functions for the two Dirichlet conditions:

$$
\begin{aligned}
-\nabla^2 u &= -6 \text{ in } \Omega, \\
u &= u_L \text{ on } \Gamma_0, \\
u &= u_R \text{ on } \Gamma_1, \\
-\frac{\partial u}{\partial n} &= g \text{ on } \Gamma_N \,.
\end{aligned}
$$

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 $g = -4y$.

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

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

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

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

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

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

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

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

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

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

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

Classes for marking Dirichlet boundaries. Instead of using a function like `left_boundary(x, on_boundary)` to mark a boundary, we can alternatively

use a class, which allows for more flexibility in more complicated problems. The class for marking a boundary is derived from class `SubDomain` and has a method `inside(self, x, on_boundary)` for the code that returns whether the `point` is on the boundary in question or not. Our previous `left_boundary` function takes this form in its class version:

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

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

1.6.3 Working with subdomains

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

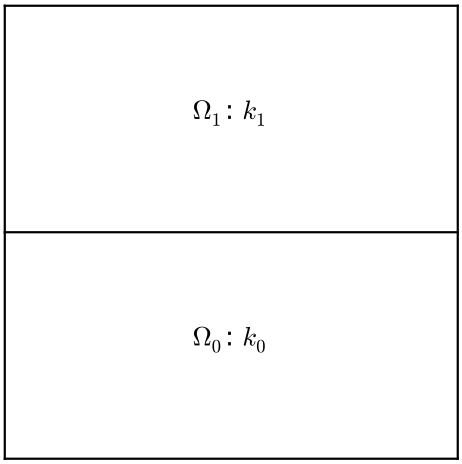


Figure 1.6: Medium with discontinuous material properties.

Suppose we want to solve

$$\nabla \cdot [k(x,y)\nabla u(x,y)] = 0, \tag{1.37}$$

in a domain Ω consisting of two subdomains where k takes on a different value in each subdomain. For simplicity, yet without loss of generality, we choose for

the current implementation the domain $\Omega = [0,1] \times [0,1]$ and divide it into two equal subdomains, as depicted in Figure 1.6,

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

We define $k(x,y) = k_0$ in Ω_0 and $k(x,y) = k_1$ in Ω_1, where $k_0 > 0$ and $k_1 > 0$ are given constants.

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

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

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

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

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

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

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

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

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

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

class Omega1(SubDomain):
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Having the `k` function ready for finite element computations, we can proceed in the normal manner with defining essential boundary conditions, as in Section 1.6.2, and the $a(u, v)$ and $L(v)$ forms, as in Section 1.3.6.

1.6.4 Multiple Neumann, Robin, and Dirichlet condition

Consider the model problem from Section 1.6.2 where we had both Dirichlet and Neumann conditions. The term `v*g*ds` in the expression for `L` implies a boundary integral over the complete boundary, or in FEniCS terms, an integral over all exterior facets. However, the contributions from the parts of the boundary where we have Dirichlet conditions are erased when the linear system is modified by the Dirichlet conditions. We would like, from an efficiency point of view, to integrate `v*g*ds` only over the parts of the boundary where we actually have Neumann conditions. And more importantly, in other problems one may have different Neumann conditions or other conditions like the Robin type condition. With the mesh function concept we can mark different parts of the boundary and integrate over specific parts. The same concept can also be used to treat multiple Dirichlet conditions. The forthcoming text illustrates how this is done.

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

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

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

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

Finally, we have *Robin conditions*

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

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

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

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

$$-\nabla\cdot(p\nabla u) = -f, \text{ in } \Omega, \tag{1.38}$$

$$u = u_{0,i} \text{ on } \Gamma_{D,i}, \quad i = 0, 1, \ldots \tag{1.39}$$

$$-p\frac{\partial u}{\partial n} = g_i \text{ on } \Gamma_{N,i}, \quad i = 0, 1, \ldots \tag{1.40}$$

$$-p\frac{\partial u}{\partial n} = r_i(u - s_i) \text{ on } \Gamma_{R,i}, \quad i = 0, 1, \ldots \tag{1.41}$$

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

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

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

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

The variational formulation then becomes

$$F = \int_\Omega p\nabla u\cdot\nabla v\,\mathrm{d}x + \sum_i \int_{\Gamma_{N,i}} g_i v\,\mathrm{d}s + \sum_i \int_{\Gamma_{R,i}} r_i(u-s_i)v\,\mathrm{d}s - \int_\Omega fv\,\mathrm{d}x = 0\,. \tag{1.42}$$

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

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

We then have

$$a(u,v) = \int_\Omega p\nabla u\cdot\nabla v\,\mathrm{d}x + \sum_i \int_{\Gamma_{R,i}} r_i uv\,\mathrm{d}s, \tag{1.43}$$

$$L(v) = \int_\Omega fv\,\mathrm{d}x - \sum_i \int_{\Gamma_{N,i}} g_i v\,\mathrm{d}s + \sum_i \int_{\Gamma_{R,i}} r_i s_i v\,\mathrm{d}s\,. \tag{1.44}$$

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

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

A general approach to the first task is to mark each of the desired boundaries with markers 0, 1, 2, and so forth. Here we aim at the four sides of the unit square, marked with 0 ($x = 0$), 1 ($x = 1$), 2 ($y = 0$), and 3 ($y = 1$). The marking of boundaries makes use of a mesh function object, but contrary to Section 1.6.3, this is not a function over cells, but a function over cell facets. We apply the `FacetFunction` for this purpose:

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

As in Section 1.6.3 we use a subclass of `SubDomain` to identify the various parts of the mesh function. Problems with domains of more complicated geometries may set the mesh function for marking boundaries as part of the mesh generation. In our case, the $x = 0$ boundary can be marked by

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

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

Similarly, we make the classes `BoundaryX1` for the $x = 1$ boundary, `BoundaryY0` for the $y = 0$ boundary, and `BoundaryY1` for the $y = 1$ boundary, and mark these as subdomains 1, 2, and 3, respectively.

For generality of the implementation, we let the user specify what kind of boundary condition that applies to each of the four boundaries. We set up a Python dictionary for this purpose, with the key as subdomain number and the value as a dictionary specifying the kind of condition as key and a function as its value. For example,

```
boundary_conditions = {
  0: {'Dirichlet': u0},
  1: {'Robin': (r, s)},
  2: {'Neumann: g}},
  3: {'Neumann', 0}}
```

specifies

- a Dirichlet condition, with values implemented by an `Expression` or `Constant` object `u0`, on subdomain 0, i.e., the $x = 1$ boundary;
- a Robin condition (1.6.4) on subdomain 1, $x = 1$, with `Expression` or `Constant` objects `r` and `s` specifying r and s;
- a Neumann condition $\partial u/\partial n = g$ on subdomain 2, $y = 0$, where an `Expression` or `Constant` object `g` implements the value g;
- a homogeneous Neumann condition $\partial u/\partial n = 0$ on subdomain 3, $y = 1$.

As explained in Section 1.6.2, multiple Dirichlet conditions must be collected in a list of `DirichletBC` objects. Based on the `boundary_conditions` data structure above, we can construct this list by the following snippet:

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

The new aspect of the variational problem is the two distinct boundary integrals over $\Gamma_{N,i}$ and $\Gamma_{R,i}$. Having a mesh function over exterior cell facets (our `boundary_parts` object), where subdomains (boundary parts) are numbered

as $0, 1, 2, \ldots$, the special symbol `ds(0)` implies integration over subdomain (part) 0, `ds(1)` denotes integration over subdomain (part) 1, and so on. The idea of multiple `ds`-type objects generalizes to volume integrals too: `dx(0)`, `dx(1)`, etc., are used to integrate over subdomain 0, 1, etc., inside Ω.

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

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

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

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

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

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

```
a = inner(nabla_grad(u), nabla_grad(v))*dx + r*u*v*ds(R)
L = f*v*dx - g*v*ds(N) + r*s*v*ds(R)
```

In our case things get a bit more complicated since the information about integrals in Neumann and Robin conditions are in the `boundary_conditions` data structure. We can collect all Neumann conditions by the code

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

Applying `sum(Nemann_integrals)` will apply the `+` operator to the variational forms in the `Numeann_integrals` list and result in the integrals we need for the right-hand side `L` of the variational form.

The integrals in the Robin condition can similarly be collected in lists:

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

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

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

Simplified handling of the variational formulation. We carefully ordered the terms in the variational formulation above into the a and L parts. This requires a splitting of the Robin condition and makes the `a` and `L` expressions less readable (still we think understanding this splitting is key for any finite element programmer!). Fortunately, UFL allow us to specify the complete variational form (1.42) as *one expression* and offer tools to extract what goes into the bilinear form $a(u,v)$ and the linear form $L(v)$:

```
F = inner(p*nabla_grad(u), nabla_grad(v))*dx + \
    sum(Robin_integrals) - f*v*dx + sum(Neumann_integrals)
a, L = lhs(F), rhs(F)
```

This time we can more naturally define the integrals from the Robin condition as `r*(u-s)*v*ds(n)`:

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

The complete code is in the `solver_bc` function in the `p2D_vc.py` file.

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

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

When implementing this test problem (and especially other test problems with more complicated expressions), it is advantageous to use symbolic computing. Below we define u_{e} as a `sympy` expression and derive other functions from their mathematical definitions. Then we turn these expressions into C/C++ code, which can be fed into `Expression` objects.

```
def application_bc_test():
    # Define manufactured solution in sympy and derive f, g, etc.
    import sympy as sym
    x, y = sym.symbols('x[0] x[1]')  # UFL needs x[0] for x etc.
```

```
u = 1 + x**2 + 2*y**2
f = -sym.diff(u, x, 2) - sym.diff(u, y, 2)  # -Laplace(u)
f = sym.simplify(f)
u_00 = u.subs(x, 0)  # x=0 boundary
u_01 = u.subs(x, 1)  # x=1 boundary
g = -sym.diff(u, y).subs(y, 1)  # x=1 boundary, du/dn=-du/dy
r = 1000 # any function can go here
s = u

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

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

boundary_conditions = {
    0: {'Dirichlet': u_00},   # x=0
    1: {'Dirichlet': u_01},   # x=1
    2: {'Robin': (r, s)},     # y=0
    3: {'Neumann': g}}        # y=1

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

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

```
    # Print numerical and exact solution at the vertices
    if u.function_space().dim() < 50:  # (small problems only)
        u_e_at_vertices = u_e.compute_vertex_values()
        u_at_vertices = u.compute_vertex_values()
        coor = u.function_space().mesh().coordinates()
        for i, x in enumerate(coor):
            print('vertex %2d (%9g,%9g): error=%g %g vs %g'
                  % (i, x[0], x[1],
                     u_e_at_vertices[i] - u_at_vertices[i],
                     u_e_at_vertices[i], u_at_vertices[i]))
```

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

Debugging the setting of boundary conditions. It is easy to make mistakes when implementing a problem with many different types of boundary conditions, as in the present case. Some helpful debugging output is to run through all vertex coordinates and check if the `SubDomain.inside` method marks the vertex as on the boundary. Another useful printout is to list which degrees of freedom that are subject to Dirichlet conditions, and for first-order Lagrange elements, add the corresponding vertex coordinate to the output.

