A FEniCS Tutorial (November 16, 2009)

H. P. Langtangen[1,2]

[1] Center for Biomedical Computing, Simula Research Laboratory
[2] Department of Informatics, University of Oslo

This document is in a preliminary state. Please send errors, typos and suggestions for improvements to the author at hpl@simula.no.

Table of Contents

1 The Fundamentals

FEniCS is a user-friendly tool for solving partial differential equations (PDEs). The purpose of this tutorial is 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 treat 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 enhances the verification of the impelementations. 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. Chapter 8.3 explains briefly how to install the necessary tools.

1.1 The Poisson Equation

Computer programming books frequently start with an example on how to print "Hello, World!" on the screen. The counterpart to the "Hello, World!" example in the world of software for partial differential equations is a program which solves the Poisson problem,

$$\begin{aligned} -\Delta u &= f \quad \text{in } \Omega, \\ u &= u_0 \text{ on } \partial\Omega\,. \end{aligned} \tag{1}$$

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

In two space dimensions with coordinates x and y, we can write out the Poisson equation (1) in detail:

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

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

The Poisson equation (1) arises in numerical physical contexts, for example, heat conduction, electrostatics, diffusion of substances, twisting of elastic rods, inviscid fluid flow, water waves. Moreover, the equation appears in numerical splitting strategies of more complicated systems of PDEs, in particular the Navier-Stokes equations.

1.2 Variational Formulation

FEniCS makes it easy to solve PDEs if finite elements are used for discretization in space and the problem is expressed as a *variational problem*. Readers who are not familiar with variational problems will get a brief introduction to the topic in this tutorial, and in Chapter **??**, but we encourage getting and reading a proper book on the finite element method in addition. Chapter 8.4 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 by the test function v and integrate,

$$-\int_\Omega (\Delta u)v \,\mathrm{d}x = \int_\Omega fv \,\mathrm{d}x \,. \tag{3}$$

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

$$-\int_\Omega (\Delta u)v \,\mathrm{d}x = \int_\Omega \nabla u\cdot\nabla v \,\mathrm{d}x - \int_{\partial\Omega} \frac{\partial u}{\partial n} v \,\mathrm{d}s. \tag{4}$$

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 (4) therefore vanishes. From (3) and (4) it follows that

$$\int_\Omega \nabla u\cdot\nabla v \,\mathrm{d}x = \int_\Omega fv \,\mathrm{d}x \,. \tag{5}$$

This equation is supposed to hold for all v in some function space $\hat{V}$. The trial function u lies in some (possible other) function space V. We refer to (5) as the *weak form* of the original boundary-value problem (1).

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

$$\int_\Omega \nabla u\cdot\nabla v \,\mathrm{d}x = \int_\Omega fv \,\mathrm{d}x \quad \forall v \in \hat{V}. \tag{6}$$

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

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

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 requirements of u in the variational statement (6), caused by the integration by parts, have great practical consequences when it comes to constructing finite elements.

To solve the Poisson equation numerically, we need to transform the continuous variational problem (6) to a discrete variational problem. This is done by introducing *finite-dimensional* test and trial spaces $\hat{V}_h \subset \hat{V}$ and $V_h \subset V$. The discrete variational problem reads: Find $u_h \in V_h \subset V$ such that

$$\int_\Omega \nabla u_h \cdot \nabla v \, \mathrm{d}x = \int_\Omega f v \, \mathrm{d}x \quad \forall v \in \hat{V}_h \subset \hat{V} \, . \tag{7}$$

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.

The mathematics literature on variational problems applies u_h for the solution of the discrete problem and u for the solution of the continous problem. To obtain (almost) a one-to-one relationshop 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 and then simply let u denote the finite element solution when we come to the discrete variational problem and the associated program development.

It turns out to be convenient to introduce a unified notation for a weak form like (7):

$$a(u, v) = L(v) \, . \tag{8}$$

In the present problem we have that

$$a(u, v) = \int_\Omega \nabla u \cdot \nabla v \, \mathrm{d}x, \tag{9}$$

$$L(v) = \int_\Omega f v \, \mathrm{d}x \, . \tag{10}$$

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

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

1. Turn the PDE problem into a discrete variational problem: Find $u \in V_h$ such that
$$a(u, v) = L(v) \quad \forall v \in \hat{V}_h \, .$$
2. Specify the choice of discrete spaces, i.e., choice of finite elements.

1.3 The Implementation

The test problem so far has a general domain Ω and general functions u_0 and f. However, we must specify Ω, u_0, and f prior to our first implementation. It will be wise to construct a specific problem where we can easily check that the solution is correct. Let us choose $u(x, y) = 1 + x^2 + 2y^2$ to be the solution of our Poisson problem since the finite element method with linear elements over a uniform mesh of triangular cells should 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 code by using very few elements and checking that the computed and the exact solution equal to machine precision. Test problems with this property will be frequently constructed throughout the present tutorial.

Specifying $u(x, y) = 1+x^2+2y^2$ in the problem from Chapter 1.2 implies $u_0(x, y) = 1 + x^2 + 2y^2$ and $f(x, y) = -6$. We let Ω be the unit square for simplicity. A FEniCS program for solving (1) with the given choices of u_0, f, and Ω may look as follows (the complete code can be found in the file `Poisson2D_D1.py`):

```
from dolfin import *

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

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

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

u0_boundary = Boundary()
bc = DirichletBC(V, u0, u0_boundary)

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

# Compute solution
problem = VariationalProblem(a, L, bc)
u = problem.solve()

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

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

```
# Hold plot
interactive()
```

We shall now dissect this FEniCS program in detail. The program is written in the Python programming language. You may either take a quick look at a Python tutorial [17] 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 this tutorial. The latter strategy has proven to work for many newcomers to FEniCS[1]. Chapter 8.5 lists some good Python books.

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

The first line in the program,

```
from dolfin import *
```

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

The statement

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

defines a uniform finite element mesh over the unit square $[0, 1] \times [0, 1]$. The mesh consists of *cells*, which are triangles with straight sides. The parameters 6 and 4 tell that the square is first divided into $6 \cdot 4$ 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*

[1] The requirement of using Python and an abstract mathematical formulation of the finite element problem may seem difficult for 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 quited limited. And Python is an easy-to-learn language that you certainly will love and use far beyond FEniCS programming.

program to create the mesh. The FEniCS program will then read the mesh from file.

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

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

The second argument reflects the type of element, while the third argument is the degree of the basis functions on the element. Here, `'CG'` stands for Continuous Galerkin, implying the standard Lagrange family of elements. Insted of `'CG'` we could have written `'Lagrange'`. With degree 1, we simply get the standard linear Lagrange element, which is a triangle with nodes at the three vertices. Some finite element practitioners refer to this element as the "linear triangle". The computed u will be continuous and linearly varying in x and y over each cell in the mesh. Higher-order polynomial approximations over each cell are trivially obtained by increasing the third parameter to `FunctionSpace`.

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

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

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 an instance describing if a point lies on the boundary where u is specified. The term *instance* means a Python object of a particular type (such as `Function`, `FunctionSpace`, etc.). Many use *instance* and *object* as interchangable terms. In other computer programming languages one may also use the term *variable* for the same thing. We shall in this tutorial mostly use the term *instance*, since that is most common in a Python context, but *object* will also be occasionally used where that is more natural.

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` instance, which is used to represent a mathematical function. The typical construction is

```
u0 = Expression(formula, V=V)
```

where `V` is a `FunctionSpace`[2] and `formula` is a string containing the mathematical expression. This formula written with C++ syntax (the expression is automatically turned into an efficient, compiled C++ function, see Chapter 8.6 for details on the syntax). The independent variables in the function expression are supposed to be available as a point vector `x`, where the first element `x[0]` corresponds to the x coordinate, the second element `x[1]` to the y coordinate, and (in a three-dimensional problem) `x[2]` to the z coordinate. With our choice of $u_0(x, y) = 1 + x^2 + 2y^2$, the formula string must be written as `1 + x[0]*x[0] + 2*x[1]*x[1]`:

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

The information about where to apply the `u0` function as boundary condition is coded in a method `inside` in a subclass of class `SubDomain`[3]:

```
class Boundary(SubDomain):
    def inside(self, x, on_boundary):
        return on_boundary

on_boundary = Boundary()
```

The method `inside` shall return a boolean value: `True` if the 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 we can just return `on_boundary`. In later examples we will demonstrate how to set Dirichlet conditions on parts of the boundary, typically achieved by some test on the `x` values inside the `inside` method (as for the formula in `Expression` instances, `x` in the `inside` method represents a point in space with coordinates `x[0]`, `x[1]`, etc.). The `inside` method is called for every discrete point in the mesh, which allows us to have boundaries where u are known also inside the domain, if desired. The choice of class name, here `Boundary`, is up to the programmer, but the class must be derived from `SubDomain` and it must have an `inside` method.

Newcomers to Python class programming often face some problems with understanding the `self` parameter in the `inside` function. For now it suffices to know that `self` is a required first argument when defining a function in a class. There is no need to understand the `self` argument before in Chapter 7.2.

[2] This does not imply that the formula is turned into a finite element function in that space. The space is just occasionally needed, usually in special cases in variational forms for providing information about elements and a mesh in an integration.

[3] If you are unfamiliar with classes and class methods in Python, stay cool and just modify the many examples on boundary specifications found in this tutorial. It may well suffice to pick up Python class programming at a later stage.

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

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

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

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

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

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

In essence, these two lines specify the PDE to be solved. Note the very close correspondence between the Python syntax and the mathematical formulas (9)–(10)! This is a key strength of FEniCS: the formulas in the variational formulation translate directly to very similar Python code, a feature that makes it easy to specify PDE problems with lots of PDEs and complicated terms in the equations. The language used to express weak forms is called UFL (Unified Form Language) and is an integral part of FEniCS.

Having `a` and `L` defined, and information about essential (Dirichlet) boundary conditions in `bc`, we can formulate a `VariationalProblem`:

```
problem = VariationalProblem(a, L, bc)
```

Solving the variational problem for the solution `u` is just a matter of writing

```
u = problem.solve()
```

Unless otherwise stated, a sparse direct solver is used to solve the underlying linear system implied by the variational formulation. The type of sparse direct solver depends on which linear algebra package that is used by default. If DOLFIN is compiled with PETSc, that package is the default linear algebra backend, otherwise it is uBLAS. The FEniCS distribution for Ubuntu Linux contains PETSc, and then the default solver becomes the sparse LU solver from UMFPACK (which PETSc has an interface to). We shall later in Chapter 4 demonstrate how to get full control of the choice of solver and any solver parameters.

The `u` variable refers to a finite element function, called simply a `Function` in FEniCS terminology. Note that we first defined `u` as a `TrialFunction` and used it to specify `a`. Thereafter, we redefined `u` to be a `Function` representing the computed solution. This redefinition of the variable `u` is possible in Python and a programming practice in FEniCS applications.

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

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

The `interactive()` call is necessary for the plot to remain on the screen. With the left, middle, and right mouse buttons you can rotate, translate, and zoom (respectively) the plotted surface to better examine how the solution looks like.

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 from Viper is intended for quick examination of the solution during program development. More in-depth visual investigations of finite element solution will normally benefit from using highly professional tools such as ParaView and VisIt.

1.4 Examining the Discrete Solution

We know that, in the particular boundary-value problem of Chapter 1.3, the computed solution u should equal the exact solution at the vertices of the cells. An important extension of our first program is therefore to examine the computed values of the solution, which is the focus of the present section.

A finite element function like u is expressed as a linear combination of basis functions ϕ_i (spanning the space V_h):

$$\sum_{j=1}^{N} U_j\phi_j \,. \tag{11}$$

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

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

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

The result is a DOLFIN `Vector` instance, which is basically an encapsulation of the vector object used in the linear algebra package that is applied to solve the linear system arising form the variational problem. Since we program in Python it is convenient to convert the `Vector` instance 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`.

The coordinates of the vertices in the mesh can be extracted by

```
coor = mesh.coordinates()
```

For a d-dimensional problem, `coor` is an $M \times d$ `numpy` array, M being the number of vertices in the mesh. Writing out the solution on the screen can now be done by a simple loop:

```
for i in range(len(u_array)):
    print 'u(%8g,%8g) = %g' % \
          (coor[i][0], coor[i][1], u_array[i])
```

The beginning of the output looks like

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

For Lagrange elements of degree higher than one, the vertices and the nodes do not coincide, and then the loop above is meaningless.

For verification purposes we want to compare the values of `u` at the nodes, i.e., the values of the vector `u_array`, with the exact solution given by `u0`. At each node, the difference between the computed and exact solution should be less than a small tolerance. The exact solution is given by the `Expression` instance `u0`, which we can evaluate directly as `u0(coor[i])` at the vertex with global number `i`, or as `u0(x)` for any spatial point. Alternatively, we can make a finite element field `u_e`, representing the exact solution, whose values at the nodes are given by the `u0` function. With mathematics, $u_e = \sum_{j=1}^{N} E_j\phi_j$, where $E_j = u_0(x_j, y_j)$, (x_j, y_j) being the coordinates of node no. j. This process is known as interpolation. FEniCS has a function for performing the operation:

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

The maximum error can now be computed as

```
u_e_array = u_e.vector().array()
diff = abs(u_array - u_e_array)
print 'Max error:', diff.max()

# or more compactly:
print 'Max error:', abs(u_e_array - u_array).max()
```

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

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