```
if debug:
    # Print the vertices that are on the boundaries
    coor = mesh.coordinates()
    for x in coor:
        if bx0.inside(x, True): print('%s is on x=0' % x)
        if bx1.inside(x, True): print('%s is on x=1' % x)
        if by0.inside(x, True): print('%s is on y=0' % x)
        if by1.inside(x, True): print('%s is on y=1' % x)
    # Print the Dirichlet conditions
    print('No of Dirichlet conditions:', len(bcs))
    d2v = dof_to_vertex_map(V)
    for bc in bcs:
        bc_dict = bc.get_boundary_values()
        for dof in bc_dict:
            print('dof %2d: u=%g' % (dof, bc_dict[dof]))
            if V.ufl_element().degree() == 1:
                print('   at point %s' %
                      (str(tuple(coor[d2v[dof]].tolist()))))
```

In addition, it is helpful to print the exact and the numerical solution at all the vertices as shown in Section 1.3.3.

Implementation of multiple subdomains. Section 1.6.3 explains how to deal with multiple subdomains of Ω and a piecewise constant coefficient function p that takes on different constant values in the different subdomains. We can easily add this type of p coefficient to the `solver_bc` function. The signature of the function is

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

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

The appropriate code for computing `p` becomes

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

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

We define $p(x,y) = p_0$ in Ω_0 and $k(x,y) = p_1$ in Ω_1, where $p_0 > 0$ and $p_1 > 0$ are given constants. As boundary conditions, we choose $u = 0$ at $y = 0$, $u = 1$

at $y = 1$, and $\partial u/\partial n = 0$ at $x = 0$ and $x = 1$. One can show that the exact solution is now given by

$$u(x,y) = \begin{cases} \frac{2yp_1}{p_0+p_1}, & y \leq 1/2 \\ \frac{(2y-1)p_0+p_1}{p_0+p_1}, & y \geq 1/2 \end{cases} \tag{1.45}$$

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

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

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

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

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

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

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

            # Compute max error in infinity norm
            u_e = interpolate(u_exact, u.function_space())
            import numpy as np
            max_error = np.abs(u_e.vector().array() -
                               u.vector().array()).max()
```

```
            assert max_error < tol, 'max error: %g' % max_error
```

1.6.5 Refactoring of a solver function into solver and problem classes

A FEniCS solver for a PDE can be implemented in a general way, but the problem-dependent data, like boundary conditions, must be specified in each case by the user. The implementation in the previous section required the user to supply a `boundary_conditions` dictionary with specifications of the boundary condition on each of the four sides of the unit square. If we, e.g., want two Dirichlet conditions at one side, as our mathematical formulation of the problem in the previous section in fact supports, this is not possible without extending the `solver_bc` function.

A different software design is to introduce a problem class and methods, supplied by the user from case to case, where boundary conditions and other input data are defined. Such a design is used in a lot of more advanced FEniCS application codes, and it is time to exemplify it here. As a counterpart to the solver function, we introduce a solver class, but all the arguments for various input data are instead method calls to an instance of a *problem class*. This puts a somewhat greater burden on the programmer, but it allows for more flexibility, and the code for, e.g., boundary conditions can be more tailored to the problem at hand than the code we introduced in the `solver_bc` function in the previous section.

The solver class will need problem information and for this purpose call up the methods in a problem class. For example, the solver gets the f and p functions in the PDE problem by calling `problem.f_rhs()` and `problem.p_coeff()`. The mesh object and the polynomial degree of the elements are supposed to be returned from `problem.mesh_degree()`. Furthermore, the problem class defines the boundary conditions in the problem as lists of minimal information from which the solver can build proper DOLFIN data structures.

The solver class is a wrapping of the previous `solver_bc` and `flux` functions as methods in a class, but some of the code for handling boundary conditions in `solver_bc` is now delegated to the user in the problem class.

```
from dolfin import *
import numpy as np

class Solver(object):
    def solve(self, problem, linear_solver='direct',
              abs_tol=1E-6, rel_tol=1E-5, max_iter=1000):
        mesh, degree = problem.mesh_degree()
        V = FunctionSpace(mesh, 'Lagrange', degree)
        bcs = [DirichletBC(V, value, boundaries, index)
               for value, boundaries, index
               in problem.Dirichlet_conditions()]
```

```
        u = TrialFunction(V)
        v = TestFunction(V)
        p = problem.p_coeff()
        self.p = p  # store for flux computations
        F = inner(p*nabla_grad(u), nabla_grad(v))*dx
        F -= sum([g*v*ds_
                  for g, ds_ in problem.Neumann_conditions()])
        F += sum([r*(u-s)*ds_
                  for r, s, ds_ in problem.Robin_conditions()])
        a, L = lhs(F), rhs(F)

        # Compute solution
        self.u = Function(V)

        if linear_solver == 'Krylov':
            prm = parameters['krylov_solver'] # short form
            prm['absolute_tolerance'] = abs_tol
            prm['relative_tolerance'] = rel_tol
            prm['maximum_iterations'] = max_iter
            solver_parameters = {'linear_solver': 'gmres',
                                 'preconditioner': 'ilu'}
        else:
            solver_parameters = {'linear_solver': 'lu'}

        solve(a == L, self.u, bcs, solver_parameters=solver_parameters)
        return self.u

    def flux(self):
        """Compute and return flux -p*grad(u)."""
        mesh = self.u.function_space().mesh()
        degree = self.u.ufl_element().degree()
        V_g = VectorFunctionSpace(mesh, 'Lagrange', degree)
        self.flux_u = project(-self.p*grad(self.u), V_g)
        self.flux_u.rename('flux(u)', 'continuous flux field')
        return self.flux_u
```

Note that this is a general Poisson problem solver that works in any number of space dimensions and with any mesh and composition of boundary conditions.

Tip: Be careful with the `mesh` variable!

In classes, one often stores the mesh in `self.mesh`. When you need the mesh, it is easy to write just `mesh`, but this gives rise to peculiar error messages, since `mesh` is a module imported in `from dolfin import *` and already available as a name in the file. When encountering strange error messages in statements containing a variable `mesh`, make sure you use `self.mesh`.

Below is the specific problem class for solving a scaled 2D Poisson problem. We have a two-material domain where a rectangle $[0.3, 0.7] \times [0.3, 0.7]$ is embedded in the unit square and where p has a constant value inside the rectangle and another value outside. On $x = 0$ and $x = 1$ we have homogeneous Neumann conditions, and on $y = 0$ and $y = 1$ we have the Dirichlet conditions $u = 1$ and $u = 0$, respectively.

```
class TestProblem1(Problem):
    def init_mesh(self, Nx, Ny):
        """Initialize mesh, boundary parts, and p."""
        self.mesh = UnitSquareMesh(Nx, Ny)

        tol = 1E-14

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

        class BoundaryX1(SubDomain):
            def inside(self, x, on_boundary):
                return on_boundary and abs(x[0] - 1) < tol

        class BoundaryY0(SubDomain):
            def inside(self, x, on_boundary):
                return on_boundary and abs(x[1]) < tol

        class BoundaryY1(SubDomain):
            def inside(self, x, on_boundary):
                return on_boundary and abs(x[1] - 1) < tol

        # Mark boundaries
        #self.boundary_parts = FacetFunction('size_t', mesh)
        self.boundary_parts = FacetFunction('uint', self.mesh)
        self.boundary_parts.set_all(9999)
        self.bx0 = BoundaryX0()
        self.bx1 = BoundaryX1()
        self.by0 = BoundaryY0()
        self.by1 = BoundaryY1()
        self.bx0.mark(self.boundary_parts, 0)
        self.bx1.mark(self.boundary_parts, 1)
        self.by0.mark(self.boundary_parts, 2)
        self.by1.mark(self.boundary_parts, 3)
        self.ds =  Measure(
            'ds', domain=self.mesh,
            subdomain_data=self.boundary_parts)

        # The domain is the unit square with an embedded rectangle
        class Rectangle(SubDomain):
            def inside(self, x, on_boundary):
```

```
                return 0.3 <= x[0] <= 0.7 and 0.3 <= x[1] <= 0.7

        self.materials = CellFunction('size_t', self.mesh)
        self.materials.set_all(0)  # "the rest"
        subdomain = Rectangle()
        subdomain.mark(self.materials, 1)
        self.V0 = FunctionSpace(self.mesh, 'DG', 0)
        self.p = Function(self.V0)
        help = np.asarray(self.materials.array(), dtype=np.int32)
        p_values = [1, 1E-3]
        self.p.vector()[:] = np.choose(help, p_values)

    def mesh_degree(self):
        return self.mesh, 2

    def p_coeff(self):
        return self.p

    def f_rhs(self):
        return Constant(0)

    def Dirichlet_conditions(self):
        """Return list of (value,boundary) pairs."""
        return [(1.0, self.boundary_parts, 2),
                (0.0, self.boundary_parts, 3)]

    def Neumann_conditions(self):
        """Return list of g*ds(n) values."""
        return [(0, self.ds(0)), (0, self.ds(1))]
```

A specific problem can be solved by

```
def demo():
    problem = TestProblem1()
    problem.init_meshNx=20, Ny=20)
    problem.solve()
    u = problem.solution()
    plot(u)
    flux_u = problem.solver.flux()
    plot(flux_u)
    interactive()
```

The complete code is found in the file `p2D_class.py`.

Pros of cons of solver/problem classes versus solver function.

What are the advantages of class `Solver` and `Problem` over the function implementation in Section 1.6.4? The primary advantage is that the class

version works for any mesh and any composition of boundary conditions, while the solver function is tied to a mesh over the unit square, only one type of boundary condition on a each side, and a piecewise constant p function. The programmer has to supply more code in the class version, but gets greater flexibility. The disadvantage of the class version is that it applies the class concept and one needs experience with Python class programming.

1.6.6 Handy methods in key FEniCS objects

In general, `pydoc dolfin.X` shows the documentation of any DOLFIN name `X` and lists all the methods (i.e,g, functions in the class) that can be called. Below, we list just a few, but very useful, methods in the most central FEniCS classes.

Mesh. Let `mesh` be a `Mesh` object.

- `mesh.coordinates()` returns an array of the coordinates of the vertices in the mesh.
- `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, `len(u_array)`, or dimension of the space `V.dim()`),
- `mesh.cells()` returns the vertex numbers of the vertices in each cell as a `numpy` array with shape (*number of cells*, *number of vertices in a cell*),
- `mesh.hmin()` returns the minimum cell diameter ("smallest cell"),
- `mesh.hmax()` returns the maximum cell diameter ("largest cell").
- `mesh.topology().dim()` returns the number of physical dimensions of the mesh.

Writing `print(mesh)` dumps a short, pretty-print description of the mesh (`print(mesh)` actually displays the result of `str(mesh)`, which defines the pretty print):

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

Function space. Let `V` be a `FunctionSpace` object.

- `V.mesh()` returns the associated mesh.
- `V.dim()` returns the dimension (number of degrees of freedom).
- `V.ufl_element()` returns the associated finite element.

Function. Let `u` be a `Function` object.

- `u.function_space()` returns the associated function space.
- `u.vector()` returns the DOLFIN vector of degrees of freedom.
- `u.vector().array()` returns a copy of the degrees of freedom in a `numpy` array.

Chapter 2

Time-dependent and nonlinear problems

2.1 Time-dependent problems

The examples in Section 1.2 illustrate that solving linear, stationary PDE problems with the aid of FEniCS is easy and requires little programming. FEniCS clearly automates the spatial discretization by the finite element method. One can use a separate, one-dimensional finite element method in the domain as well, but very often, it is easier to just use a finite difference method, or to formulate the problem as an ODE system and leave the time-stepping to an ODE solver.

hpl 13: Should exemplify all three approaches? With emphasis on simple finite differences?

2.1.1 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, i.e., the diffusion problem

$$\frac{\partial u}{\partial t} = \nabla^2 u + f \text{ in } \Omega, \text{ for } t > 0, \tag{2.1}$$

$$u = u_0 \text{ on } \partial\Omega, \text{ for } t > 0, \tag{2.2}$$

$$u = I \text{ at } t = 0\,. \tag{2.3}$$

Here, u varies with space and time, e.g., $u = u(x, y, t)$ if the spatial domain Ω is two-dimensional. The source function f and the boundary values u_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:

$$\frac{\partial}{\partial t}u^k = \nabla^2 u^k + f^k\,. \tag{2.4}$$

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

$$\frac{\partial}{\partial t}u^k \approx \frac{u^k - u^{k-1}}{\Delta t}, \tag{2.5}$$

where Δt is the time discretization parameter. Inserting (2.5) in (2.4) yields

$$\frac{u^k - u^{k-1}}{\Delta t} = \nabla^2 u^k + f^k\,. \tag{2.6}$$

This is our time-discrete version of the diffusion PDE (2.1).

We reorder (2.6) so that the left-hand side contains the terms with the unknown u^k and the right-hand side contains computed terms only. The result is a recursive set of spatial (stationary) problems for u^k (assuming u^{k-1} is known from computations at the previous time level):

$$u^0 = I, \tag{2.7}$$

$$u^k - \Delta t\nabla^2 u^k = u^{k-1} + \Delta t f^k, \quad k = 1, 2, \ldots \tag{2.8}$$

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 (2.7) and (2.8). 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 (2.7) and

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

for (2.8), where

$$a_0(u,v) = \int_\Omega uv\,\mathrm{d}x, \tag{2.9}$$

$$L_0(v) = \int_\Omega Iv\,\mathrm{d}x, \tag{2.10}$$

$$a(u,v) = \int_\Omega (uv + \Delta t\nabla u\cdot\nabla v)\,\mathrm{d}x, \tag{2.11}$$

$$L(v) = \int_\Omega \left(u^{k-1} + \Delta t f^k\right) v\,\mathrm{d}x\,. \tag{2.12}$$

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 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 1.3.5, except that the test and trial functions are now scalar fields and not vector fields. Instead of solving (2.7) by a finite element method, i.e., 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.

2.1.2 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 demonstrated in Section 1.3.7. 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.)
- let u_1 interpolate I or be the projection of I

- define a and L
- assemble matrix A from a
- set some stopping time T
- $t = \Delta t$
- while $t \leq T$
  - assemble vector b from L
  - apply essential boundary conditions
  - solve $AU = b$ for U and store in u
  - $t \leftarrow t + \Delta t$
  - $u_1 \leftarrow u$ (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,

$$u = 1 + x^2 + \alpha y^2 + \beta t, \tag{2.13}$$

yields a function whose computed values at the nodes will be exact, regardless of the size of the elements and Δt, as long as the mesh is uniformly partitioned. By inserting (2.13) in the PDE problem (2.1), it follows that u_0 must be given as (2.13) 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 (2.13). 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 1.4.1 for `Expression` objects with parameters):

```
alpha = 3; beta = 1.2
u0 = Expression('1 + x[0]*x[0] + alpha*x[1]*x[1] + beta*t',
                {'alpha': alpha, 'beta': beta})
u0.t = 0
```

This function expression has the components of `x` as independent variables, while `alpha`, `beta`, and `t` are parameters. The parameters can either be set through a dictionary at construction time, as demonstrated for `alpha` and `beta`, or anytime through attributes in the function object, as shown for the `t` parameter.

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

```
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

```
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 1.3.5 and form the associated variational problem.

Projecting versus interpolating the initial condition.

To actually recover the exact solution (2.13) 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:

```
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:

```
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)
```

Remember to update expression objects with the current time!

Inside the time loop, 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:

```
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

```
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 goes as follows:

```
from dolfin import *
import numpy

# Create mesh and define function space
nx = ny = 2
mesh = UnitSquareMesh(nx, ny)
V = FunctionSpace(mesh, 'Lagrange', 1)

# Define boundary conditions
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)

class Boundary(SubDomain):  # define the Dirichlet boundary
    def inside(self, x, on_boundary):
        return on_boundary

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

```
# Initial condition
u_1 = interpolate(u0, V)
#u_1 = project(u0, V)  # will not result in exact solution!

dt = 0.3      # time step

# Define variational problem
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
b = None          # necessary for memory saving assemeble call

# Compute solution
u = Function(V)   # the unknown at a new time level
T = 1.9           # total simulation time
t = dt
while t <= T:
    print('time =', t)
    b = assemble(L, tensor=b)
    u0.t = t
    bc.apply(A, b)
    solve(A, u.vector(), b)

    # Verify
    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))

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

The code is available in the file `d2D_plain.py`[1] in the directory `diffusion`[2].

2.1.3 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 $\alpha \geq 1$. Typically, $\alpha \in [1, 2]$. As $N \to \infty$, the solve operation will dominate for $\alpha > 1$, but for the values of N typically used on

[1] `https://github.com/hplgit/fenics-tutorial/blob/master/src/diffusion/d2D_plain.py`
[2] `https://github.com/hplgit/fenics-tutorial/blob/master/src/diffusion`

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.

Deriving recursive linear systems. To see how repeated assembly can be avoided, we look at the $L(v)$ form in (2.12), which in general varies with time through u^{k-1}, f^k, and possibly also with Δt 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 1.3.7, 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

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

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

$$\sum_{j=1}^N M_{ij}U_j^{k-1} + \Delta t \sum_{j=1}^N M_{ij}F_j^k,$$

which is nothing but two matrix-vector products,

$$MU^{k-1} + \Delta t MF^k,$$

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

$$U^{k-1} = (U_1^{k-1}, \ldots, U_N^{k-1})^T,$$

and

$$F^k = (F_1^k, \ldots, F_N^k)^T\,.$$

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

$$b = MU^{k-1} + \Delta t MF^k\,.$$

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 (2.11) to get

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

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

$$MU^k + \Delta t K U^k = (M + \Delta t K) U^k,$$

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

$$K_{ij} = \int_\Omega \nabla \hat{\phi}_i \cdot \nabla \phi_j \,\mathrm{d}x \,. \tag{2.14}$$

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

$$a_K(u, v) = \int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x, \tag{2.15}$$

$$a_M(u, v) = \int_\Omega uv \,\mathrm{d}x \,. \tag{2.16}$$

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 $A = M + \Delta t K$ at $t = 0$, while b is computed as $b = MU^{k-1} + \Delta t M F^k$ at each time level.

Implementation. 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 $A = M + \Delta t$, K
4. Define f as an `Expression`
5. Interpolate the formula for f to a finite element function F^k
6. Compute $b = MU^{k-1} + \Delta t M F^k$

The relevant code segments become

```
# 1.
a_K = inner(nabla_grad(u), nabla_grad(v))*dx
a_M = u*v*dx
# No need for L

# 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 = f_k.vector()
    b = M*u_1.vector() + dt*M*F_k
```

We implement these modification in a refactored version of the program `d2D_plain.py`, where the solver is a function as explained in Section 1.2.4 rather than a flat program.