```
# Compare numerical and exact solution at (0.5, 0.5)
center = (0.5, 0.5)
u_value = u(center)
u0_value = u0(center)
print 'numerical u at the center point:', u_value
print 'exact     u at the center point:', u0_value
```

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

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

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

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

- `mesh.num_cells()` returns the number of cells (triangles) in the mesh,
- `mesh.num_vertices()` returns the number of vertices in the mesh (with our choice of linear Lagrange elements this equals the number of nodes),
- `str(mesh)` returns a short "pretty print" description of the mesh, e.g.,
  ```
  <Mesh of topological dimension 2 (triangles) with
  16 vertices and 18 cells, ordered>
  ```
  and `print mesh` is actually the same as `print str(mesh)`.

All mesh objects are of type `Mesh` so typing the command `pydoc dolfin.Mesh` in a terminal window will give a list of methods that can be called through any `Mesh` instance. In fact, `pydoc dolfin.X` shows the documentation of any DOLFIN name `X` (at the time of this writing, some names have missing or incomplete documentation).

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 snippet performs the task:

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

That is, we manipulate `u_array` as desired, and then we insert this array into `u`'s `Vector` instance. 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`. We can equally well insert the entries of `u_array` into `u`'s `numpy` array:

```
u.vector().array()[:] = u_array
```

All the code in this subsection can be found in the file `Poisson2D_D2.py`.

1.5 Formulating a Real Physical Problem

Perhaps you are not particularly amazed by viewing the simple surface of u in the test problem from Chapters 1.3 and 1.4. However, solving a real physical problem with a more interesting and amazing solution on the screen is only a matter of specifying a more exciting domain, boundary condition, and/or right-hand side f.

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

$$-T\Delta D = p(x, y) \quad \text{in } \Omega = \{(x, y) \,|\, x^2 + y^2 \le R\}, \tag{12}$$

with

$$p(x, y) = \frac{A}{2\pi\sigma} \exp\left(-\frac{1}{2}\left(\frac{x - x_0}{\sigma}\right)^2 - \frac{1}{2}\left(\frac{y - y_0}{\sigma}\right)^2\right). \tag{13}$$

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 condition is $D = 0$.

Introducing a scaling with R as characteristic length and $8\pi\sigma T/A$ as characteristic size of D, we can derive the equivalent scaled problem on the unit circle,

$$-\Delta w = 4 \exp\left(-\frac{1}{2}\left(\frac{Rx - x_0}{\sigma}\right)^2 - \frac{1}{2}\left(\frac{Ry - y_0}{\sigma}\right)^2\right), \tag{14}$$

with $w = 0$ on the boundary. We have that $D = Aw/(8\pi\sigma T)$.

A mesh over the unit circle can be created by

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

where `n` is the typical number of elements in the radial direction. You should now be able to figure out how to modify the `Poisson2D_D1.py` code to solve this membrane problem. More specifically, you are recommended to perform the following extensions:

1. initialize R, x_0, y_0, σ, T, and A in the beginning of the program,
2. build a string expression for p with correct C++ syntax (use printf formatting in Python to build the expression),
3. define the `a` and `L` variables in the variational problem for w and compute the solution,
4. plot the mesh, w, and the scaled pressure function p (the right-hand side of (14)),
5. write out the maximum real deflection D (i.e., the maximum of the w values times $A/(8\pi\sigma T)$).

Use variable names in the program similar to the mathematical symbols in this problem.

Choosing a small width σ (say 0.01) and a location (x_9, y_0) toward the circular boundary (say $(0.6R\cos\theta, 0.6R\sin\theta)$ for any $\theta \in [0, 2\pi]$), may produce an exciting visual comparison of w and p that demonstrates the very smoothed elastic response to a peak force (or mathematically, the smoothing properties of the Laplace operator). You need to experiment with the mesh resolution to get a smooth visual representation of p.

In the limit $\sigma \to \infty$, the right-hand side p of (14) approaches the constant 4, and then the solution should be $w(x, y) = 1 - x^2 - y^2$. Compute the absolute value of the difference between the exact and the numerical solution if $\sigma \geq 50$ and write out the maximum difference to provide some evidence that the implementation is correct.

You are strongly encouraged to spend some time on doing this exercise and play around with the plots and different mesh resolutions. A suggested solution to the exercise can be found in the file `membrane1.py`.

```
from dolfin import *

# Set pressure function:
T = 10.0  # tension
A = 1.0   # pressure amplitude
R = 0.3   # radius of domain
theta = 0.2
x0 = 0.6*R*cos(theta)
y0 = 0.6*R*sin(theta)
sigma = 0.025
#sigma = 50  # verification
pressure = '4*exp(-0.5*(pow((%g*x[0] - %g)/%g, 2)) '\
           '     - 0.5*(pow((%g*x[1] - %g)/%g, 2)))' % \
           (R, x0, sigma, R, y0, sigma)

n = 40   # approx no of elements in radial direction
mesh = UnitCircle(n)
V = FunctionSpace(mesh, 'CG', 1)
```

```
# Define boundary condition w=0

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

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

# Define variational problem
v = TestFunction(V)
w = TrialFunction(V)
p = Expression(pressure, V=V)
a = dot(grad(w), grad(v))*dx
L = v*p*dx

# Compute solution
problem = VariationalProblem(a, L, bc)
w = problem.solve()

# Plot solution and mesh
plot(mesh, title='Mesh over scaled domain')
plot(w, title='Scaled deflection')
plot(p, title='Scaled pressure')

# Find maximum real deflection
max_w = w.vector().array().max()
max_D = A*max_w/(8*pi*sigma*T)
print 'Maximum real deflection is', max_D

# Verification for "flat" pressure (big sigma)
if sigma >= 50:
    w_exact = Expression('1 - x[0]*x[0] - x[1]*x[1]', V=V)
    w_e = interpolate(w_exact, V)
    w_e_array = w_e.vector().array()
    w_array = w.vector().array()
    diff_array = abs(w_e_array - w_array)
    print 'Verification of the solution, max difference is %.4E' % \
          diff_array.max()

    # Create finite element field over V and fill with error values
    difference = Function(V)
    difference.vector()[:] = diff_array
    #plot(difference, title='Error field for sigma=%g' % sigma)

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

1.6 Computing Derivatives

In many Poisson and 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, `grad(u)` denotes the gradient. However, the gradient of a 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 $w = \nabla u$ by a finite element method[4], using the the same elements for the components of w as we used for u.

The variational problem for w reads: Find $w \in V_h$ such that

$$a(w, v) = L(v) \quad \forall v \in \hat{V}_h^{(g)}, \tag{15}$$

where

$$a(w, v) = \int_\Omega w \cdot v \, \mathrm{d}x, \tag{16}$$

$$L(v) = \int_\Omega \nabla u \cdot v \, \mathrm{d}x \,. \tag{17}$$

The function spaces V_h and $\hat{V}_h^{(g)}$ are vector versions of the function space for u, with boundary conditions removed (if V_h is the space we used for u, with no restrictions on boundary values, $V_h^{(g)} = \hat{V}_h^{(g)} = [V_h]^d$, 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 `gradu` in the code, is easy to solve in FEniCS:

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

a = dot(w, v)*dx
L = dot(grad(u), v)*dx
problem = VariationalProblem(a, L)
gradu = problem.solve()

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

[4] This process is known as *projection*. Looking at the component $\partial u/\partial x$ of the gradient, we project the (discrete) derivative $\sum_j U_j \partial\phi_j/\partial x$ onto another function space with basis $\bar{\phi}_1, \ldots \bar{\phi}$ such that the derivative in this space is expressed by the standard sum $\sum_j \bar{U}_j \bar{\phi}_j$, for suitable (new) coefficients $\bar{U}_j$.

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.

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

```
gradu_x, gradu_y = gradu.split(deepcopy=True)  # extract components
plot(gradu_x, title='x-component of grad(u)')
plot(gradu_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 `gradu_x` and `gradu_y` variables behave as `Function` instances. In particular, we can extract the underlying arrays of nodal values by

```
gradu_x_array = gradu_x.vector().array()
gradu_y_array = gradu_y.vector().array()
```

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

```
gradu_array = gradu.vector().array()
```

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

The program `Poisson2D_D3.py` extends the code `Poisson2D_D2.py` from Chapter 1.4 with computations and visualizations of the gradient. Examining the arrays `gradu_x_array` and `gradu_y_array`, or looking at the plots of `gradu_x` and `gradu_y`, quickly reveals that the computed `gradu` field does not equal the exact gradient $(2x, 4y)$ in this particular test problem where $u = 1 + x^2 + y^2$. 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). See Chapter 1.8 for illustrations of this phenomenon.

Representing the gradient by the same elements as we used for the solution is a very common step in finite element programs, so the formation and solution of a variational problem for w as shown above can be replaced by a one-line call:

```
gradu = project(grad(u), VectorFunctionSpace(mesh, 'CG', 1))
```

The `project` function can take an expression involving some finite element function in some space and project the expression onto another space. The applications 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 first-order field which is required by many visualization packages.

1.7 Computing Functionals

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

$$\frac{1}{2}||\nabla u||^2 \equiv \frac{1}{2}\int_\Omega \nabla u\cdot\nabla u\,\mathrm{d}x, \tag{18}$$

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

$$||u-u_e|| = \left(\int_\Omega (u_e-u)^2\,\mathrm{d}x\right)^{1/2}, \tag{19}$$

which is of particular interest when studying convergence properties. Sometimes the interst concerns the flux out of a part Γ of the boundary $\partial\Omega$,

$$F = -\int_\Gamma p\nabla u\cdot \boldsymbol{n}\,\mathrm{d}s, \tag{20}$$

where $\boldsymbol{n}$ is an outward unit normal at Γ and p is a coefficient (see the problem in Chapter 1.12 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 (18) is described in the UFL language in the same manner as we describe weak forms:

```
energy = 0.5*dot(grad(u), grad(u))*dx
E = assemble(energy, mesh=mesh)
```

The `assemble` call performs the integration. It is possible to restrict the integration to subdomains, using a mesh function to mark the subdomains as explained in Chapter 6.3. The program `membrane2.py` carries out the computation of the elastic energy $\frac{1}{2}||T\nabla w||^2$ in the membrane problem from Chapter 1.5.

Convergence Estimation. To illustrate error computations and convergence of finite element solutions, we modify the `Poisson2D_D3.py` program from Chapter 1.6 and specify a more complicated solution,

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

on the unit square. It follows that $u_0 = 0$ and that $f(x,y) = 2\omega^2\pi^2 u(x,y)$. We must define the appropriate boundary conditions, the exact solution, and the f function:

```
class Boundary(SubDomain):
    def inside(self, x, on_boundary):
        return on_boundary

bc = DirichletBC(V, Constant(mesh, 0.0), Boundary())

omega = 1.0
u_exact = Expression('sin(%g*pi*x[0])*sin(%g*pi*x[1])' % \
                     (omega, omega), V=V)

f = 2*pi**2*omega**2*u_exact
```

The computation of (19) can be done by

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

However, `u_exact` will here be interpolated onto the function space `V`, i.e., 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 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, 'CG', degree=3)
u_e = interpolate(u_exact, Ve)
error = (u - u_e)**2*dx
E = sqrt(assemble(error))
```

The `u` function will here be automatically interpolated and represented in the `Ve` space. When functions in different function spaces enter UFL expressions, they will be represented in the space of highest order before integrations are carried out. When in doubt, we should explicitly interpolate `u`:

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

The square in the expression for `error` will be expanded and lead to a lot of terms that almost cancel when the error is small, with the potential of introducing significant round-off errors. The function `errornorm` is available for avoiding this effect by first interpolating `u` and `u_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)
```

A the time of this writing, `errornorm` does not work with `Expression` instances for `u_exact`, making the function inapplicable for most practical purposes. Nevertheless, we can easily express the procedure explicitly:

```
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, mesh=Ve.mesh()))
```

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

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

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

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

```
def compute(nx, ny, degree):
    mesh = UnitSquare(nx, ny)
    V = FunctionSpace(mesh, 'CG', degree)
    ...
    Ve = FunctionSpace(mesh, 'CG', degree=3)
    E = errornorm(u_exact, u, Ve)
    return E
```

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

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

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

The procedure above can easily be turned into Python code:

```
# Perform experiments
degree = int(sys.argv[1])
h = []  # element sizes
E = []  # errors
for nx in [4, 8, 16, 32, 64, 128]:
    h.append(1.0/nx)
    E.append(compute(nx, nx, degree))
```

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

The resulting program has the name `Poisson2D_D4.py`. Running this program for first-order elements yields the output

```
h=  1.25E-01 r=1.76
h=  6.25E-02 r=1.94
h=  3.12E-02 r=1.98
h=  1.56E-02 r=2.00
h=  7.81E-03 r=2.00
```

That is, we approach the expected second-order convergence of linear Lagrange elements as the meshes become sufficiently fine. Running the program for third-order elements results in the expected value $r = 4$:

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

Checking convergence rates is the next best method for verifying PDE codes (the best being exact recovery of a solution as in Chapter 1.4 and many other places in this tutorial).

Flux Functionals. To compute flux integrals like (20) we need to define the $\boldsymbol{n}$ vector, referred to as *facet normal* in FEniCS. If Γ is the complete boundary we can perform the flux computation by

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

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 Chapter 6.3. Assuming that the part corresponds to subdomain no. 0, the relevant form for the flux is `-p*dot(grad(u), n)*ds(0)`.

1.8 Quick Visualization with VTK

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

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

We have made a program `membrane1v.py` for the membrane deflection problem in Chapter 1.5 and added various demonstrations of Viper capabilities. You are encouraged to play around with `membrane1v.py` and modify the code as you read about various features. The `membrane1v.py` program solves the two-dimensional Poisson equation for a scalar field `w` (the membrane deflection).

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')
```

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

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

```
viz1 = plot(w,
            wireframe=False,
            title='Scaled membrane deflection',
            rescale=False,
            axes=True,              # include axes
            basename='deflection',  # default plotfile name
            )

viz1.elevate(-65) # tilt camera -65 degrees (latitude dir)
viz1.set_min_max(0, 0.5*max_w)  # color scale
viz1.update(w)    # bring settings above into action
viz1.write_png('deflection.png')
viz1.write_ps('deflection', format='eps')
```

The `format` argument in the latter line can also take the values `'ps'` for a standard PostScript file and `'pdf'` for a PDF file. Note the necessity of the `viz.update(w)` call – without it we will not see the effects of tilting the camera and changing the color scale. Figure 1 shows the resulting scalar surface.