```
def solver_minimize_assembly(
    f, u0, I, dt, T, Nx, Ny, degree=1,
    user_action=None, I_project=False):
    # Create mesh and define function space
    mesh = UnitSquareMesh(Nx, Ny)
    V = FunctionSpace(mesh, 'Lagrange', degree)

    class Boundary(SubDomain):  # define the Dirichlet boundary
        def inside(self, x, on_boundary):
            return on_boundary

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

    # Initial condition
    u_1 = project(I, V) if I_project else interpolate(I, V)
    user_action(0, u_1, V)

    # Define variational problem
    u = TrialFunction(V)
    v = TestFunction(V)
    a_M = u*v*dx
    a_K = inner(nabla_grad(u), nabla_grad(v))*dx

    M = assemble(a_M)
    K = assemble(a_K)
```

```
    A = M + dt*K
    # Compute solution
    u = Function(V)   # the unknown at a new time level
    t = dt
    while t <= T:
        f_k = interpolate(f, V)
        F_k = f_k.vector()
        b = M*u_1.vector() + dt*M*F_k
        try:
            u0.t = t
        except AttributeError:
            pass  # ok if no t attribute in u0
        bc.apply(A, b)
        solve(A, u.vector(), b)

        user_action(t, u, V)
        t += dt
        u_1.assign(u)
```

A special feature in this program is the `user_action` callback function: at every time level, the solution is sent to `user_action`, which is some function provided by the user where the solution can be processed, e.g., stored, analyzed, or visualized. In a unit test for the test example without numerical approximation errors, we can write a call to the solver function,

```
def test_solver():
    import numpy as np
    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)
    f = Constant(beta - 2 - 2*alpha)
    dt = 0.3; T = 1.9
    u0.t = 0
    solver_minimize_assembly(
        f, u0, u0, dt, T, Nx, Ny, degree,
        user_action=assert_max_error, I_project=False)
```

The `user_action` function asserts equality of the exact and numerical solution at every time level:

```
def assert_max_error(t, u, V):
    u_e = interpolate(u0, V)
    max_error= np.abs(u_e.vector().array() -
                      u.vector().array()).max()
    tol = 2E-12
    assert max_error < tol, 'max_error: %g' % max_error
```

2.1.4 A physical example

With the basic programming techniques for time-dependent problems from Sections 2.1.3 and 2.1.2 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

$$T_0(t) = T_R + T_A \sin(\omega t),$$

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, i.e., 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 2.1 shows a sketch of the problem, with a small region where the heat conductivity is much lower.

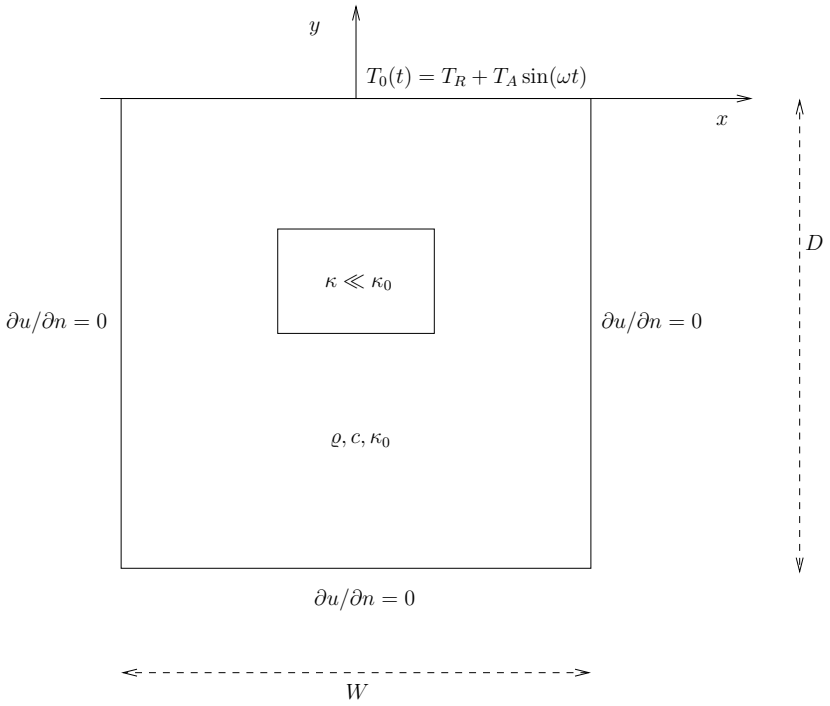


Figure 2.1: Sketch of a (2D) problem involving heating and cooling of the ground due to an oscillating surface temperature.

The initial-boundary value problem for this problem reads

$$\varrho c \frac{\partial T}{\partial t} = \nabla \cdot (\kappa \nabla T) \text{ in } \Omega \times (0, t_{\text{stop}}], \tag{2.17}$$

$$T = T_0(t) \text{ on } \Gamma_0, \tag{2.18}$$

$$\frac{\partial T}{\partial n} = 0 \text{ on } \partial\Omega \backslash \Gamma_0, \tag{2.19}$$

$$T = T_R \text{ at } t = 0\,. \tag{2.20}$$

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, i.e., the evolution equation $\partial P/\partial t = Q(t)$ is discretized as

$$\frac{P^k - P^{k-1}}{\Delta t} = \theta Q^k + (1-\theta)Q^{k-1},$$

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

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

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

$$a(T, v) = \int_\Omega \left(\varrho c T v + \theta \Delta t \kappa \nabla T \cdot \nabla v\right)\, \mathrm{d}x, \tag{2.21}$$

$$L(v) = \int_\Omega \left(\varrho c T^{k-1} v - (1-\theta)\Delta t \kappa \nabla T^{k-1} \cdot \nabla v\right)\, \mathrm{d}x\,. \tag{2.22}$$

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 `BoxMesh`, `RectangleMesh`, and `IntervalMesh` at our disposal. The mesh and the function space can be created by the following code:

```
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 = IntervalMesh(divisions[0], -D, 0)
elif d == 2:
    mesh = RectangleMesh(-W/2, -D, W/2, 0, divisions[0], divisions[1])
elif d == 3:
    mesh = BoxMesh(-W/2, -W/2, -D, W/2, W/2, 0,
                   divisions[0], divisions[1], divisions[2])
V = FunctionSpace(mesh, 'Lagrange', degree)
```

The `RectangleMesh` and `BoxMesh` objects are defined by the coordinates of the "minimum" and "maximum" corners.

Setting Dirichlet conditions at the upper boundary can be done by

```
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 2.1. 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 $\kappa(\boldsymbol{x})$ 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`:

```
class Kappa(Expression):
    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 much more complicated and not covered here.) Using inline if-tests in C++, we can make string expressions for κ:

```
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:

```
T_prev = 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_1), 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 2.1.3, 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 2.1.2:

```
T = Function(V)   # unknown at the current time level
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,

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

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

```
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`:

```
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 complete code is found in the file 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, $\kappa_0 = 2.3$ $\mathrm{K}^{-1}\mathrm{Ns}^{-1}$, $\kappa_1 = 100$ $\mathrm{K}^{-1}\mathrm{Ns}^{-1}$, $\varrho = 1500$ $\mathrm{kg/m}^3$, $c = 1480$ $\mathrm{Nm}\cdot\mathrm{kg}^{-1}\mathrm{K}^{-1}$, $\omega = 2\pi/24$ 1/h $= 7.27\cdot 10^{-5}$ 1/s, $D = 1.5$ m
5. As above, but $\kappa_0 = 12.3$ $\mathrm{K}^{-1}\mathrm{Ns}^{-1}$ and $\kappa_1 = 10^4$ $\mathrm{K}^{-1}\mathrm{Ns}^{-1}$

Data set number 4 is relevant for real temperature variations in the ground (not necessarily the large value of κ_1), while data set number 5 exaggerates the effect of a large heat conduction contrast so that it becomes clearly visible in an animation.

2.2 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:

$$-\nabla\cdot(q(u)\nabla u) = f\,. \tag{2.23}$$

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

$$u(x_0,\ldots,x_{d-1}) = \left((2^{m+1}-1)x_0 + 1\right)^{1/(m+1)} - 1\,. \tag{2.24}$$

We refer to Section 1.3.4 for details on formulating a PDE problem in d space dimensions.

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

$$F(u;v) = 0 \quad \forall v \in \hat{V}, \tag{2.25}$$

where

$$F(u;v) = \int_\Omega q(u)\nabla u\cdot\nabla v\,\mathrm{d}x, \tag{2.26}$$

and

$$\hat{V} = \{v \in H^1(\Omega) : v = 0 \text{ on } x_0 = 0 \text{ and } x_0 = 1\},$$
$$V = \{v \in H^1(\Omega) : v = 0 \text{ on } x_0 = 0 \text{ and } v = 1 \text{ on } x_0 = 1\}\,.$$

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 with some element type. Similarly, we let u be the discrete solution and use u_{e} for the exact solution if it becomes necessary to distinguish between the two.

The discrete nonlinear problem is then written as: find $u \in V$ such that

$$F(u; v) = 0 \quad \forall v \in \hat{V}, \tag{2.27}$$

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

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 manual control of the solution process for nonlinear equations (which also has some pedagogical advantages, especially for newcomers to nonlinear finite element problems).

2.2.1 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*,

$$\nabla \cdot \left(q(u^k)\nabla u^{k+1}\right) = 0, \quad k = 0, 1, \ldots \tag{2.28}$$

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 (2.28). Equivalently, we can approximate $q(u)$ by $q(u^k)$ in (2.26) to obtain the same linear variational problem. In both cases, the problem consists of seeking $u^{k+1} \in V$ such that

$$\tilde{F}(u^{k+1}; v) = 0 \quad \forall v \in \hat{V}, \quad k = 0, 1, \ldots, \tag{2.29}$$

with

$$\tilde{F}(u^{k+1}; v) = \int_\Omega q(u^k)\nabla u^{k+1} \cdot \nabla v \, \mathrm{d}x \,. \tag{2.30}$$

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

$$a(u^{k+1}, v) = L(v), \tag{2.31}$$

with

$$a(u, v) = \int_\Omega q(u^k)\nabla u \cdot \nabla v \, \mathrm{d}x \tag{2.32}$$

$$L(v) = 0 \,. \tag{2.33}$$

The iterations can be stopped when $\epsilon \equiv ||u^{k+1} - u^k|| < \text{tol}$, where tol is a small tolerance, 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:

```
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` contains the complete code for this nonlinear Poisson problem. The implementation is d dimensional, with mesh construction and setting of Dirichlet conditions as explained in Section 1.3.4. For a 33×33 grid with $m = 2$ we need 9 iterations for convergence when the tolerance is 10^{-5}.

2.2.2 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 (2.25), the discrete version (2.27) 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 (2.27)):

$$F_i(U_1, \ldots, U_N) \equiv \sum_{j=1}^N \int_\Omega \left(q\left(\sum_{\ell=1}^N U_\ell \phi_\ell\right) \nabla\phi_j U_j \right) \cdot \nabla\hat{\phi}_i \,\mathrm{d}x = 0, \quad i = 1, \ldots, N\,. \tag{2.34}$$

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

$$\sum_{j=1}^N \frac{\partial}{\partial U_j} F_i(U_1^k, \ldots, U_N^k)\delta U_j = -F_i(U_1^k, \ldots, U_N^k), \quad i = 1, \ldots, N, \tag{2.35}$$

$$U_j^{k+1} = U_j^k + \omega\delta U_j, \quad j = 1, \ldots, N, \tag{2.36}$$

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. It means that one takes a smaller step than what is suggested by Newton's method.

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 (2.34). The derivative $\partial F_i/\partial U_j$ becomes

$$\int_\Omega \left[q'(\sum_{\ell=1}^N U_\ell^k \phi_\ell)\phi_j \nabla(\sum_{j=1}^N U_j^k \phi_j) \cdot \nabla\hat{\phi}_i + q\left(\sum_{\ell=1}^N U_\ell^k \phi_\ell\right) \nabla\phi_j \cdot \nabla\hat{\phi}_i \right] \mathrm{d}x\,. \tag{2.37}$$

The following results were used to obtain (2.37):

$$\frac{\partial u}{\partial U_j} = \frac{\partial}{\partial U_j}\sum_{j=1}^{N} U_j\phi_j = \phi_j, \quad \frac{\partial}{\partial U_j}\nabla u = \nabla\phi_j, \quad \frac{\partial}{\partial U_j}q(u) = q'(u)\phi_j\,. \tag{2.38}$$

We can reformulate the Jacobian matrix in (2.37) by introducing the short notation $u^k = \sum_{j=1}^{N} U_j^k\phi_j$:

$$\frac{\partial F_i}{\partial U_j} = \int_\Omega \left[q'(u^k)\phi_j\nabla u^k\cdot\nabla\hat{\phi}_i + q(u^k)\nabla\phi_j\cdot\nabla\hat{\phi}_i\right]\,\mathrm{d}x\,. \tag{2.39}$$

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 linear discrete weak form to be solved in each Newton iteration:

$$\int_\Omega \left[q'(u^k)\delta u\nabla u^k\cdot\nabla v + q(u^k)\nabla\delta u\cdot\nabla v\right]\,\mathrm{d}x = -\int_\Omega q(u^k)\nabla u^k\cdot\nabla v\,\mathrm{d}x\,. \tag{2.40}$$

This variational form fits the standard notation $a(\delta u, v) = L(v)$ with

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

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 $\nabla^2 u^0 = 0$. The recipe for solving this problem appears in Sections 1.1.2, 1.2, and 1.6.1. The code for computing u^0 becomes as follows:

```
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, i.e., 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 δu, in 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 boundary values. We therefore define an additional list of Dirichlet boundary conditions objects for δu:

```
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:

```
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 2.2.1:

```
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:

```
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 1.3.4. The complete program appears in the file `alg_newton_np.py`.

2.2.3 A Newton method at the PDE level

Although Newton's method in PDE problems is normally formulated at the linear algebra level, i.e., 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 in Section 2.2.2.

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

$$u^{k+1} = u^k + \delta u \tag{2.41}$$

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

$$q(u^{k+1}) = q(u^k) + q'(u^k)\delta u + \mathcal{O}((\delta u)^2) \approx q(u^k) + q'(u^k)\delta u, \tag{2.42}$$

and dropping 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,

$$\nabla\cdot\left(q(u^k)\nabla\delta u\right) + \nabla\cdot\left(q'(u^k)\delta u\nabla u^k\right) = -\nabla\cdot\left(q(u^k)\nabla u^k\right), \tag{2.43}$$

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

$$\int_\Omega \left(q(u^k)\nabla\delta u\cdot\nabla v + q'(u^k)\delta u\nabla u^k\cdot\nabla v\right)\,\mathrm{d}x = -\int_\Omega q(u^k)\nabla u^k\cdot\nabla v\,\mathrm{d}x\,. \tag{2.44}$$

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

$$a(\delta u, v) = \int_\Omega \left(q(u^k)\nabla\delta u\cdot\nabla v + q'(u^k)\delta u\nabla u^k\cdot\nabla v\right)\,\mathrm{d}x, \tag{2.45}$$

$$L(v) = -\int_\Omega q(u^k)\nabla u^k\cdot\nabla v\,\mathrm{d}x\,. \tag{2.46}$$

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

$$a_0(u, v) = \int_\Omega \nabla u\cdot\nabla v\,\mathrm{d}x, \quad L_0(v) = 0\,.$$

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 (2.45) and (2.46), we see that the variational form is the same as for the Newton method at the algebraic level in Section 2.2.2. 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 this author finds more straightforward, although not so commonly documented in the literature on numerical methods for PDEs. 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 2.2.2 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 2.2.2.

2.2.4 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

```
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 (in general a 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 (2.26) is straightforwardly defined as follows, assuming `q(u)` is coded as a Python function:

```
u_ = Function(V)     # most recently computed solution
v  = TestFunction(V)
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 u and a test function for v, and then create the proper `F` by

```
u  = TrialFunction(V)
v  = TestFunction(V)
F  = inner(q(u)*nabla_grad(u), nabla_grad(v))*dx
u_ = Function(V)     # the 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. (Note that u^k and u^{k+1} in the previous notation correspond to u_- and u in the present notation. We have changed notation to better align the mathematics with the associated UFL code.)

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

$$\lim_{\epsilon\to 0}\frac{d}{d\epsilon}F_i(u_- + \epsilon\delta u;v)\,. \tag{2.47}$$

The δu is now the trial function and u_- is the previous approximation to the solution u. We start with

$$\frac{d}{d\epsilon}\int_\Omega \nabla v\cdot(q(u_- + \epsilon\delta u)\nabla(u_- + \epsilon\delta u))\,\mathrm{d}x$$

and obtain

$$\int_\Omega \nabla v\cdot[q'(u_- + \epsilon\delta u)\delta u\nabla(u_- + \epsilon\delta u) + q(u_- + \epsilon\delta u)\nabla\delta u]\,\mathrm{d}x,$$

which leads to

$$\int_\Omega \nabla v\cdot[q'(u_-)\delta u\nabla(u_-) + q(u_-)\nabla\delta u]\,\mathrm{d}x, \tag{2.48}$$

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`:

```
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

```
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:

```
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

```
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 with the aid of the `derivative` function is most naturally matched by 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`. The directory `nonlinear_poisson` contains both files. 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 here 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 iterations.

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

prm = solver.parameters
info(prm, True)
prm_n = prm['newton_solver']
prm_n['absolute_tolerance'] = 1E-8
prm_n['relative_tolerance'] = 1E-7
prm_n['maximum_iterations'] = 25
prm_n['relaxation_parameter'] = 1.0
if iterative_solver:
    prec = 'jacobi' if 'jacobi' in \
           list(zip(*krylov_solver_preconditioners()))[0] \
           else 'ilu'
    prm_n['linear_solver'] = 'gmres'
    prm_n['preconditioner'] = prec
    prm_n['krylov_solver']['absolute_tolerance'] = 1E-9
    prm_n['krylov_solver']['relative_tolerance'] = 1E-7
    prm_n['krylov_solver']['maximum_iterations'] = 1000
    prm_n['krylov_solver']['monitor_convergence'] = True
    prm_n['krylov_solver']['nonzero_initial_guess'] = False
    prm_n['krylov_solver']['gmres']['restart'] = 40
    prm_n['krylov_solver']['preconditioner']['structure'] = \
```

```
                                        'same_nonzero_pattern'
    prm_n['krylov_solver']['preconditioner']['ilu']['fill_level'] = 0
PROGRESS = 16
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()`.

2.3 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.

2.3.1 Built-in mesh generation tools

DOLFIN has a few tools for creating various types of meshes over domains with simple shape: `UnitIntervalMesh`, `UnitSquareMesh`, `UnitCubeMesh`, `IntervalMesh`, `RectangleMesh`, and `BoxMesh`. 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:

```
# 1D domains
mesh = UnitIntervalMesh(20)     # 20 cells, 21 vertices
mesh = IntervalMesh(20, -1, 1)  # domain [-1,1]

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

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

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

# Domain [-1,1]x[-1,0]x[-1,2] with 6x10x5 divisions
```

```
mesh = BoxMesh(-1, -1, -1, 1, 0, 2, 6, 10, 5)
```

2.3.2 Transforming mesh coordinates

Coordinate stretching. 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,

$$\bar{x} = a + (b - a)\left(\frac{x - a}{b - a}\right)^s \tag{2.49}$$

toward $x = a$, or

$$\bar{x} = a + (b - a)\left(\frac{x - a}{b - a}\right)^{1/s} \tag{2.50}$$

toward $x = b$. Figure 2.2 shows the effect of making a rectangular mesh denser toward $x = 0$ (prior to the coordinate transformation below).

Rectangle to hollow circle mapping. 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 $a \leq \bar{x} \leq b$ and $0 \leq \bar{y} \leq 1$, the mapping

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

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:

```
Theta = pi/2
a, b = 1, 5.0
nr = 10  # divisions in r direction
nt = 20  # divisions in theta direction
mesh = RectangleMesh(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
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 2.2.

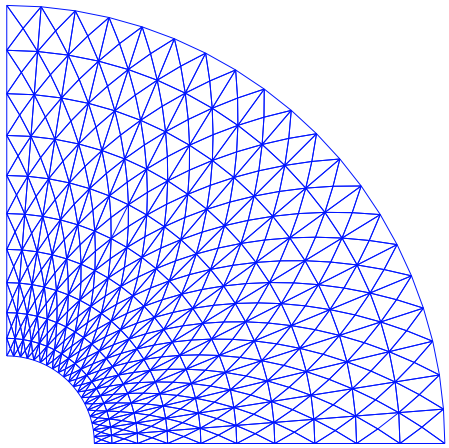


Figure 2.2: Hollow cylinder generated by mapping a rectangular mesh, stretched toward the left side.

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.

2.4 A General d-Dimensional multi-material test problem

This section is in a preliminary state!

The purpose of the present section is to generalize the basic ideas from the previous section to a problem involving an arbitrary number of materials in 1D, 2D, or 3D domains. The example also highlights how to build more general and flexible FEniCS applications.

More to be done:

- Batch compilation of subdomains, see mailinglist.txt, lots of useful stuff in Hake's example with "pointwise", see what the bcs are etc.
- Use of `near` or similar function (better: user-adjusted tolerance)

2.4.1 The PDE problem

We generalize the problem in Section 1.6.3 to the case where there are s materials $\Omega_0, \ldots, \Omega_{s-1}$, with associated constant k values $k_0, k_1, \ldots, k_{s-1}$, as illustrated in Figure 2.3.

Although the sketch of the domain is in two dimensions, we can easily define this problem in any number of dimensions, using the ideas of Section 1.3.4, but the layer boundaries are planes $x_0 = \text{const}$ and u varies with x_0 only.