1.9 Combining Dirichlet and Neumann Conditions

Let us make a slight extension of our two-dimensional Poisson problem from Chapter 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).

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

$$\begin{aligned} -\Delta u &= f \text{ in } \Omega, & (21)\\ u &= u_0 \text{ on } \Gamma_D, & (22)\\ -\frac{\partial u}{\partial n} &= g \text{ on } \Gamma_N & (23)\end{aligned}$$

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

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

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 at the Γ_D. We have

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

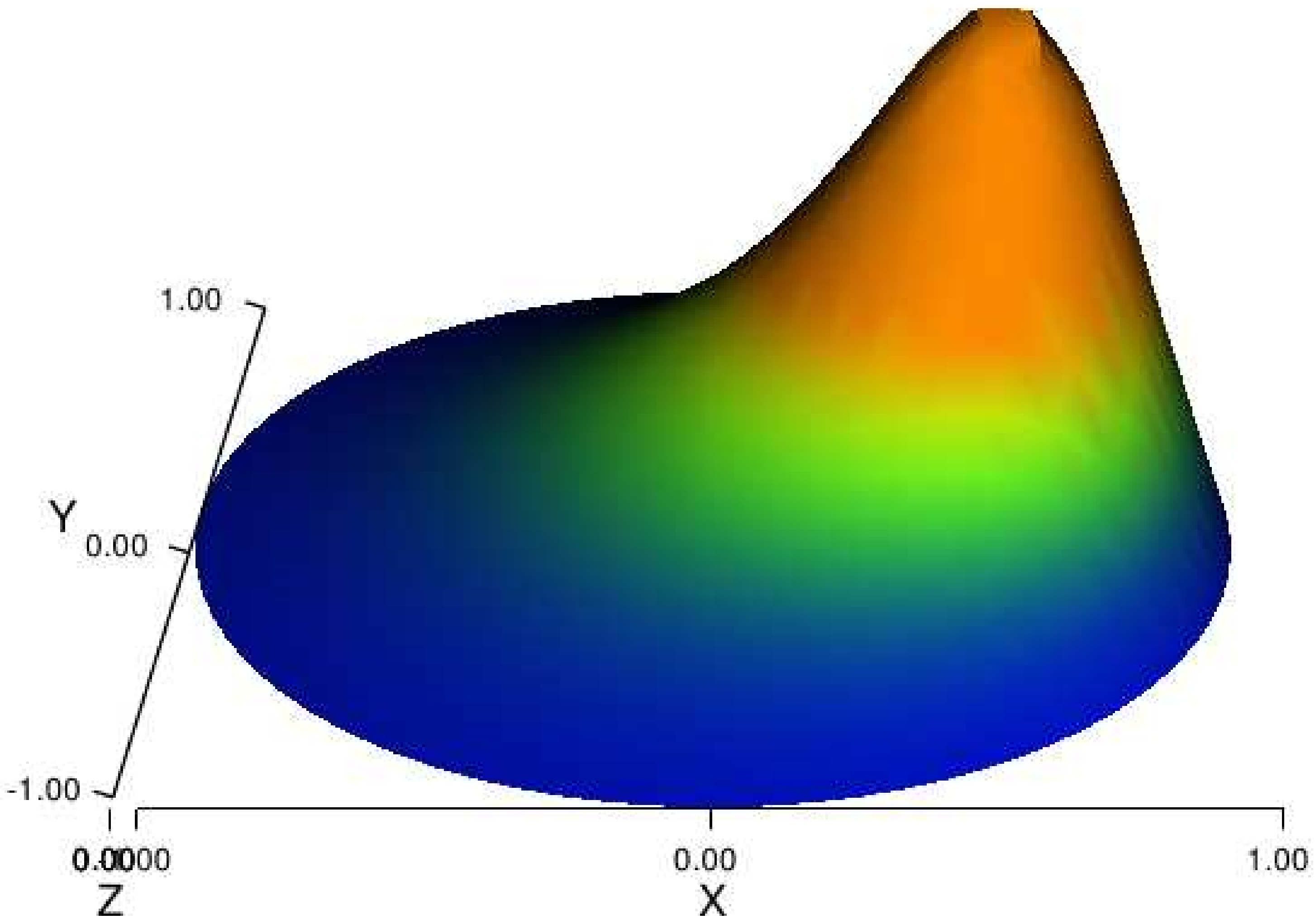


Fig. 1. Plot of the deflection of a membrane.

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 at Γ_N. The resulting weak form reads

$$\int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x + \int_{\Gamma_N} gv \,\mathrm{d}s = \int_\Omega fv \,\mathrm{d}x\,. \tag{24}$$

Expressing (24) in the standard notation $a(u,v) = L(v)$ is straightforward with

$$a(u,v) = \int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x, \tag{25}$$

$$L(v) = \int_\Omega fv \,\mathrm{d}x - \int_{\Gamma_N} gv \,\mathrm{d}s\,. \tag{26}$$

How does the Neumann condition impact the implementation? The code in the file `Poisson2D_D2.py` remains almost the same. Only two adjustments are necessary:

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

Step 1 can be coded as

```
class DirichletBoundary(SubDomain):
    def inside(self, 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

```
class DirichletBoundary(SubDomain):
    def inside(self, x, on_boundary):
        return on_boundary and (x[0] == 0 or x[0] == 1)
```

We remark that testing for an exact match of real numbers, as in `x[0] == 1`, is not good programming practice, because small round-off errors in the computation of the `x` values could make the outcome `False` even though `x` lies on the Dirichlet boundary. A better test is to check for equality with a tolerance:

```
class DirichletBoundary(SubDomain):
    def inside(self, 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]', V=V)
L = f*v*dx - g*v*ds
```

The `ds` variable implies a boundary integral, while `dx` implies an intergral over the domain Ω. No more modifications are necessary. Running the resulting program, found in the file `Poisson2D_DN1.py`, shows a successful verification – u equals the exact solution at all the nodes, regardless of how many elements we use.

1.10 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 at 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 Chapter 1.9 and define two separate functions for the two Dirichlet conditions:

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

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 $g = -4y$. For the left boundary Γ_0 we define the usual triple of a function for the boundary value, a subclass of `SubDomain` for defining the boundary of interest, and a `DirichletBC` instance:

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

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

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

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

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

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

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

The various essential conditions are then collected in a list and passed onto our problem object of type `VariationalProblem`:

```
bc = [Gamma_0, Gamma_1]
...
problem = VariationalProblem(a, L, bc)
```

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

The file `Poisson2D_DN2.py` contains a complete program which demonstrates the constructions above. An extended example with multiple Neumann conditions would have been quite natural now, but this requires marking various parts of the boundary using the mesh function concept and is therefore left to Chapter 6.3.

1.11 A Linear Algebra Formulation

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

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

which is nothing but a linear system,

$$AU = b,$$

where the entries in A and b are given by

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

The examples so far have constructed a `VariationalProblem` instance and called its `solve` method to compute the solution `u`. The `VariationalProblem` instance creates a linear system $AU = b$ and calls an appropriate solution method for such systems. An alternative is dropping the use of a `VariationalProblem` instance and instead asking FEniCS to create the matrix A and right-hand side b, and then solve for the solution vector U of the linear system. The relevant statements read

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

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

The matrix A and vector b are first assembled without incorporating essential (Dirichlet) boundary conditions. Thereafter, the `bc.apply(A, b)` call performs the necessary modifications to the linear system. The first three statements above can alternatively be carried out by[5]

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

When we have multiple Dirichlet conditions stored in a list `bc`, as explained in Chapter 1.10, we must apply each condition in `bc` to the system:

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

Alternatively, we can make the call

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

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

```
if mesh.numCells() < 16:  # print for small meshes only
    print A.array()
    print b.array()
```

[5] The essential boundary conditions are now applied to the element matrices and vectors prior to assembly.

```
bc.apply(A, b)
if mesh.numCells() < 16:
    print A.array()
    print b.array()
```

You will see that `A` is modified in a symmetric way: for each degree of freedom that is known, the corresponding row and column is zero'ed out and 1 is placed on the main diagonal. The right-hand side `b` is modified accordingly (the column times the value of the degree of freedom is subtracted from `b`, and then the corresponding entry in `b` is replaced by the known value of the degree of freedom).

Sometimes it can be handy to transfer the linear system to Matlab or Octave for futher analysis, e.g., computation of eigenvalues of A. This is easily done by opening a `File` instance with a filename extension `.m` and dump the `Matrix` and `Vector` instances as follows:

```
mfile = File('A.m'); mfile << A
mfile = File('b.m'); mfile << b
```

The data files `A.m` and `b.m` can be loaded directly into Matlab or Octave.

The complete code where our Poisson problem is solved by forming the linear system $AU = b$ explicitly, is stored in the files `Poisson2D_DN_la1.py` (one common Dirichlet condition) and `Poisson2D_DN_la2.py` (two separate Dirichlet conditions).

Creating the linear system explicitly in the user's program, as an alternative to using a `VariationalProblem` instance, 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. Chapters 3.2 and 3.3 deal with this topic in detail.

1.12 A Variable-Coefficient Poisson Problem

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

$$\begin{aligned} -\nabla\cdot[p(x,y)\nabla u(x,y)] &= f(x,y) \quad \text{in } \Omega, \\ u(x,y) &= u_0(x,y) \text{ on } \partial\Omega\,. \end{aligned} \tag{27}$$

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

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

What are the modifications we need to do in the `Poisson2D_D2.py` program from Chapter 1.4?

1. `f` must be an `Expression` since it is no longer a constant,
2. a new `Expression` `p` must be defined for the variable coefficient,
3. the variational problem is slightly changed.

First we address the modified variational problem. Multiplying the PDE in (27) 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 Chapter 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

$$a(u,v) = \int_\Omega p\nabla u\cdot\nabla v\,\mathrm{d}x, \tag{28}$$

$$L(v) = \int_\Omega fv\,\mathrm{d}x\,. \tag{29}$$

In the code from Chapter 1.3 we must replace

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

by

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

The definitions of `p` and `f` read

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

No additional modifications are necessary. The complete code can be found in in the file `Poisson2D_Dvc.py`. You can run it and confirm that it recovers the exact u at the nodes.

The flux $-p\nabla u$ may be of particular interest in variable-coefficient Poisson problems. As explained in Chapter 1.6, 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 $w\in V_h^{(g)}$ such that

$$a(w,v) = L(v^g)\quad \forall v\in \hat{V}_h^{(g)}, \tag{30}$$

where

$$a(w,v) = \int_\Omega w\cdot v\,\mathrm{d}x, \tag{31}$$

$$L(v^g) = \int_\Omega (-p\nabla u)\cdot v\,\mathrm{d}x\,. \tag{32}$$

This problem is identical to the one in Chapter 1.6, except that p enters the integral in L.

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

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

a = dot(w, v)*dx
L = dot(-p*grad(u), v)*dx
problem = VariationalProblem(a, L)
flux = problem.solve()
```

The convenience function `project` was made to condense the frequently occuring statements above:

```
flux = project(-p*grad(u),
               VectorFunctionSpace(mesh, 'CG', 1))
```

Plotting the flux vector field is naturally as easy as plotting the gradient in Chapter 1.6:

```
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))')
```

Data analysis of the nodal values of the flux field may conveniently apply the underlying `numpy` arrays:

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

The program `Poisson2D_Dvc.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 Chapter 1.13.

1.13 Visualization of Structured Mesh Data

When finite element computations are done on a structured rectangular mesh, maybe with uniform partitioning, VTK-based tools for completely unstructured 2D/3D meshes are not required. Instead we can use visualization tools for structured data, like the data appearing in finite difference simulations and image analysis. We shall demonstrate the potential of such tools.

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

Let us go back to the Poisson2D_Dvc.py code from Chapter 1.12 and map u onto a BoxField instance:

```
from scitools.BoxField import *
u_box = dolfin_function2BoxField(u, mesh, (nx,ny), uniform_mesh=True)
```

Here, nx and ny are the number of divisions in each space direction that were used when calling UnitSquare to make the mesh instance. The result u_box is a BoxField instance that supports "finite difference" indexing and an underlying grid suitable for numpy operations on 2D data. Also 1D and 3D functions in DOLFIN can be turned into BoxField instances.

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

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

For instance, the x coordinates are reached by u_box.grid.coor[X], where X is an integer (0) imported from scitools.BoxField. The grid attribute is an instance of class UniformBoxGrid.

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

Gnuplot as plotting program, and this is a highly relevant choice for scalar fields over two-dimensional, structured meshes, or for curve plots along lines through the domain.