The PDE reads

$$\nabla \cdot (k\nabla u) = 0\,. \tag{2.51}$$

To construct a problem where we can find an analytical solution that can be computed to machine precision regardless of the element size, we choose Ω as a hypercube $[0,1]^d$, and the materials as layers in the x_0 direction, as depicted in Figure 2.3 for a 2D case with four materials. The boundaries $x_0 = 0$ and $x_0 = 1$ have Dirichlet conditions $u = 0$ and $u = 1$, respectively, while Neumann conditions $\partial u/\partial n = 0$ are set on the remaining boundaries. The complete boundary-value problem is then

$$\begin{aligned} \nabla \cdot (k(x_0)\nabla u(x_0, \ldots, x_{d-1})) &= 0 \quad \text{in } \Omega, \\ u &= 0 \quad \text{on } \Gamma_0, \\ u &= 1 \quad \text{on } \Gamma_1, \\ \tfrac{\partial u}{\partial n} &= 0 \quad \text{on } \Gamma_N\,. \end{aligned} \tag{2.52}$$

The domain Ω is divided into s materials Ω_i, $i = 0, \ldots, s-1$, where

$$\Omega_i = \{(x_0, \ldots, x_{d-1}) \,|\, L_i \leq x_0 < L_{i+1}\}$$

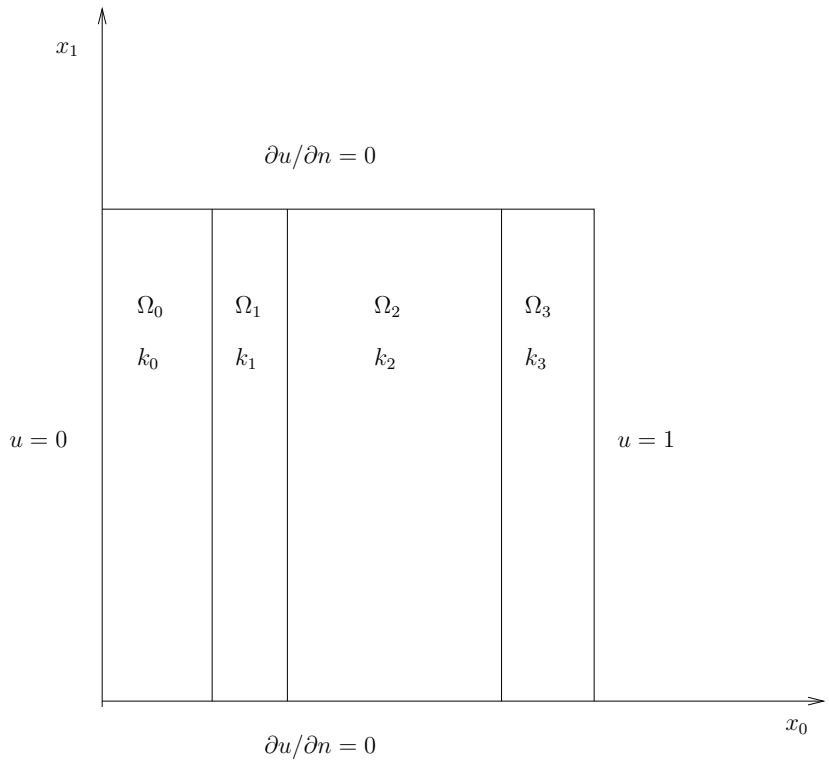


Figure 2.3: Sketch of a multi-material problem.

for given x_0 values $0 = L_0 < L_1 < \cdots < L_s = 1$ of the material (subdomain) boundaries. The $k(x_0)$ function takes on the value k_i in Ω_i.

The exact solution of the basic PDE in (2.52)

$$u(x_0, \ldots, x_{d-1}) = \frac{\int_0^{x_0} (k(\tau))^{-1} d\tau}{\int_0^1 (k(\tau))^{-1} d\tau}\,.$$

For a piecewise constant $k(x_0)$ as explained, we get

$$u(x_0, \ldots, x_{d-1}) = \frac{(x_0 - L_i)k_i^{-1} + \sum_{j=0}^{i-1}(L_{j+1} - L_j)k_j^{-1}}{\sum_{j=0}^{s-1}(L_{j+1} - L_j)k_j^{-1}}, \quad L_i \leq x_0 \leq L_{i+1}\,. \tag{2.53}$$

That is, $u(x_0, \ldots, x_{d-1})$ is piecewise linear in x_0 and constant in all other directions. If L_i coincides with the element boundaries, Lagrange elements will reproduce this exact solution to machine precision, which is ideal for a test case.

2.4.2 Preparing a mesh with subdomains

Our first task is to generate a mesh for $\Omega = [0,1]^d$ and divide it into subdomains

$$\Omega_i = \{(x_0, \ldots, x_{d-1}) \,|\, L_i < x_0 < L_{i+1}\}$$

for given subdomain boundaries $x_0 = L_i$, $i = 0, \ldots, s$, $L_0 = 0$, $L_s = 1$. Note that the boundaries $x_0 = L_i$ are points in 1D, lines in 2D, and planes in 3D.

Let us, on the command line, specify the polynomial degree of Lagrange elements and the number of element divisions in the various space directions, as explained in detail in Section 1.3.4. This results in an object `mesh` representing the interval $[0,1]$ in 1D, the unit square in 2D, or the unit cube in 3D.

Specification of subdomains (and boundary parts, if desired) is done using a user-defined subclass of `SubDomain`, as explained in Section 1.6.3. We could, in principle, introduce one subclass of `SubDomain` for each subdomain, and this would be feasible if one has a small and fixed number of subdomains as in the example in Section 1.6.3 with two subdomains. Our present case is more general as we have s subdomains. It then makes sense to create one subclass `Material` of `SubDomain` and have an attribute to reflect the subdomain (material) number. We use this number in the test whether a spatial point `x` is inside a subdomain or not:

```
class Material(SubDomain):
    """Define material (subdomain) no. i."""
    def __init__(self, subdomain_number, subdomain_boundaries):
        self.number = subdomain_number
        self.boundaries = subdomain_boundaries
        SubDomain.__init__(self)

    def inside(self, x, on_boundary):
        i = self.number
        L = self.boundaries          # short form (cf. the math)
        if L[i] <= x[0] <= L[i+1]:
            return True
        else:
            return False
```

The `<=` in the test if a point is inside a subdomain is important as `x` will equal vertex coordinates in the cells, and all vertices of a cell must lead to a `True` return value from the `inside` method for the cell to be a part of the actual subdomain. That is, the more mathematically natural test `L[i] <= x[0] < L[i+1]` fails to include elements with $x = L_{i+1}$ as boundary in subdomain Ω_i.

The marking and numbering of all subdomains goes as follows:

```
cell_entity_dim = mesh.topology().dim()  # = d
subdomains = MeshFunction('uint', mesh, cell_entity_dim)
# Mark subdomains with numbers i=0,1,\ldots,s (=len(L)-1)
for i in range(s):
    material_i = Material(i, L)
```

```
    material_i.mark(subdomains, i)
```

We have now all the geometric information about subdomains in a `MeshFunction` object `subdomains`. The subdomain number of mesh entity number `e`, here cell `e`, is given by `subdomains.array()[e]`.

The code presented so far had the purpose of preparing a mesh and a mesh function defining the subdomain. It is smart to put this code in a separate file, say `define_layers.py`, and view the code as a preprocessing step. We must then store the computed mesh and mesh function in files. Another program may load the files and perform the actually solve the boundary-value problem.

Storing the mesh itself and the mesh function in XML format is done by

```
file = File('hypercube_mesh.xml.gz')
file << mesh
file = File('layers.xml.gz')
file << subdomains
```

This preprocessing code knows about the layer geometries and the corresponding k, which must be propagated to the solver code. One idea is to let the preprocessing code write a Python module containing the `L` and `k` lists as well as an implementation of a function that evaluates the exact solution. The solver code can import this module to get access to `L`, `k`, and the exact solution (for comparison). The relevant Python code for generating a Python module may take the form

```
f = open('u_layered.py', 'w')
f.write("""
import numpy
L = numpy.array(
#s, float)
k = numpy.array(
#s, float)
s = len(L)-1

def u_e(x):
    # First find which subdomain x0 is located in
    for i in range(len(L)-1):
        if L[i] <= x <= L[i+1]:
            break

    # Vectorized implementation of summation:
    s2 = sum((L[1:s+1] - L[0:s])*(1.0/k[:]))
    if i == 0:
        u = (x - L[i])*(1.0/k[0])/s2
    else:
        s1 = sum((L[1:i+1] - L[0:i])*(1.0/k[0:i]))
        u = ((x - L[i])*(1.0/k[i]) + s1)/s2
    return u
```

```
if __name__ == '__main__':
    # Plot the exact solution
    from scitools.std import linspace, plot, array
    x = linspace(0, 1, 101)
    u = array([u_e(xi) for xi in x])
    print(u)
    plot(x, u)
"""
# (L, k))
f.close()
```

2.4.3 Solving the PDE problem

The solver program starts with loading a prepared mesh with a mesh function representing the subdomains:

```
mesh = Mesh('hypercube_mesh.xml.gz')
subdomains = MeshFunction('uint', mesh, 'layers.xml.gz')
```

The next task is to define the k function as a finite element function. As we recall from Section 1.6.3, a k that is constant in each element is suitable. We then follow the recipe from Section 1.6.3 to compute k:

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

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

The essential boundary conditions are defined in the same way is in `dn2_p2D.py` from Section 1.6.2 and therefore not repeated here. The variational problem is defined and solved in a standard manner,

```
u = TrialFunction(V)
v = TestFunction(V)
f = Constant(0)
a = k*inner(nabla_grad(u), nabla_grad(v))*dx
L = f*v*dx

problem = VariationalProblem(a, L, bc)
u = problem.solve()
```

Plotting the discontinuous `k` is often desired. Just a `plot(k)` makes a continuous function out of `k`, which is not what we want. Making a `MeshFunction` over cells and filling in the right k values results in an object that can be displayed as a discontinuous field. A relevant code is

```
k_meshfunc = MeshFunction('double', mesh, mesh.topology().dim())
```

```
# Scalar version
for i in range(len(subdomains.array())):
    k_meshfunc.array()[i] = k_values[subdomains.array()[i]]

# Vectorized version
help = numpy.asarray(subdomains.array(), dtype=numpy.int32)
k_meshfunc.array()[:] = numpy.choose(help, k_values)

plot(k_meshfunc, title='k as mesh function')
```

The file `Poisson_layers.py` contains the complete code.

2.5 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 Poission 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 Nedelec 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, the equations of hyperelasticity, fluid-structure interaction, viscous mantle flow, and the bidomain model of electrophysiology. Most of these solvers are described in the "FEniCS book" [15] (`https://launchpad.net/fenics-book`). Another project, cbc.rans (`https://launchpad.net/cbc.rans`), offers an environment for very flexible and easy implementation of Navier-Stokes solvers and turbulence [21, 20]. 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 im-

plementation of large systems of nonlinear PDEs in general. The functionality is found in the cbc.pdesys package (`https://launchpad.net/cbcpdesys`).

2.6 Miscellaneous topics

2.6.1 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 1.5.1).

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 = UnitIntervalMesh(10)` creates an instance of class `UnitIntervalMesh`, which is reached by the name `mesh`. (Class `UnitIntervalMesh` 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`

argument `self` (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`

2.6.2 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

Terminal

```
pydoc dolfin.X
```

to look up documentation of a Python class `X` from the DOLFIN library (`X` can be `UnitSquareMesh`, `Function`, `Viper`, etc.). Below is an overview of the most important classes and functions in FEniCS programs, in the order they typically appear within programs.

`UnitSquareMesh(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.

`UnitIntervalMesh`, `UnitCubeMesh`, `UnitCircleMesh`, `UnitSphere`, `IntervalMesh`, `RectangleMesh`, and `BoxMesh`: generate mesh over domains of simple geometric shape, see Section 2.3.

`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 `FunctionSpace` 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 1.2) 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 1.6.3) or for boundary conditions (see Section 1.6.4).

`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.

`TestFunction(V)`: define a test function on a space `V` to be used in a variational form.

`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.

`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 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`.

2.6.3 Linear solvers and preconditioners

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

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

Possible choices of preconditioners include

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

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

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

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

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

2.6.4 Using a backend-specific solver

Warning.

The linear algebra backends in FEniCS have recently changed. This section is not yet up-to-date.

The linear algebra backend determines the specific data structures that are used in the `Matrix` and `Vector` classes. For example, with the PETSc backend, `Matrix` encapsulates a PETSc matrix storage structure, and `Vector` encapsulates a PETSc vector storage structure. Sometimes one wants to perform operations directly on (say) the underlying PETSc objects. These can be fetched by

```
A_PETSc =
down_cast(A).mat() b_PETSc = down_cast(b).vec() U_PETSc =
down_cast(u.vector()).vec()
```

Here, `u` is a `Function`, `A` is a `Matrix`, and `b` is a `Vector`. The same syntax applies if we want to fetch the underlying Epetra, uBLAS, or MTL4 matrices and vectors.

Sometimes one wants to implement tailored solution algorithms, using special features of the underlying numerical packages. Here is an example where we create an ML preconditioned Conjugate Gradient solver by programming with Trilinos-specific objects directly. Given a linear system $AU = b$, represented by a `Matrix` object `A`, and two `Vector` objects `U` and `b` in a Python program, the purpose is to set up a solver using the Aztec Conjugate Gradient method from Trilinos' Aztec library and combine that solver with the algebraic multigrid preconditioner ML from the ML library in Trilinos. Since the various parts of Trilinos are mirrored in Python through the PyTrilinos package, we can operate directly on Trilinos-specific objects.

```
try:
    from PyTrilinos import Epetra, AztecOO, TriUtils, ML
except:
    print('''You Need to have PyTrilinos with
Epetra, AztecOO, TriUtils and ML installed
for this demo to run''')
    exit()

from dolfin import *

if not has_la_backend('Epetra'):
    print('Warning: Dolfin is not compiled with Trilinos')
    exit()

parameters['linear_algebra_backend'] = 'Epetra'

# create matrix A and vector b in the usual way
# u is a Function

# Fetch underlying Epetra objects
A_epetra = down_cast(A).mat()
b_epetra = down_cast(b).vec()
U_epetra = down_cast(u.vector()).vec()

# Sets up the parameters for ML using a python dictionary
ML_param = {"max levels"        : 3,
            "output"            : 10,
            "smoother: type"    : "ML symmetric Gauss-Seidel",
            "aggregation: type" : "Uncoupled",
            "ML validate parameter list" : False
}

# Create the preconditioner
prec = ML.MultiLevelPreconditioner(A_epetra, False)
prec.SetParameterList(ML_param)
prec.ComputePreconditioner()

# Create solver and solve system
```

```
solver = AztecOO.AztecOO(A_epetra, U_epetra, b_epetra)
solver.SetPrecOperator(prec)
solver.SetAztecOption(AztecOO.AZ_solver, AztecOO.AZ_cg)
solver.SetAztecOption(AztecOO.AZ_output, 16)
solver.Iterate(MaxIters=1550, Tolerance=1e-5)

plot(u)
```

2.6.5 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, I strongly recommend to use Ubuntu Linux. (Even though Mac users now can get FEniCS by a one-click install, I recommend using Ubuntu on Mac, unless you have significant experience with compiling and linking C++ libraries on Mac OS X.) 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.

On Windows you can use the tool Wubi[3] to automatically install Ubuntu in a dual boot fashion. A very popular alternative is to run Ubuntu in a separate window in your existing operation system, using a *virtual machine*. There are several virtual machine solutions to chose among, e.g., the free VirtualBoxMesh[4] or the commercial tool VMWare Fusion[5]. VirtualBoxMesh works well for many, but there might be hardware integration problems on Mac, so the superior VMWare Fusion tool is often worth the investment. The author has a description of how to install Ubuntu in a VMWare Fusion virtual machine[6].

Once Ubuntu is up and running, FEniCS is painlessly installed by

Terminal

```
sudo apt-get install fenics
```

Sometimes the FEniCS software in a standard Ubuntu installation lacks some recent features and bug fixes. Go to fenicsproject.org[7], click on *Download* and then the Ubuntu logo, move down to *Ubuntu PPA* and copy a few Unix commands to install the newest version of the FEniCS software.

A different type of virtual machine technology is Vagrant[8], which allows you to download a big file with a complete Ubuntu environment and run that environment in a terminal window on your Mac or Windows computer. This

[3] `http://www.ubuntu.com/download/desktop/windows-installer`
[4] `https://www.virtualbox.org/`
[5] `http://www.vmware.com/products/fusion/`
[6] `http://hplgit.github.io/teamods/ubuntu/vmware/index.html`
[7] `http://fenicsproject.org`
[8] `http://www.vagrantup.com/`

Ubuntu machine is integrated with the file system on your computer. The author has made a Vagrant box with most of the scientific computing software you need for programming with FEniCS, see a preliminary guide[9] for download, installation, and usage.

The FEniCS installation also features a set of demo programs. These are stored in locations depending on the type of operating system. For Ubuntu the programs are stored in `/usr/share/dolfin/demo`.

The graphical user interface (GUI) of Ubuntu is quite similar to both Windows and Mac OS X, but to be efficient when doing science with FEniCS I recommend to run programs in a terminal window and write them in a text editor like Emacs or Vim. You can employ an integrated development environment such as Eclipse, but intensive FEniCS developers and users tend to find terminal windows and plain text editors more efficient and user friendly.

2.6.6 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. The author has in development a book[10] that explains all details of the finite element method with the abstract mathematical formulations that FEniCS employ.

An easy-to-read book, which provides a good general background for using FEniCS, is Gockenbach [9]. The book by Donea and Huerta [6] has a similar style, but aims at readers with interest in fluid flow problems. Hughes [11] 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 [2] 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.* [7] is a very good choice. On the other hand, FEniCS users with a strong background in mathematics and interest in the mathematical properties of the finite element method, will appreciate the texts by Brenner and Scott [4], Braess [3], Ern and Guermond [8], Quarteroni and Valli [22], or Ciarlet [5].

2.6.7 Books on Python

Two very popular introductory books on Python are "Learning Python" by Lutz [17] and "Practical Python" by Hetland [10]. More advanced and comprehensive books include "Programming Python" by Lutz [16], and "Python Cookbook" [19] and "Python in a Nutshell" [18] by Martelli. The web page `http://wiki.python.org/moin/PythonBooks` lists numerous additional books.

[9] `http://hplgit.github.io/INF5620/doc/web/vagrant_inf5620.html`
[10] `http://tinyurl.com/opdfafk/`

Very few texts teach Python in a mathematical and numerical context, but the references [13, 14, 12] are exceptions.

Chapter 3

Troubleshooting

3.1 Compilation Problems

Expressions and variational forms in a FEniCS program need to be compiled to C++ and linked with libraries if the expressions or forms have been modified since last time they were compiled. The tool Instant, which is part of the FEniCS software suite, is used for compiling and linking C++ code so that it can be used with Python.

Sometimes the compilation fails. You can see from the series of error messages which statement in the Python program that led to a compilation problem. Make sure to scroll back and identify whether the problematic line is associated with an expression, variational form, or the solve step.

The final line in the output of error messages points to a log file from the compilation where one can examine the error messages from the compiler. It is usually the last lines of this log file that are of interest. Occasionally, the compiler's message can quickly lead to an understanding of the problem. A more fruitful approach is normally to examine the below list of common compilation problems and their remedies.

3.1.1 Problems with the Instant cache

Instant remembers information about previous compilations and versions of your program. Sometimes removal of this information can solve the problem. Just run

Terminal

```
instant-clean
```

in a terminal window whenever you encounter a compilation problem.

3.1.2 Syntax errors in expressions

If the compilation problem arises from line with an `Expression` object, examine the syntax of the expression carefully. Section 1.2.3 contains some information on valid syntax. You may also want to examine the log file, pointed to in the last line in the output of error messages. The compiler's message about the syntax problem may lead you to a solution.

Some common problems are

1. using `a**b` for exponentiation (illegal in C++) instead of `pow(a, b)`,
2. forgetting that the spatial coordinates are denoted by a vector `x`,
3. forgetting that the x, y, and z coordinates in space correspond to `x[0]`, `x[1]`, and `x[2]`, respectively.

Failure to initialize parameters in the expressions lead to a compilation error where this problem is explicitly pointed out.

Example. The implementation

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

fails with compilation error

```
RuntimeError: In instant.recompile: The module did not compile with
command 'make VERBOSE=1', see '/some/path/.../compile.log'
```

Looking up the `compile.log` file and searching for `error`, we see the following message from the C++ compiler:

```
error: expected ‘’) before ‘’p_0
   values[0] = x[1] <= 0.5? 2*x[1]*p_1/(p_0 + p_1) :
   ((2*x[1]-1)p_0 + p_1)/(p_0 + p_1);
                ^
```

Now we realize that a `*` symbol is missing between `)` and `p_0`.