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

```
from scitools.easyviz import contour, title, hardcopy
contour(u_box.grid.coorv[X], u_box.grid.coorv[Y], u_box.values,
        5, clabels='on')
title('Contour plot of u')
hardcopy('u_contours.eps')

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

The resulting plot can be viewed in Figure 3a. The `contour` function needs arrays with the x and y coordinates expanded to 2D arrays (in the same way as demanded when making vectorized `numpy` calculations of arithmetic expressions over all grid points). The correctly expanded arrays are stored in `grid.coorv`. The above call to `contour` creates 5 equally spaced contour lines, and with `clabels='on'` the contour values can be seen in the plot.

Other functions for visualizing 2D scalar fields are `surf` and `mesh` as known from Matlab. Because the `from dolfin import *` statement imports several names that are also present in `scitools.easyviz` (e.g., `plot`, `mesh`, and `figure`), we use functions from the latter package through a module prefix `ev` (for easyviz) from now on:

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

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

Figure 2 exemplifies the surfaces arising from the two plotting commands above. You can type `pydoc scitools.easyviz` in a terminal window to get a full tutorial.

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

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

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

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

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

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

```
ev.figure()
flux_x_box = dolfin_function2BoxField(flux_x, mesh, (nx,ny),
                                      uniform_mesh=True)
x, fluxval, y_fixed, snapped = \
      flux_x_box.gridline(start, direction=X)
y = y_fixed
flux_x_exact = -(x + y)*2*x
ev.plot(x, fluxval, 'r-',
        x, flux_x_exact, 'b-',
        legend=('numerical (projected) flux', 'exact flux'),
        title='Flux in x-direction (at y=%g)' % y_fixed,
        hardcopy='flux.eps')
```

As seen from Figure 3b, the numerical flux is accurate except in the elements closest to the boundaries.

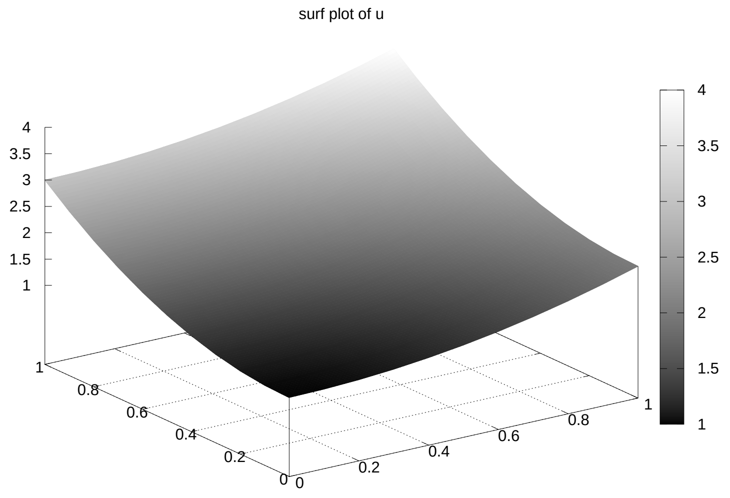
It should be easy with the information above to a transform finite element field over a uniform rectangular or box-shaped mesh to the corresponding `BoxField` instance and perform Matlab-style visualizations of the whole field or the field over planes or along lines through the domain. By the transformation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

1.14 Parameterizing the Number of Space Dimensions

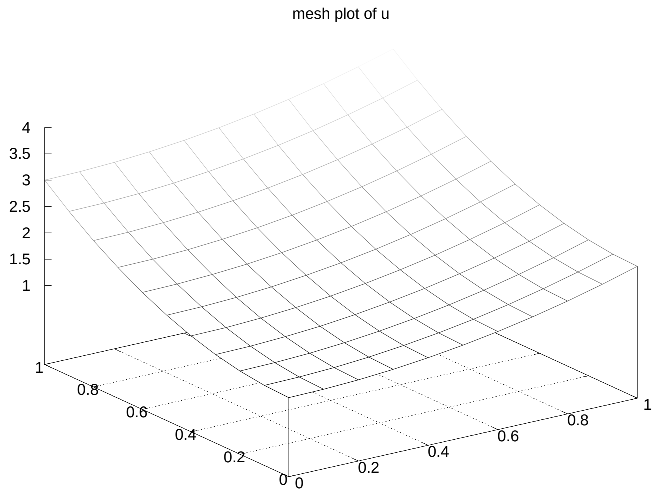
FEniCS makes it is easy to write a unified simulation code that can operate in 1D, 2D, and 3D. We will conveniently make use of this feature in forthcoming examples. The relevant technicalities are therefore explained below.

Consider the simple problem

$$u''(x) = 2 \text{ in } [0,1], \quad u(0) = 0,\ u(1) = 0, \tag{33}$$

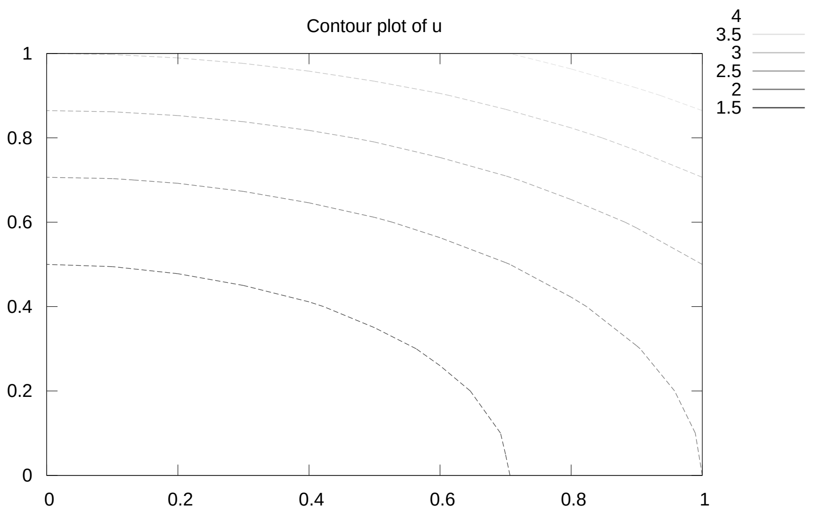


(a)

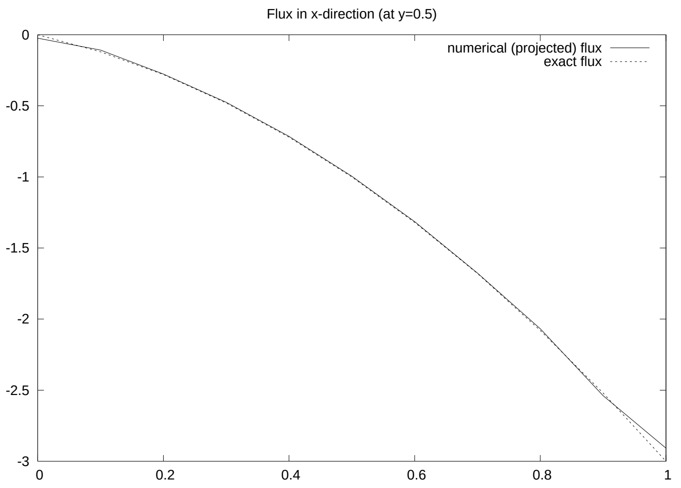


(b)

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



(a)



(b)

Fig. 3. Examples on plots created by transforming the finite element field to a field on a uniform, structured 2D grid: (a) contour plot of the solution; (b) curve plot of the exact flux $-p\partial u/\partial x$ against the corresponding projected numerical flux.

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

$$\begin{aligned} \Delta u &= 2 \text{ in } \Omega,\\ u &= 0 \text{ on } \Gamma_0,\\ u &= 1 \text{ on } \Gamma_1,\\ \frac{\partial u}{\partial n} &= 0 \text{ on } \partial\Omega \cap (\Gamma_0 \cup \Gamma_1)\,, \end{aligned} \qquad (34)$$

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

Implementing (34) for any d is no more complicated than solving a dimension-specific problem. The only non-trivial part of the code is actually to define the mesh. We use the command line to provide user-input to the program. The first argument can be the degree of the polynomial in the finite element basis functions. Thereafter, we supply the cell divisions in the various spatial directions. The number of command-line arguments will then imply the number of space dimensions. For example, writing `3 10 3 4` on the command-line means that we want to approximate u by piecewise polynomials of degree 3, and that the domain is a three-dimensional cube with $10 \times 3 \times 4$ divisions in the x, y, and z directions, respectively. Each of the $10 \times 3 \times 4 = 120$ boxes will be divided into six tetrahedras. The Python code can be quite compact:

```
degree = int(sys.argv[1])
divisions = [int(arg) for arg in sys.argv[2:]]
d = len(divisions)
domain_type = [UnitInterval, UnitSquare, UnitCube]
mesh = domain_type[d-1](*divisions)
V = FunctionSpace(mesh, 'CG', degree)
```

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

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

is equivalent to

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

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

```
tol = 1E-14   # tolerance for coordinate comparisons
class DirichletBoundary0(SubDomain):
    def inside(self, x, on_boundary):
        return on_boundary and abs(x[0]) < tol

class DirichletBoundary1(SubDomain):
    def inside(self, x, on_boundary):
        return on_boundary and abs(x[0] - 1) < tol

bc0 = DirichletBC(V, Constant(mesh, 0), DirichletBoundary0())
bc1 = DirichletBC(V, Constant(mesh, 1), DirichletBoundary1())
bc = [bc0, bc1]
```

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

```
v = TestFunction(V)
u = TrialFunction(V)
f = Constant(mesh, -2)
a = dot(grad(u), grad(v))*dx
L = f*v*dx

problem = VariationalProblem(a, L, bc)
u = problem.solve()
```

The complete code is found in `Poisson123D_DN1.py`.

Observe that if we actually want to test variations in one selected space direction, parameterized by `e`, we only need to replace `x[0]` in the code by `x[e]` (!). The parameter `e` could be given as the second command-line argument. This extension appears in the file `Poisson123D_DN2.py`. You can run a 3D problem with this code where u varies in, e.g., z direction and is approximated by, e.g., a 5-th degree polynomial. For any legal input the numerical solution coincides with the exact solution at the nodes (because the exact solution is a second-order polynomial).

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:

$$-\nabla \cdot (q(u)\nabla u) = f\,. \tag{35}$$

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

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 is the unit hypercube $[0,1]^d$ in d dimensions, $q(u) = (1+u)^m$, $f = 0$, $u = 0$ for $x_0 = 0$, $u = 1$ for $x_0 = 1$, and $\partial u/\partial n = 0$ at all other boundaries $x_i = 0$ and $x_i = 1$, $i = 1, \ldots, d-1$. The coordinates are now represented by the symbols $x_0, \ldots, x_{d-1}$. The exact solution is then

$$u(x_0, \ldots, x_d) = \left((2^{m+1} - 1)x_0 + 1\right)^{1/(m+1)} - 1\,. \tag{36}$$

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

$$F(u; v) = 0 \quad \forall v \in \hat{V}, \tag{37}$$

where

$$F(u; v) = \int_\Omega q(u) \nabla u \cdot \nabla v \,\mathrm{d}x, \tag{38}$$

and

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

The discrete problem arises as usual by restricting V and $\hat{V}$ to a pair of discrete spaces: Find $u_h \in V_h$ such that

$$F(u_h; v) = 0 \quad \forall v \in \hat{V}_h, \tag{39}$$

with $u_h = \sum_{j=1}^N U_j \phi_j$. Since F is a nonlinear function of u_h, (39) gives rise to a system of nonlinear algebraic equations. From now on the interest is only in the discrete problem, and as mentioned in Chapter 1.2, we simply write u instead of u_h to get a closer notation between the mathematics and the Python code. When the exact solution needs to be distinguished, we denote it by u_e.

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

2.1 Picard Iteration

Picard iteration is an easy way of handling nonlinear PDEs: we simply use a known, previous solutions in the nonlinear terms such that these terms become linear in the unknown u. For our particular problem, this means that 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*

$$\nabla \cdot \left(q(u^k)\nabla u^{k+1}\right) = 0, \quad k = 0, 1, \ldots \tag{40}$$

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

We can easily formulate a variational problem for u^{k+1} from Equation (40). Equivalently, we can use u^k in $q(u)$ in the nonlinear variational problem (38) to obtain the same linear variational problem. In both cases, the problem consists of seeking $u^{k+1} \in V$ such that

$$F(u^{k+1}; v) = 0 \quad \forall v \in \hat{V}, \quad k = 0, 1, \ldots, \tag{41}$$

with

$$F(u^{k+1}; v) = \int_\Omega q(u^k)\nabla u^{k+1} \cdot \nabla v \,\mathrm{d}x\,. \tag{42}$$

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

$$a(u^{k+1}, v) = L(v), \tag{43}$$

with

$$a(u, v) = \int_\Omega q(u^k)\nabla u \cdot \nabla v \,\mathrm{d}x \tag{44}$$

$$L(v) = 0\,. \tag{45}$$

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

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

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

# Define variational problem
v = TestFunction(V)
u = TrialFunction(V)
uk = interpolate(Expression('0.0', V=V), V)  # previous (known) u
a = dot(q(uk)*grad(u), grad(v))*dx
f = Constant(mesh, 0.0)
L = f*v*dx

# Picard iterations
u = Function(V)     # new unknown function
eps = 1.0           # error measure ||u-uk||
tol = 1.0E-5        # tolerance
iter = 0            # iteration counter
maxiter = 25        # max no of iterations allowed
while eps > tol and iter < maxiter:
    iter += 1
    problem = VariationalProblem(a, L, bc)
    u = problem.solve()
    diff = u.vector().array() - uk.vector().array()
    eps = numpy.linalg.norm(diff, ord=numpy.Inf)
    print 'Norm, iter=%d: %g' % (iter, eps)
    uk.assign(u)    # update for next iteration
```

We need to define the previous solution in the iterations, `uk`, as a finite element function so that `uk` can be updated with `u` at the end of the loop. We may create the initial `Function` `uk` 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/infinity 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 `nlPoisson_Picard.py` contains the complete code for this problem. The implementation is d dimensional, with mesh construction and setting of Dirichlet conditions as explained in Chapter 1.14. For a 33×33 grid with $m = 2$ we need 9 iterations for convergence when the tolerance is 10^{-5}.