3.1.3 Problems in the solve step

Sometimes the problem lies in the solve step where a variational form is turned into a system of algebraic equations. The error message *Unable to extract all indicies* points to a problem with the variational form. Common errors include

1. missing either the `TrialFunction` or the `TestFunction` object,
2. no terms without `TrialFunction` objects.

3. mathematically invalid operations in the variational form.

The first problem implies that one cannot make a matrix system or system of nonlinear algebraic equations out of the variational form. The second problem means that there is no "right-hand side" terms in the PDE with known quantities. Sometimes this is seemingly the case mathematically because the "right-hand side" is zero. Variational forms must represent this case as `Constant(0)*v*dx` where `v` is a `TestFunction` object. An example of the third problem is to take the `inner` product of a scalar and a vector (causing in this particular case the error message to be "Shape mismatch").

The message *Unable to extract common cell; missing cell definition in form or expression* will typically arise from a term in the form where a test function (holding mesh and cell information) is missing. For example, a zero right-hand side `Constant(0)*dx` will generate this error.

3.1.4 Unable to convert object to a UFL form

One common reason for the above error message is that a form is written without being multiplied by `dx` or `ds`.

3.1.5 UFL reports that a numpy array cannot be converted to any UFL type

One reason may be that there are mathematical functions like `sin` and `exp` operating on `numpy` arrays. The problem is that the

```
from dolfin import *
```

statement imports `sin`, `cos`, and similar mathematical functions from UFL and these are aimed at taking `Function` or `TrialFunction` objects as arguments and not `numpy` arrays. The remedy is to use prefix mathematical functions aimed at `numpy` arrays with `numpy`, or `np` if `numpy` is imported as `np`: `numpy.exp` or `np.exp`, for instance. Normally, boundary conditions and analytical solutions are represented by `Expression` objects and then this problem does not arise. The problem usually arises when pure Python functions with, e.g., analytical solutions are introduced for, e.g., plotting.

3.1.6 All programs fail to compile

When encountering a compilation problem where the Instant log file says something about missing double quote in an `Expression`, try compiling a previously working program. If that program faces the same problem, reboot Ubuntu and try again. If the problem persists, try running the Update Manager (because unfinished updates can cause compiler problems), reboot and try again.

3.2 Problems with Expression Objects

3.2.1 There seems to be some bug in an Expression object

Run the command `instant-clean` to ensure that everything is (re)compiled. Check the formulas in string expressions carefully, and make sure that divisions do not lead to integer division (i.e., at least one of the operands in a division must be a floating-point variable).

3.2.2 Segmentation fault when using an Expression object

One reason may be that the point vector `x` has indices out of bounds, e.g., that you access `x[2]` but the mesh is only a 2D mesh. Also recall that the components of `x` are `x[0]`, `x[1]`, etc. Accessing `x[2]` as the "y" coordinate is a common error.

3.3 Other Problems

3.3.1 Very strange error message involving a `mesh` variable

If you encounter a really strange error message, and the statement in question involves a variable with name `mesh`, check if this is really your mesh variable. When doing `from dolfin import *`, you get a `mesh` variable, which is actually a module, and sending this module to functions creates a `TypeError`. Substitute with the actual name of your mesh object.

One should also note other names that get imported by `from dolfin import *`: `i`, `j`, `k`, `l`, `p`, `q`, `r`, `s`. It is easy to use such variables without initializing them, and strange error message arises (since the mentioned names are UFL `Index` objects).

3.3.2 The plot disapperas quickly from the screen

You have forgotten to insert `interactive()` as the last statement in the program.

3.3.3 Only parts of the program are executed

Check if a call to `interactive()` appears in the middle of the program. The computations are halted by this call and not continued before you press `q` in a plot window. Most people thus prefer to have `interactive()` as the last statement.

3.3.4 Error in the definition of the boundary

Consider this code and error message:

```
class DirichletBoundary(SubDomain):  # define the Dirichlet boundary
    def inside(self, x, on_boundary):
        return on_boundary and abs(x) < 1E-14

bc = DirichletBC(V, u0, xleft_boundary)

Error: ValueError: The truth value of an array with more than
one element is ambiguous. Use a.any() or a.all()
```

The reason for this error message is that `x` is a point vector, not just a number. In the `inside` function one must work with the components of `x`: `x[0]`, `x[1]`, etc.

3.3.5 The solver in a nonlinear problems does not converge

There can be many reasons for this common problem:

1. The form (variational formulation) is not consistent with the PDE(s).
2. The boundary conditions in a Newton method are wrong. The correction vector must have vanishing essential conditions where the complete solution has zero or non-zero values.
3. The initial guess for the solution is not appropriate. In some problems, a simple function equal to 0 just leads to a zero solution or a divergent solver. Try 1 as initial guess, or (better) try to identify a linear problem that can be used to compute an appropriate initial guess, see Section 2.2.2.

3.4 How To Debug a FEniCS Program?

Here is an action list you may follow.

Step 1. Examine the weak form and its implementation carefully. Check that all terms are multiplied by `dx` or `ds`, and that the terms do not vanish; check that at least one term has both a `TrialFunction` and a `TestFunction` (term with unknown); and check that at least one term has no `TrialFunction` (known term).

Step 2. Check that Dirichlet boundary conditions are set correctly.

```
# bcs is list of DirichletBC objects
for bc in bcs:
bc_dict = bc.get_boundary_values()
    for dof in bc_dict:
        print('dof %d: value=%s' % (dof, bc_dict[dof]))
```

See also an expanded version of this snippet in the `solvers_bc p2D_vc.py` file located in the directory `poisson`.

A next step in the debugging, if these values are wrong, is to call the functions that define the boundary parts. For example,

```
for coor in mesh.coordinates():
    if my_boundary_function(coor, True):
        print('%s is on the boundary' % coor)

# or, in case of a SubDomain subclass my_subdomain_object,
for coor in mesh.coordinates():
    if my_subdomain_object.inside(coor, True):
        print('%s is on the boundary' % coor)
```

You may map the solution to a structured grid with structured data, i.e., a `BoxField`, see Chapters 1.4.2 and 2.1.4, and then examine the solution field along grid lines in x and y directions. For example, you can easily check that correct Dirichlet conditions are set, e.g., at the upper boundary (check `u_box[:,-1]`).

Step 4. Switching to a simple set of coefficients and boundary conditions, such that the solution becomes simple too, but still obeys the same PDE, may help since it is then easier to examine numerical values in the solution array.

Step 5. Formulate a corresponding 1D problem. Often this can be done by just running the problem with a 1D mesh. Doing hand calculations of element matrices and vectors, and comparing the assembled system from these hand calculations with the assembled system from the FEniCS program can uncover bugs. For nonlinear problems, or problems with variable coefficients, it is usually wise to choose simple coefficients so that the problem becomes effectively linear and the hand calculations are doable.

Bibliography

[1] M. S. Alnæs, A. Logg, K. B. Ølgaard, M. E. Rognes, and G. N. Wells. Unified Form Language: A domain-specific language for weak formulations of partial differential equations. *ACM Transactions on Mathematical Software*, 40(2), 2014. doi:10.1145/2566630, arXiv:1211.4047.

[2] W. B. Bickford. *A First Course in the Finite Element Method*. Irwin, 2nd edition, 1994.

[3] Dietrich Braess. *Finite Elements*. Cambridge University Press, Cambridge, third edition, 2007.

[4] Susanne C. Brenner and L. Ridgway Scott. *The Mathematical Theory of Finite Element Methods*, volume 15 of *Texts in Applied Mathematics*. Springer, New York, third edition, 2008.

[5] Philippe G. Ciarlet. *The Finite Element Method for Elliptic Problems*, volume 40 of *Classics in Applied Mathematics*. SIAM, Philadelphia, PA, 2002. Reprint of the 1978 original [North-Holland, Amsterdam; MR0520174 (58 #25001)].

[6] J. Donea and A. Huerta. *Finite Element Methods for Flow Problems*. Wiley Press, 2003.

[7] K. Eriksson, D. Estep, P. Hansbo, and C. Johnson. *Computational Differential Equations*. Cambridge University Press, 1996.

[8] A. Ern and J.-L. Guermond. *Theory and Practice of Finite Elements*. Springer, 2004.

[9] M. Gockenbach. *Understanding and Implementing the Finite Element Method*. SIAM, 2006.

[10] M. L. Hetland. *Practical Python*. APress, 2002.

[11] T. J. R. Hughes. *The Finite Element Method: Linear Static and Dynamic Finite Element Analysis*. Prentice-Hall, 1987.

[12] J. Kiusalaas. *Numerical Methods in Engineering With Python*. Cambridge University Press, 2005.

[13] H. P. Langtangen. *Python Scripting for Computational Science*. Springer, third edition, 2009.

[14] H. P. Langtangen. *A Primer on Scientific Programming With Python*. Texts in Computational Science and Engineering, vol 6. Springer, second edition, 2011.

[15] A. Logg, K.-A. Mardal, and G. N. Wells. *Automated Solution of Partial Differential Equations by the Finite Element Method*. Springer, 2012.

[16] M. Lutz. *Programming Python*. O'Reilly, third edition, 2006.

[17] M. Lutz. *Learning Python*. O'Reilly, third edition, 2007.

[18] A. Martelli. *Python in a Nutshell*. O'Reilly, second edition, 2006.

[19] A. Martelli and D. Ascher. *Python Cookbook*. O'Reilly, second edition, 2005.

[20] M. Mortensen, H. P. Langtangen, and J. Myre. Cbc.rans - a new flexible, programmable software framework for computational fluid dynamics. In H. I. Andersson and B. Skallerud, editors, *Sixth National Conference on Computational Mechanics (MekIT'11)*. Tapir, 2011.

[21] M. Mortensen, H. P. Langtangen, and G. N. Wells. A FEniCS-based programming framework for modeling turbulent flow by the Reynolds-averaged Navier-Stokes equations. *Advances in Water Resources*, 2011. doi: 10.1016/j.advwatres.2011.02.013.

[22] A. Quarteroni and A. Valli. *Numerical Approximation of Partial Differential Equations*. Springer Series in Computational Mathematics. Springer, 1994.

[23] A. Henderson Squillacote. *The Paraview Guide*. Kitware, 2007.

Index