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 (38), the discrete version (39) 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 (39)):

$$F_i(U_1,\ldots,U_N) \equiv \sum_{j=1}^N \int_\Omega \nabla\hat{\phi}_i \cdot \left(q(\sum_{\ell=1}^N U_\ell\phi_\ell)\nabla\phi_j U_j \right) \mathrm{d}x = 0, \quad i = 1,\ldots,N\,. \tag{46}$$

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

$$\sum_{j=1}^N \frac{\partial}{\partial U_j} F_i(U_1^k,\ldots,U_N^k)\delta U_j = -F_i(U_1^k,\ldots,U_N^k), \tag{47}$$

$$U_j^{k+1} = U_j^k + \omega\delta U_j, \tag{48}$$

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

We need, in a program, to compute the Jacobian matrix $\partial F_i/\partial U_j$ and the right-hand side vector $-F_i$. Our present problem has F_i given by (46). The derivative $\partial F_i/\partial U_j$ becomes

$$\int\limits_{\Omega} \left[\nabla\hat{\phi}_i \cdot ((q'(\sum_{\ell=1}^{N} U_\ell^k \phi_\ell)\phi_j \nabla(\sum_{j=1}^{N} U_j^k \phi_j)) + \nabla\hat{\phi}_i \cdot (q(\sum_{\ell=1}^{N} U_\ell^k \phi_\ell)\nabla\phi_j) \right] \mathrm{d}x \,. \tag{49}$$

The following results were used to obtain (49):

$$\frac{\partial u}{\partial U_j} = \frac{\partial}{\partial U_j}\sum_{j=1}^{N} U_j\phi_j = \phi_j, \quad \frac{\partial}{\partial U_j}\nabla u = \nabla\phi_j, \quad \frac{\partial}{\partial U_j}q(u) = q'(u)\phi_j \,. \tag{50}$$

We can reformulate the Jacobian matrix in (49) by introducing the short notation $u^k = \sum_{j=1}^{N} U_j^k \phi_j$:

$$\frac{\partial F_i}{\partial U_j} = \int_\Omega \left[q'(u^k)\phi_j \nabla u^k \cdot \nabla\hat{\phi}_i + q(u^k)\nabla\phi_j \cdot \nabla\hat{\phi}_i \right] \mathrm{d}x \,. \tag{51}$$

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

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

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

$$\int_\Omega \left[q'(u^k)\delta u \nabla u^k \cdot \nabla v + q(u^k)\nabla\delta u \cdot \nabla v \right] \mathrm{d}x = -\int_\Omega q(u^k)\nabla u^k \cdot \nabla v \,\mathrm{d}x \,. \tag{52}$$

Equation (52) fits the standard form $a(u, v) = L(v)$ with

$$\begin{aligned} a(u, v) &= \int_\Omega \left[q'(u^k)\delta u \nabla u^k \cdot \nabla v + q(u^k)\nabla\delta u \cdot \nabla v \right] \mathrm{d}x \\ L(v) &= -\int_\Omega q(u^k)\nabla u^k \cdot \nabla v \,\mathrm{d}x \,. \end{aligned}$$

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

We now turn to the implementation. To obtain a good initial guess u^0, we can solve a simplified, linear problem, typically with $q(u) = 1$, which yields the standard Laplace equation $\Delta u^0 = 0$. The receipe for solving this problem appears in Chapters 1.2, 1.3, and 1.9. The code for computing u^0 becomes as follows:

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

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

Gamma_0  = DirichletBC(V, Constant(mesh, 0.0),
                       LeftDirichletBoundary())
Gamma_1 = DirichletBC(V, Constant(mesh, 1.0),
                      RightDirichletBoundary())
bc_u = [Gamma_0, Gamma_1]

# Define variational problem for initial guess (q(u)=1, i.e., m=0)
v = TestFunction(V)
u = TrialFunction(V)
a = dot(grad(u), grad(v))*dx
f = Constant(mesh, 0.0)
L = f*v*dx
A, b = assemble_system(a, L, bc_u)
uk = Function(V)
solve(A, uk.vector(), b)
```

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

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

```
Gamma_0_du  = DirichletBC(V, Constant(mesh, 0.0),
                          LeftDirichletBoundary())
Gamma_1_du = DirichletBC(V, Constant(mesh, 0.0),
                          RightDirichletBoundary())
bc_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 = uk + omega*du
a = dot(q(uk)*grad(du), grad(v))*dx + \
    dot(Dq(uk)*du*grad(uk), grad(v))*dx
L = -dot(q(uk)*grad(uk), grad(v))*dx
```

The Newton iteration loop is very similar to the Picard iteration loop in Chapter 2.1:

```
du = Function(V)
u  = Function(V)  # u = uk + 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, bc_du)
    solve(A, du.vector(), b)
    eps = numpy.linalg.norm(du.vector().array(), ord=numpy.Inf)
    print 'Norm:', eps
    u.vector()[:] = uk.vector() + omega*du.vector()
    uk.assign(u)
```

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

```
u.assign(uk)  # u = uk
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` instance. 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 Chapter 1.14. The complete program appears in the file `nlPoisson_algNewton.py`.

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 of Chapter 2.2.

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

$$u^{k+1} = u^k + \delta u \tag{53}$$

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

$$q(u^{k+1}) = q(u^k) + q'(u^k)\delta u + \mathcal{O}((\delta u)^2) \approx q(u^k) + q'(u^k)\delta u, \tag{54}$$

and dropping other nonlinear terms in δu, we get

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

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

$$\nabla\cdot\big(q(u^k)\nabla\delta u\big) + \nabla\cdot\big(q'(u^k)\delta u\nabla u^k\big) = -\nabla\cdot\big(q(u^k)\nabla u^k\big)\,, \tag{55}$$

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

$$\int_\Omega \big(q(u^k)\nabla\delta u\cdot\nabla v + q'(u^k)\delta u\nabla u^k\cdot\nabla v\big)\,\mathrm{d}x = \int_\Omega q(u^k)\nabla u^k\cdot\nabla v\,\mathrm{d}x\,. \tag{56}$$

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

$$a(\delta u, v) = \int_\Omega \big(q(u^k)\nabla\delta u\cdot\nabla v + q'(u^k)\nabla u^k\cdot\nabla v\big)\,\mathrm{d}x, \tag{57}$$

$$L(v) = \int_\Omega q(u^k)\nabla u^k\cdot\nabla v\,\mathrm{d}x\,. \tag{58}$$

The continuous function spaces V and $\hat{V}$, and their discrete counterparts, V_h and $\hat{V}_h$, are as in the linear Poisson problem from Chapter 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 $a_0(u, v) = \int_\Omega \nabla u\cdot\nabla v\,\mathrm{d}x$ and $L(v) = 0$. Thereafter, we enter a loop and solve $a(\delta u, v) = L(v)$ for δt and compute a new approximation $u^{k+1} = u^k + \delta u$. Looking at (58) and (58), we see that the variational form is the same as for the Newton method at the algebraic level in Chapter 2.2. Since Newton's method at the algebraic level required some "backward" construction of the underlying weak forms, FEniCS users may prefer Newton's method at the PDE level, which is more straightforward. There is seemingly no need for differentiations to derive a Jacobian matrix, but a mathematically equivalent derivation is done when nonlinear terms are linearized using the first two Taylor series terms and when products in the perturbation δu are neglected.

The implementation is identical to the one in Chapter 2.2 and is found in the file `nlPoisson_pdeNewton.py` (for the fun of it we use a `VariationalProblem` instance instead of assembling a matrix and vector and calling `solve`). 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 (Chapter 2.2).

2.4 Solving the Nonlinear Variational Problem Directly

DOLFIN has a built-in Newton solver and is able to automate the computation of nonlinear, stationary boundary-value problems. The automation is demonstrated next. A nonlinear variational problem (37) can be solved by

```
VariationalProblem(a, L, nonlinear=True)
```

where `L` corresponds to the form $F(u; v)$ in (37) and `a` is a form for the derivative of `L`.

The `L` form is straightforwardly defined (assuming `q(u)` is coded):

```
v = TestFunction(V)
u = Function(V)  # the unknown
L = dot(q(u)*grad(u), grad(v))*dx
```

The derivative `a` of `L` is formally the Gateaux derivative of $F(u; v)$ in the direction of the trial function. Technically, this Gateaux derivative is derived by computing

$$\lim_{\epsilon\rightarrow 0} \frac{d}{d\epsilon} F_i(u^k + \epsilon\delta u; v) \tag{59}$$

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

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

and obtain

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

which leads to

$$\int_\Omega \nabla v\cdot\left[q'(u^k)\delta u\nabla(u^k) + q(u^k)\nabla\delta u\right]\,\mathrm{d}x, \tag{60}$$

as $\epsilon\rightarrow 0$. This last expression is the Gateaux derivative of F and is denoted by $a(\delta u, v)$. The corresponding implementation goes as

```
du = TrialFunction(V)
a = dot(q(u)*grad(du), grad(v))*dx + \
    dot(Dq(u)*du*grad(u), grad(v))*dx
```

The UFL language we use to specify weak forms supports differentiation of forms. This means that when `L` is given as above, we can simply compute the Gateaux derivative by

```
a = derivative(L, u, du)
```

The differentiation is done symbolically so no numerical approximation formulas are involved. The `derivative` function is obviously very convenient in problems where differentiating `L` by hand implies lengthy calculations.

The solution of the nonlinear problem is now a question of two statements:

```
problem = VariationalProblem(a, L, bc, nonlinear=True)
u = problem.solve(u)
```

The `u` we feed to `problem.solve` is filled with the solution and returned, implying that the `u` on the left-hand side actually refers to the same `u` as provided on the right-hand side[6]. The file `nlPoisson_vp1.py` contains the complete code, where `a` is calculated manually, while `nlPoisson_vp2.py` is a counterpart where `a` is computed by `derivative(L, u, du)`. The latter file represents clearly the most automated way of solving the present nonlinear problem in FEniCS.

3 Time-Dependent Problems

The examples in Chapter 1 illustrate that solving linear, stationary PDE problems with the aid of FEniCS is easy and requires little programming. That is, FEniCS automates the spatial discretization by the finite element method. The solution of nonlinear problems, as we showed in Chapter 37, can also be automated (cf. Chapter 2.4), but many scientists will prefer to code the solution strategy of the nonlinear problem themselves and experiment with various combination of strategies in difficult problems. Time-dependent problems are somewhat similar in this respect: we have to add a time discretization scheme, which is often quite simple, making it natural to explicitly code the details of the scheme so that the programmer have full control. We shall explain how easily this is accomplished through examples.

[6] Python has a convention that all input data to a function or class method are represented as arguments, while all output data are returned to the calling code. Data used as both input and output, as in this case, will then be arguments and returned. It is not necessary to have a variable on the left-hand side, as the function instance is modified correctly anyway, but it is convention that we follow here.

3.1 A Diffusion Problem and Its Discretization

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

$$\frac{\partial u}{\partial t} = \Delta u + f \text{ in } \Omega, \tag{61}$$

$$u = u_0 \text{ on } \partial\Omega, \tag{62}$$

$$u = I \text{ for } t = 0\,. \tag{63}$$

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

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

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

$$\frac{\partial}{\partial t}u^k = \Delta u^k + f^k\,. \tag{64}$$

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

$$\frac{\partial}{\partial t}u^k \approx \frac{u^k - u^{k-1}}{\Delta t}, \tag{65}$$

where Δt is the time discretization parameter. Inserting (65) in (64) yields

$$\frac{u^k - u^{k-1}}{\Delta t} = \Delta u^k + f^k\,. \tag{66}$$

This is our time-discrete version of the diffusion PDE (61). Reordering (66) so that u^k appears on the left-hand side only, shows that (66) is a recursive set of spatial (stationary) problems for u^k (assuming u^{k-1} is know from compuations at the previous time level):

$$u^0 = I, \tag{67}$$

$$u^k - \Delta u^k = u^{k-1} + \Delta t f^k, \quad k = 1, 2, \ldots \tag{68}$$

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 (67) and (68). 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 forms can be conveniently written in the standard notation: $a_0(u,v) = L_0(v)$ for (67) and $a(u,v) = L(v)$ for (68), where

$$a_0(u,v) = \int_\Omega uv \, \mathrm{d}x, \tag{69}$$

$$L_0(v) = \int_\Omega Iv \, \mathrm{d}x, \tag{70}$$

$$a(u,v) = \int_\Omega \left(uv + \Delta t \nabla u \cdot \nabla v\right) \, \mathrm{d}x, \tag{71}$$

$$L(v) = \int_\Omega \left(u^{k-1} + \Delta t f^k\right) v \, \mathrm{d}x \, . \tag{72}$$

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 V_h and $\hat{V}_h$, 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. With u^{k-1} we mean, from now on, the finite element approximation of the solution at time t_{k-1}.

Note that the forms a_0 and L_0 are identical to the forms met in Chapter 1.6, except that the unknown now is a scalar field and not a vector field. Instead of solving (67) by a finite element method, i.e., projecting I onto V_h 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 no. 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.

3.2 Implementation

Our program needs to perform the time stepping explicitly, but can rely on FEniCS to easily compute a_0, L_0, a, and L, and solve the linear systems for the unknowns. We realize that a does not depend on time, which means that its associated matrix also will be time independent. Therefore, it is wise to explicitly create matrices and vectors as in Chapter 1.11. 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 u^k and u_{prev} for the previous solution u^{k-1}:

```
define Dirichlet boundary condition ($u_0$, Dirichlet boundary, etc.)
if $u_{\mathrm{prev}}$ is to be computed by projecting $I$:
    define $a_0$ and $L_0$
    assemble matrix $M$ from $a_0$ and vector $b$ from $L_0$
    solve $MU = b$ and store $U$ in $u_{\mathrm{prev}}$
else: (interpolation)
    let $u_{\mathrm{prev}}$ interpolate $I$
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 $u$
    $t \leftarrow t + \Delta t$
    $u_{\mathrm{prev}} \leftarrow u$ (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-order polynomial in space with a linear term in time,

$$u = 1 + x^2 + \alpha y^2 + \beta t, \tag{73}$$

yields a function whose computed values at the nodes may be exact, regardless of the size of the elements and Δt, as long as the mesh is uniformly partitioned. Inserting (73) in the PDE problem (61), it follows that u_0 must be given as (73) 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 (73). Given a `mesh` and an associated function space `V`, we can specify the u_0 function as

```
alpha = 3; beta = 1.2
u0 = Expression('1 + x[0]*x[0] + alpha*x[1]*x[1] + beta*t',
                {'alpha': alpha, 'beta': beta},
                V=V)
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 instance, as shown for the `t` parameter.

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

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

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

The initial condition 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_prev = interpolate(u0, V)
# or
u_prev = project(u0, V)
```

Note that we could, as an equivalent alternative to using `project`, define a_0 and L_0 as we did in Chapter 1.6 and form a `VariationalProblem` instance. To actually recover (73) 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 will imply approximative values at the nodes).

The definition of a and L goes as follows:

```
dt = 0.3      # time step

v = TestFunction(V)
u = TrialFunction(V)
f = Constant(mesh, beta - 2 - 2*alpha)

a = u*v*dx + dt*dot(grad(u), grad(v))*dx
L = (u_prev + 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_prev.assign(u)
```

Observe that `u0.t` must be updated before `bc` applies it to enforce the Dirichlet conditions at the current time level.

The time loop above does not contain any examination of the numerical 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 = (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 is stored in the file `diffusion2D_D1.py`.

3.3 Avoiding Assembly

The purpose of this section is to present a technique for speeding up FEniCS simulators for time-dependent problems where it is possible to perform all assembly operations prior to the time loop. There are two costly operations in the time loop: assembly of the right-hand side b and solution of the linear system via the `solve` call. The assembly process involves work proportional to the number of degrees of freedom N, while the solve operation has a work estimate of $\mathcal{O}(N^{1+p})$, where $p \geq 0$. As $N \rightarrow \infty$, the solve operation will for $p > 0$ dominate, but for the values of N typically used on smaller computers, the assembly step may still represent a considerable part of the total work at each time level. Avoiding repeated assembly can therefore contribute to a significant speed-up of a finite element code in time-dependent problems.

To see how repeated assembly can be avoided, we look at the $L(v)$ form in (72), 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 Chapter 1.11, 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

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

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

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

which is nothing but two matrix-vector products,

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

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

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

and

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

We have immediate access to U^{k-1} in the program since that is the vector in the `u_prev` 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 (71) 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,

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

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

$$K_{ij} = \int_\Omega \nabla\hat{\phi}_i \cdot \nabla\phi_j \,\mathrm{d}x\,. \qquad (74)$$

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

$$a_K(u, v) = \int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x, \tag{75}$$

$$a_M(u, v) = \int_\Omega uv \,\mathrm{d}x, \,. \tag{76}$$

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

The following modifications are needed in the `diffusion2D_D1.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 $A = M + \Delta K$
4. Define f as an `Expression`
5. Interpolate the formula for f to a finite element function F^k
6. Compute $b = MU^{k-1} + \Delta tMF^k$

The relevant code segments become

```
# 1.
a_K = dot(grad(u), grad(v))*dx
a_M = u*v*dx

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

# 4.
f = Expression('beta - 2 - 2*alpha',
               {'beta': beta, 'alpha': alpha},
               V=V)

# 5. and 6.
while t <= T:
    fk = interpolate(f, V)
    Fk = fk.vector()
    b = M*u_prev.vector() + dt*M*Fk
```

The complete program appears in the file `diffusion2D_D2.py`..

3.4 A Physical Example

With the basic programming techniques for time-dependent problem from Chapters 3.3–3.2 we are ready to attack more physically realistic examples.

The next example concerns the question: How is the temperature in the ground affected by day and night variations at the earth's surface? We consider some box-shaped domain Ω in d dimensions with coordinates $x_0, \ldots, x_{d-1}$ (the problem is meaningful in 1D, 2D, and 3D). At the top of the domain, $x_{d-1} = 0$, we have an oscillating temperature

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

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

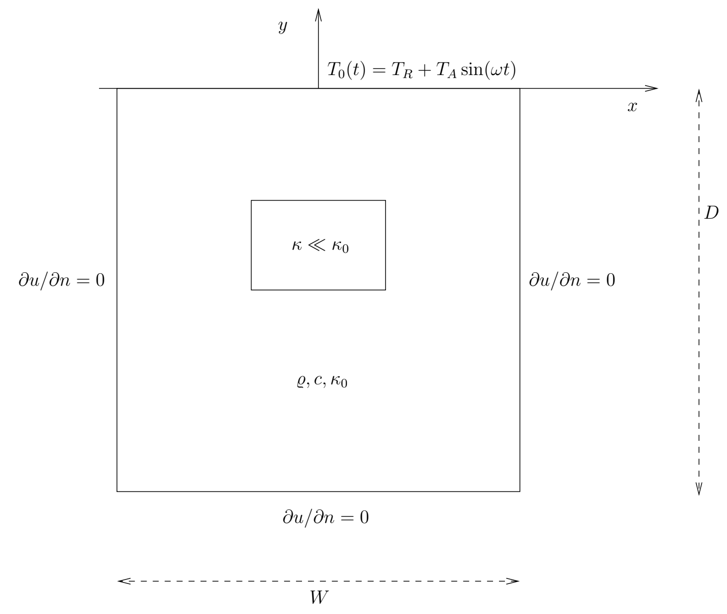

y
$T_0(t) = T_R + T_A \sin(\omega t)$
x
D
$\kappa \ll \kappa_0$
$\partial u/\partial n = 0$
$\partial u/\partial n = 0$
ϱ, c, κ_0
$\partial u/\partial n = 0$
W

Fig. 4. 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

$$\varrho c\frac{\partial T}{\partial t} = \nabla\cdot(k\nabla T) \text{ in } \Omega\times(0,T], \tag{77}$$

$$T = T_0(t) \text{ on } \Gamma_0, \tag{78}$$

$$\frac{\partial T}{\partial n} = 0 \text{ on } \partial\Omega\cap\Gamma_0, \tag{79}$$

$$T = T_R \text{ at } t = 0\,. \tag{80}$$

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

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

Bringing this time-discrete PDE on 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

$$a(T,v) = \int_\Omega \left(\varrho c T v + \theta\Delta t\kappa\nabla T\cdot\nabla v\right)\,\mathrm{d}x, \tag{81}$$

$$L(v) = \int_\Omega \left(\varrho c v T^{k-1} - (1-\theta)\Delta t\kappa\nabla T\cdot\nabla v\right)\,\mathrm{d}x\,. \tag{82}$$

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

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

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

The `Rectangle` and `Box` instances 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},
                 V=V)

class Surface(SubDomain):
    def inside(self, x, on_boundary):
        return on_boundary and abs(x[d-1]) < 1E-14

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

Quite simple values (non-physical for soil and real temperature variations) are chosen for the initial testing.

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 4. Outside this area κ is a constant κ_0. The domain of the rectangular area are 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(Function):
    def eval(self, value, x):
        """x: spatial point, value[0]: function value."""
        d = len(x)  # no of space dimensions
        material = 0  # 0: outside, 1: inside
        if d == 1:
            if -D/2. < x[d-1] < -D/2. + D/4.:
                material = 1
        elif d == 2:
            if -D/2. < x[d-1] < -D/2. + D/4. and \
               -W/4. < x[0] < W/4.:
                material = 1
        elif d == 3:
            if -D/2. < x[d-1] < -D/2. + D/4. and \
               -W/4. < x[0] < W/4. and -W/4. < x[1] < W/4.:
                material = 1
        value[0] = kappa_0 if material == 0 else kappa_1
```

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

```
kappa_0 = 0.2
kappa_1 = 0.001
kappa_str = {}
kappa_str[1] = 'x[0] > -%s/2 && x[0] < -%s/2 + %s/4 ? %g : %g' % \
               (D, D, D, kappa_1, kappa_0)
kappa_str[2] = 'x[0] > -%s/4 && x[0] < %s/4 '\
            '&& x[1] > -%s/2 && x[1] < -%s/2 + %s/4 ? %g : %g' % \
               (W, W, D, D, D, kappa_1, kappa_0)
kappa_str[3] = 'x[0] > -%s/4 && x[0] < %s/4 '\
               'x[1] > -%s/4 && x[1] < %s/4 '\
            '&& x[2] > -%s/2 && x[2] < -%s/2 + %s/4 ? %g : %g' % \
               (W, W, W, W, D, D, D, kappa_1, kappa_0)

kappa = Expression(kappa_str[d], V=V)
```

For example, in 2D `kappa_str[1]` becomes

```
x[0] > -0.5/4 && x[0] < 0.5/4 && x[1] > -1.0/2 &&
x[1] < -1.0/2 + 1.0/4 ? 1e-07 : 0.2
```

for $D = 1$ and $W = D/2$ (the string is one line, but broken into two here to fit the page width). It is very important to have a `D` that is `float` and not `int`, otherwise one gets integer divisions in the C++ expression and a completely wrong κ function.

We are now ready to define the initial condition and the `a` and `L` forms of our problem:

```
T_prev = interpolate(Constant(mesh, 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

v = TestFunction(V)
T = TrialFunction(V)
f = Constant(mesh, 0)
a = rho*c*T*v*dx + theta*dt*kappa*dot(grad(T), grad(v))*dx
L = (rho*c*T_prev*v + dt*f*v -
     (1-theta)*dt*kappa*dot(grad(T), grad(v)))*dx
```

```
A = assemble(a)
b = None  # variable used for memory savings in assemble calls
```

We could, alternatively, break `a` and `L` up in subexpressions and assemble a mass matrix and stiffness matrix, as exemplified in Chapter 3.3, to avoid assembly of `b` at every time level. This modification is straightforward and left as an exercise. The speed-up can be significant in 3D problems.

The time loop is very similar to what we have displayed in Chapter 3.2:

```
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 `diffusion123D_sin.py` contains several statements related to visualization 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 κ is constant throughout Ω. The reader is encouraged to play around with the code and test out various parameter sets:

- $T_R = 0$, $T_A = 1$, $\kappa_0 = \kappa_1 = 0.2$, $\varrho = c = 1$, $\omega = 2\pi$
- $T_R = 0$, $T_A = 1$, $\kappa_0 = 0.2$, $\kappa_1 = 0.01$, $\varrho = c = 1$, $\omega = 2\pi$
- $T_R = 0$, $T_A = 1$, $\kappa_0 = 0.2$, $\kappa_1 = 0.001$, $\varrho = c = 1$, $\omega = 2\pi$
- $T_R = 10$ C, $T_A = 10$ C, $\kappa_0 = 1.1\ \mathrm{K}^{-1}\mathrm{Ns}^{-1}$, $\kappa_0 = 2.3\ \mathrm{K}^{-1}\mathrm{Ns}^{-1}$, $\varrho = 1500\ \mathrm{kg/m}^3$, $c = 1600\ \mathrm{Nm\,kg}^{-1}\mathrm{K}^{-1}$, $\omega = 2\pi/24\ 1/\mathrm{h} = 7.27\cdot 10^{-5}\ 1/\mathrm{s}$, $D = 1.5$ m

The latter set of data is relevant for real temperature variations in the ground.

4 Controlling the Solution of Linear Systems

Several linear algebra packages, referred to as linear algebra *backends*, can be used in FEniCS to solve linear systems: PETSc, uBLAS, Epetra (Trilinos), or MTL4. Which backend to apply can be controlled by setting

```
parameters['linear algebra backend'] = backendname
```

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

The 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. The underlying PETSc objects 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.

4.1 Variational Problem Objects

Let us explain how one can choose between direct and iterative solvers. We have seen that there are two ways of solving linear systems, either we call the `solve()` method in a `VariationalProblem` instance or we call the `solve(A, U, b)` function with the assembled coefficient matrix `A`, right-hand side vector `b`, and solution vector `U`.

In case we use a `VariationalProblem` instance, named `problem`, it has a `parameters` instance that behaves like a Python dictionary, and we can use this object to choose between a direct or iterative solver:

```
problem.parameters['linear_solver'] = 'direct'
# or
problem.parameters['linear_solver'] = 'iterative'
```

Another parameter `'symmetric'` can be set to `True` if the coefficient matrix is symmetric so that a method exploiting symmetry can be utilized. For example, the default iterative solver is GMRES, but when solving a Poisson equation, the iterative solution process will be more efficient by setting the `'symmetry'` parameter so that a Conjugate Gradient is applied.

Having chosen an interative solver, we can invoke a submenu `'krylov_solver'` in the `parameters` object for setting various parameters for the iterative solver (GMRES or Conjugate Gradients, depending on whether the matrix is symmetric or not):

```
itsolver = problem.parameters['krylov_solver'] # short form
itsolver['absolute_tolerance'] = 1E-10
itsolver['relative_tolerance'] = 1E-6
itsolver['divergence_limit'] = 1000.0
itsolver['gmres_restart'] = 50
itsolver['monitor_convergence'] = True
itsolver['report'] = True
```

Here, `'divergence_limit'` governs the maximum allowable number of iterations, the `'gmres_restart'` parameter tells how many iterations GMRES performs before it restarts, `'monitor_convergence'` prints detailed information about the development of the residual of a solver, `'report'` governs whether a one-line report about the solution method and the number of iterations is written on the screen or not. The absolute and relative tolerances enter (usually residual-based) stopping criteria, which are dependent on the implementation of the underlying iterative solver in the actual backend.

When direct solver is chosen, there is similarly a submenu `'lu_solver'` to set parameters, but here only the `'report'` parameter is available (since direct solvers very soldom have any adjustable parameters). For nonlinear problems there is also submenu `'newton_solver'` where tolerances, maximum iterations, and so on, for a the Newton solver in `VariationalProblem` can be set.

A complete list of all parameters and their default values is printed to the screen by

```
info(problem.parameters, True)
```

4.2 Solve Function

For the `solve(A, x, b)` approach, a 4th argument to `solve` determines the type of method:

- `'lu'` for a sparse direct (LU decomposition) method,
- `'cg'` for the Conjugate Gradient (CG) method, which is applicable if `A` is symmetric and positive definite,
- `'gmres'` for the GMRES iterative method, which is applicable when `A` is nonsymmetric,
- `'bicgstab'` for the BiCGStab iterative method, which is applicatble when `A` is nonsymmetric.

The default solver is `'lu'`.

Good performance of an iterative method requires preconditioning of the linear system. The 5th argument to `solve` determines the preconditioner:

- `'none'` for no preconditioning.
- `'jacobi'` for the simple Jacobi (diagonal) preconditioner,

– `'sor'` for SOR preconditioning,
– `'ilu'` for incomplete LU factorization (ILU) preconditioning,
– `'icc'` for incomplete Cholesky factorization preconditioning (requires `A` to be symmetric and positive definite),
– `'amg_hypre'` for algebraic multigrid (AMG) preconditioning with the Hypre package (if available),
– `'mag_ml'` for algebraic multigrid (AMG) preconditioning with the ML package from Trilinos (if available),
– `'default_pc'` for a default preconditioner, which depends on the linear algebra backend (`'ilu'` for PETSc).

If the 5th argument is not provided, `'ilu'` is taken as the default preconditioner.

Here are some sample calls to `solve` demonstrating the choice of solvers and preconditioners:

```
solve(A, u.vector(), b)       # 'lu' is default solver
solve(A, u.vector(), cg)      # CG with ILU prec.
solve(A, u.vector(), 'gmres', 'amg_ml')  # GMRES with ML prec.
```

4.3 Setting the Start Vector

The choice of start vector for the iterations in a linear solver is often important. With the `solve(A, U, b)` function the start vector is the vector we feed in for the solution. A start vector with random numbers in the interval $[-1, 1]$ can be computed as

```
n = u.vector().array().size
u.vector()[:] = numpy.random.uniform(-1, 1, n)
solve(A, u.vector(), b, cg, ilu)
```

Or if a `VariationalProblem` instance is used, its `solve` method may take an optional `u` function as argument (which we can fill with the right values):

```
problem = VariationalProblem(a, L, bc)
n = u.vector().array().size
u.vector()[:] = numpy.random.uniform(-1, 1, n)
u = problem.solve(u)
```

The program `Poisson2D_DN_laprm.py` demonstrates the various control mechanisms for steering linear solvers as described above.

4.4 Using a Backend-Specific Solver

Here is a demo where we operate on Trilinos-specific vectors, matrices, iterative solvers, and preconditioners. Given a linear system $AU = b$, represented by a `Matrix` instance `A`, and two `Vector` instances `U` and `b`, 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.

```
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)
```

5 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.

5.1 Built-In Mesh Generation Tools

DOLFIN has a few tools for creating various types of meshes over domains with simple shape: `UnitInterval`, `UnitSphere`, `UnitSquare`, `Interval`, `Rectangle`, `Box`, `UnitCircle`, and `UnitCube`. 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 = UnitInterval(20)     # 20 cells, 21 vertices
mesh = Interval(20, -1, 1)  # domain [-1,1]

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

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

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

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

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

5.2 Transforming Mesh Coordinates

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

$$\bar{x} = a + (b - a)\left(\frac{x - a}{b - a}\right)^s \tag{83}$$

toward $x = a$, or

$$\bar{x} = a + (b-a)\left(\frac{x-a}{b-a}\right)^{1/s} \tag{84}$$

toward $x = b$

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

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

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

x_bar, y_bar = denser(x, y)
xy_bar_coor = numpy.array([x_bar, y_bar]).transpose()
mesh.coordinates()[:] = xy_bar_coor
plot(mesh, title='stretched mesh')

def cylinder(r, s):
    return [r*cos(Theta*s), r*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 convention an $M \times 2$ array so we need to take the transpose. The resulting mesh is displayed in Figure 5.

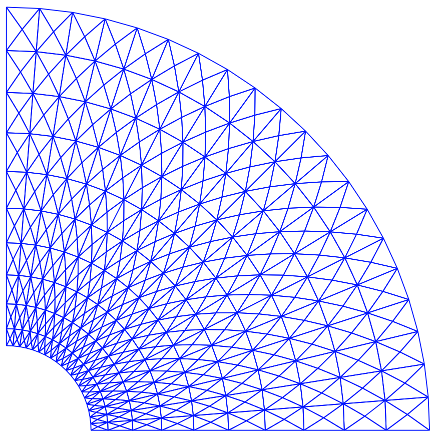


Fig. 5. Hollow cylinder generated by mapping a rectangular mesh, stretched toward the left side.

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

5.3 Separate Preprocessor Applications

6 Handling Domains with Different Materials

Solving PDEs in domains made up of different materials is a frequently encountered task. In FEniCS, this 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. Later, a multi-material problem in d space dimensions is addressed.

6.1 Working with Two Subdomains

Suppose we want to solve

$$\nabla\cdot[k(x,y)\nabla u(x,y)]=0, \tag{85}$$

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 6,

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

We define $k(x,y) = 1$ in Ω_0 and $k(x,y) = 10$ in Ω_1. As boundary conditions, we choose $u = 0$ at $x = 0$, $u = 1$ at $x = 1$, and $\partial u/\partial n = 0$ at $y = 0$ and $y = 1$. This choice implies the simple solution $u(x,y) = x$, which we should recover exactly with linear or higher order finite elements.

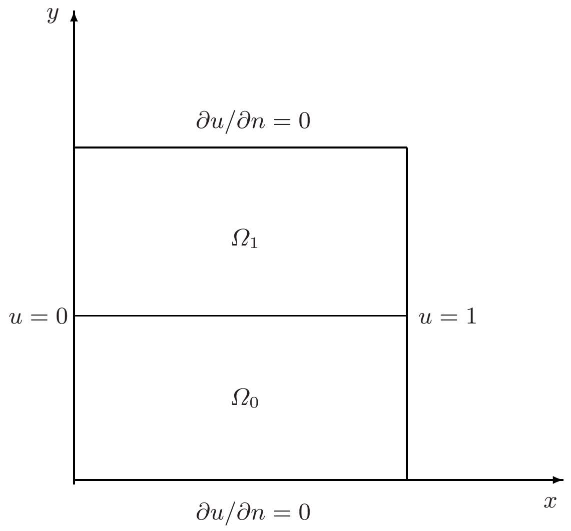


Fig. 6. Sketch of a Poisson problem with a variable coefficient that is constant in each of the two subdomains Ω_0 and Ω_1.

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

6.2 The Implementation

The new functionality in this subsection regards how to to define the subdomains Ω_0 and Ω_1. Defining a subdomain is done by creating a subclass of `SubDomain` and implementing the `inside` function. In the present case we define

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

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

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

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

Since we need to define subdomains of Ω in the present example, we must make use of a `MeshFunction` over cells. The `MeshFunction` constructor is fed with three arguments: 1) the type of value: `'int'` for integers, `'uint'` for positive (unsigned) integers, `'double'` for real numbers, and `'bool'` for logical values; 2) a `Mesh` instance, and 3) the topological dimension of the mesh entity in question: cells have topological dimension equal to the number of space dimensions in the PDE problem, and facets have one dimension lower. Alternatively, the constructor can take just a filename and initialize the `MeshFunction` from data in a file. We shall demonstrate this functionality in the next multi-material problem in Chapter 7.

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

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

Calling `subdomains.values()` returns a `numpy` array of the subdomain values. That is, `subdomain.values()[i]` is the subdomain value of cell no. `i`. This array is used to look up the subdomain or material number of a specific element.

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

```
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 `subdomain.values()`), 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(subdomains.values())):
    subdomain_no = subdomains.values()[cell_no]
    k.vector()[cell_no] = k_values[subdomain_no]
```

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

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

The `help` array is required since `choose` cannot work with `subdomain.values()` 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 Chapter 1.10, and the $a(u, v)$ and $L(v)$ forms, as in Chapter 1.12. All the details can be found in the file `Poisson2D_2mat.py`.

6.3 Multiple Neumann, Robin, and Dirichlet Conditions

Let us go back to the model problem from Chapter 1.10 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 cell facets. However, the contributions from the parts of the boundary where we have Dirichlet conditions are erased when the linear system is modified by the Dirichlet conditions. We would like, from an efficiency point of view, to integrate `v*g*ds` only over the parts of the boundary where we actually have Neumann conditions. And more importantly, in other problems one may have different Neumann conditions or other conditions like the Robin type condition. With the mesh function concept we can mark different parts of the boundary and integrate over specific parts. The same concept can also be used to treat multiple Dirichlet conditions. The forthcoming text illustrates how this is done.

Essentially, we still stick to the model problem from Chapter 1.10, but replace the Neumann condition at $y = 0$ by a *Robin condition*[7]:

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

where p and q are specified functions. Since we have prescribed a simple solution in our model problem, $u = 1 + x^2 + y^4$, we adjust p and q such that the condition holds at $y = 0$. This implies that $q = 1 + x^2 + 2y^2$ and p can be arbitrary (the normal derivative at $y = 0$: $\partial u/\partial n = -\partial u/\partial y = -4y = 0$).

Now we have four parts of the boundary: Γ_N which corresponds to the upper side $y = 1$, Γ_R which corresponds to the lower part $y = 0$, Γ_0 which corresponds to the left part $x = 0$, and Γ_1 which corresponds to the right part $x = 1$. The complete boundary-value problem reads

$$-\Delta u = -6 \text{ in } \Omega, \tag{86}$$
$$u = u_L \text{ on } \Gamma_0, \tag{87}$$
$$u = u_R \text{ on } \Gamma_1, \tag{88}$$
$$-\frac{\partial u}{\partial n} = p(u-q) \text{ on } \Gamma_R, \tag{89}$$
$$-\frac{\partial u}{\partial n} = 4y \text{ on } \Gamma_N\,. \tag{90}$$

The involved prescribed functions are $u_L == 1 + 2y^2$, $u_R = 2 + 2y^2$, $q = 1 + x^2 + 2y^2$, p is arbitrary, and $g = -4y$.

Integration by parts of $-\int_\Omega v$
Deltau $\mathrm{d}x$ becomes as usual

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

The boundary integral vanishes on $\Gamma_0 \cup \Gamma_1$, and we split the parts over Γ_N and Γ_R since we have different conditions at those parts:

$$-\int_{\partial\Omega} v\frac{\partial u}{\partial n}\,\mathrm{d}s = -\int_{\Gamma_N} v\frac{\partial u}{\partial n}\,\mathrm{d}s - \int_{\Gamma_R} v\frac{\partial u}{\partial n}\,\mathrm{d}s = \int_{\Gamma_N} vg\,\mathrm{d}s + \int_{\Gamma_R} vp(u-q)\,\mathrm{d}s\,.$$

The weak form then becomes

$$\int_\Omega \nabla u\cdot\nabla v\,\mathrm{d}x + \int_{\Gamma_N} gv\,\mathrm{d}s + \int_{\Gamma_R} p(u-q)v\,\mathrm{d}s = \int_\Omega fv\,\mathrm{d}x,$$

[7] The Robin condition is most often used to model heat transfer to the surroundings and arise naturally from Newton's cooling law.

We want to write this weak form in the standard notation $a(u,v) = L(v)$, which requires that we indentify 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:

$$\int_{\Gamma_R} vp(u-q)\,\mathrm{d}s = \int_{\Gamma_R} vpu\,\mathrm{d}s - \int_{\Gamma_R} vpq\,\mathrm{d}s\,.$$

We then have

$$a(u,v) = \int_{\Omega} \nabla u\cdot\nabla v\,\mathrm{d}x + \int_{\Gamma_R} puv\,\mathrm{d}s, \tag{91}$$

$$L(v) = \int_{\Omega} fv\,\mathrm{d}x - \int_{\Gamma_N} gv\,\mathrm{d}s + \int_{\Gamma_R} pqv\,\mathrm{d}s\,. \tag{92}$$

A natural starting point for implementation is the `Poisson2D_DN2.py` program, which we now copy to `Poisson2D_DNR.py`. The new aspects are

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

Task 1 makes use of the `MeshFunction` object, but contrary to Chapter 6.2, this is not a function over cells, but a function over cell facets. The topological dimension of cell facets is one lower than the cell interiors, so in a two-dimensional problem the dimension becomes 1. In general, the facet dimension is given as `mesh.topology().dim()-1`, which we use in the code for ease of direct reuse in other problems. The construction of a `MeshFunction` instance to mark boundary parts now reads

```
boundary_parts = \
  MeshFunction("uint", mesh, mesh.topology().dim()-1)
```

As in Chapter 6.2 we use a subclass of `SubDomain` to identify the various parts of the mesh function. Problems with domains of more complicated may set the mesh function for marking boundaries as part of the mesh generation. In our case, the $y=0$ boundary can be marked by

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

Gamma_R = LowerRobinBoundary()
Gamma_R.mark(boundary_parts, 0)
```

The code for the $y=1$ boundary is similar and is seen in `Poisson2D_DNR.py`.

The Dirichlet boundaries are marked similarly, using subdomain number 2 for Γ_0 and 3 for Γ_1:

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

Gamma_0 = LeftDirichletBoundary()
Gamma_0.mark(boundary_parts, 2)

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

Gamma_1 = RightDirichletBoundary()
Gamma_1.mark(boundary_parts, 3)
```

Specifying the `DirichletBC` instances may now make use of the mesh function (instead of a `SubDomain` subclass object) and an indicator for which subdomain each condition should be applied to:

```
u_L = Expression('1 + 2*x[1]*x[1]', V=V)
u_R = Expression('2 + 2*x[1]*x[1]', V=V)
bc = [DirichletBC(V, u_L, boundary_parts, 2),
      DirichletBC(V, u_R, boundary_parts, 3)]
```

Some functions need to be defined before we can go on with the `a` and `L` of the variational problem:

```
q = Expression('1 + x[0]*x[0] + 2*x[1]*x[1]', V=V)
p = Constant(mesh, 100)  # arbitrary function can go here
v = TestFunction(V)
u = TrialFunction(V)
f = Constant(mesh, -6.0)
```

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

The variational problem can be defined as

```
a = dot(grad(u), grad(v))*dx + p*u*v*ds(0)
L = f*v*dx - g*v*ds(1) + p*q*v*ds(0)
```

For the `ds(0)` and `ds(1)` symbols to work we must obviously connect them (or `a` and `L`) to the mesh function marking parts of the boundary. This is done by a certain keyword argument to the `assemble` function:

```
A = assemble(a, exterior_facet_domains=boundary_parts)
b = assemble(L, exterior_facet_domains=boundary_parts)
```

Then essential boundary conditions are enforced, and the system can be solved in the usual way:

```
for condition in bc: condition.apply(A, b)
u = Function(V)
solve(A, u.vector(), b)
```

At the time of this writing, it is not possible to perform integrals over different parts of the domain or boundary using the `assemble_system` function or the `VariationalProblem` instance.

7 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.

7.1 The PDE Problem

We generalize the problem in Chapter 6.1 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 7. 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 Chapter 1.14, but the layer boundaries are planes $x_0 = \text{const}$ and u varies with x_0 only.

The PDE reads

$$\nabla \cdot (k\nabla u) = 0\,. \tag{93}$$

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 7 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

$$\begin{aligned}
\nabla \cdot (k(x_0)\nabla u(x_0, \ldots, x_{d-1})) &= 0 \text{ in } \Omega,\\
u &= 0 \text{ on } \Gamma_0,\\
u &= 1 \text{ on } \Gamma_1,\\
\tfrac{\partial u}{\partial n} &= 0 \text{ on } \Gamma_N\,.
\end{aligned} \tag{94}$$

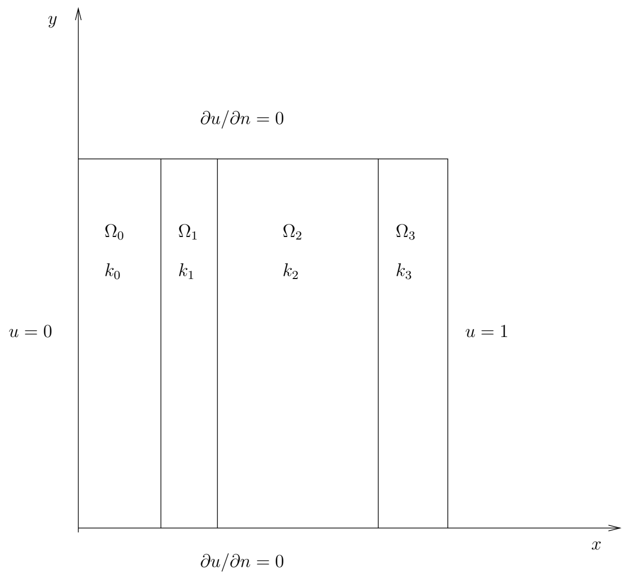


Fig. 7. Sketch of a multi-material problem.

The domain Ω is divided into s materials Ω_i, $i = 0, \ldots, s-1$, where

$$\Omega_i = \{(x_0, \ldots, x_{d-1} \,|\, L_i \leq x_0 < L_{i+1}\}$$

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 (94) is

$$u(x_0, \ldots, x_{d-1}) = \frac{\int_0^{x_0} (k(\tau))^{-1} d\tau}{\int_0^1 (k(\tau))^{-1} d\tau} \,.$$

For a piecewise constant $k(x_0)$ as explained, we get

$$u(x_0, \ldots, x_{d-1}) = \frac{(x_0 - L_i)k_i^{-1} + \sum_{j=0}^{i-1}(L_{j+1} - L_j)k_j^{-1}}{\sum_{j=0}^{s-1}(L_{j+1} - L_j)k_j^{-1}}, \quad L_i \leq x_0 \leq L_{i+1} \,. \tag{95}$$

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, any standard finite element method will reproduce this exact solution to machine precision, which is ideal for a test case.

7.2 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 Chapter 1.14. This results in an instance `mesh` representing the interval $[0,1]$ in 1D, the unit square in 2D, or the unit cube in 3D.

The subdomains Ω_i must be defined through subclasses of `SubDomain`. Would 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 Chapter 6.1 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 elements, and many of these will lie on the subdomain boundaries. All vertices `x` in a cell must be lead to a `True` return value from `inside` for the cell to be a part of a subdomain.

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` instance `subdomains`. The subdomain number of mesh entity number `e`, here cell `e`, is given by `subdomains.values()[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 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_exact(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_exact(xi) for xi in x])
    print u
    plot(x, u)
""" % (L, k))
f.close()
```

7.3 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 Chapter 6.2, a k that is constant in each element is suitable. We then follow the recipe from Chapter 6.2 to compute k:

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

# Vectorized calculation
help = numpy.asarray(subdomains.values(), dtype=numpy.int32)
k.vector()[:] = numpy.choose(help, k_values)
```

The essential boundary conditions are defined in the same way is in `Poisson2D_DN2.py` from Chapter 1.10 and therefore not repeated here. The variational problem is defined and solved in a standard manner,

```
v = TestFunction(V)
u = TrialFunction(V)
f = Constant(mesh, 0)
a = k*dot(grad(u), 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.values())):
    k_meshfunc.values()[i] = k_values[subdomains.values()[i]]

# Vectorized version
help = numpy.asarray(subdomains.values(), dtype=numpy.int32)
k_meshfunc.values()[:] = numpy.choose(help, k_values)

plot(k_meshfunc, title='k as mesh function')
```

The file `Poisson_layers.py` contains the complete code.

8 Miscellaneous Topics

8.1 Glossary

Below we explain some key terms used in this tutorial.

FEniCS: name of a software suite composed of many individual software components (see `fenics.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 Chapter 1.7).

Class (Python): a programming construction for creating objects containing a set of variables and functions. Most types of FEniCS objects are defined through the class concept.

Instance (Python): an object of a particular type, where the type is implemented as a class. For instance, `mesh = UnitInterval(10)` creates an instance of class `UnitInterval`, which is reached by the the name `mesh`. (Class `UnitInterval` is actually just an interface to a corresponding C++ class in the DOLFIN C++ library.)

Class method (Python): a function in a class, reached by dot notation: `instance_name.method_name`

`self` parameter (Python): required first parameter in class methods, representing a particular instance 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 `X`. 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`

8.2 Overview of Objects and Functions

Most objects in FEniCS have a explanation of the purpose and usuage that can be seen by using the general documentation command `pydoc` for Python objects. You can type

```
pydoc dolfin.X
```

to look up documentation of a Python class `X` from the DOLFIN library (`X` can be `UnitSquare`, `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.

`UnitSquare(nx, ny)`: generate mesh over the unit square $[0, 1] \times [0, 1]$ using `nx` divisions in x direction and `ny` divisions in y direction. Each of the `nx*ny` squares are divided into two cells of triangular shape.

`UnitInterval`, `UnitCube`, `UnitCircle`, `UnitSphere`, `Interval`, `Rectangle`, and `Box`: generate mesh over domains of simple geometric shape, see Chapter 5.

`FunctionSpace(mesh, element_type, degree)`: a function space defined over a mesh, with a given element type (e.g., `'CG'` or `'DG'`), with basis functions as polynomials of a specified degree.

`Expression(formula, V=W)`: a scalar- or vector-valued function, given as a mathematical expression `formula` (string) written in C++ syntax. The keyword argument `V` is used to specify a function space, here called `W`, sometimes needed when an `Expression` enters a variational form.

`Function(V)`: a scalar- or vector-valued finite element field in the function space `V`.

`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 Chapter 1.3) for telling whether a point `x` is inside the subdomain or not.

`MeshFunction`: tool for marking parts of the domain or the boundary. Used for variable coefficients ("material properties", see Chapter 6.1) or for boundary conditions (see Chapter 6.3).

`DirichletBC(V, value, where)`: specification of Dirichlet (essential) boundary conditions via a function space `V`, a function `value(x)` for computing the value of the condition at a point `x`, and a specification `where` of the boundary, either as a `SubDomain` subclass instance, a plain function, or as a `MeshFunction` instance. In the latter case, a 4th argument is provided to describe which subdomain number that describes the relevant boundary.

`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, bc)`: assemble the matrix and the right-hand side from a bilinear (`a`) and linear (`L`) form written with UFL syntax. The `bc` parameter holds one or more `DirichletBC` instances.

`VariationalProblem(a, L, bc)`: define and solve a variational problem, given a bilinear (`a`) and linear (`L`) form, written with UFL syntax, and one or more `DirichletBC` instances stored in `bc`. A 4th argument, `nonlinear=True`, can be given to define and solve nonlinear variational problems (see Chapter 2.4).

`solve(A, U, b)`: solve a linear system with `A` as coefficient matrix (`Matrix` instance), `U` as unknown (`Vector` instance), and `b` as right-hand side (`Vector` instance). Usually, `U` is replaced by `u.vector()`, where `u` is a `Function` instance 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`.

8.3 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 `fenics.org` website. Here are just some quick descriptions and recommendations by the author.

To make the installation of FEniCS as painless and reliable as possible, the reader is strongly recommended to use Ubuntu Linux. Any standard PC can easily be equipped with Ubuntu Linux, which may live side by side with either Windows or Mac OS X or another Linux installation. Basically, you download Ubuntu from `www.ubuntu.com/getubuntu/download`, burn the file on a CD, reboot the machine with the CD, and answer some usually straightforward questions (if necessary). Ubuntu is quite similar to both Windows 7 and Mac OS X, but to be efficient when doing science with FEniCS this author recommends to run programs in a terminal window and write them in a text editor like Emacs or Vim. You can employ integrated development environment such as Eclipse, but intensive FEniCS developers and users tend to find terminal windows and plain text editors more user friendly.

Instead of making it possible to boot your machine with the Linux Ubuntu operating system, you can run Ubuntu in a separate window in your existing operation system. On Mac, you can use the VirtualBox software available from `http://www.virtualbox.org` to run Ubuntu. On Windows, Wubi makes a tool that automatically installs Ubuntu on the machine. Just give a username and password for the Ubuntu installation, and Wubi performs the rest. You can also use VirtualBox on Windows machines.

Once the Ubuntu window is up and running, go to the `fenics.org` cite and paste in the five few lines that are needed to install what you need.

8.4 Books on the Finite Element Method

There are a large number of books on the finite element method. The books typically fall in either of two categories: the abstract mathematical version of the method and the engineering "structural analysis" formulation. FEniCS builds heavily on concepts in the abstract mathematical exposition. An easy-to-read book that provides a good general background for using FEniCS, is Gockenbach [7]. The book by Donea and Huerta [5] has a similar style, but aims at readers with interest in fluid flow problems. Hughes [9] 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 [1] 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.* [**?**] 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 [3], Braess and Dietrich [2], Ern and Guermond [6], Quarteroni and Valli [16], or Ciarlet [4].

8.5 Books on Python

Two very popular introductory books on Python are "Learning Python" by Lutz [13] and "Practical Python" by Hetland [8]. More advanced and comprehensive books include "Programming Python" by Lutz [12], and "Python Cookbook" [15] and "Python in a Nutshell" [14] by Martelli. The web page `http:://python.org/...` lists numerous additional books. Very few texts teach Python in a mathematical and numerical context, but the references [11,10,**?**] are exceptions.

8.6 User-Defined Functions

When defining a function in terms of a mathematical expression inside a string formula, e.g.,

```
myfunc = Expression('sin(x[0])*cos(x[1])', V=V)
```

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

Acknowledgments. The author is very thankful to Johan Hake, Anders Logg, Kent-Andre Mardal, and Kristian Valen-Sendstad for promptly answering all my questions about FEniCS functionality and for implementing all my requests. Professor Douglas Arnold is thanked for valuable feedback on this tutorial.

References

1. W. B. Bickford. *A First Course in the Finite Element Method.* Irwin, 2nd edition, 1994.
2. Dietrich Braess. *Finite elements.* Cambridge University Press, Cambridge, third edition, 2007. Theory, fast solvers, and applications in elasticity theory, Translated from the German by Larry L. Schumaker.
3. 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.
4. P. G. Ciarlet. *The Finite Element Method for Elliptic Problems.* SIAM, 2002.
5. J. Donea and A. Huerta. *Finite Element Methods for Flow Problems.* Wiley, 2003.
6. A. Ern and J.-L. Guermond. *Theory and Practice of Finite Elements.* Springer, 2004.
7. M. Gockenbach. *Understanding and Implementing the Finite Element Method.* SIAM, 2006.
8. M. L. Hetland. *Practical Python.* APress, 2002.
9. T. Hughes. *The Finite Element Method – Linear Static and Dynamic Finite Element Analysis.* Dover, 2000.

10. H. P. Langtangen. *A Primer on Scientific Programming with Python*. Texts in Computational Science and Engineering, vol 6. Springer, 2009.
11. H. P. Langtangen. *Python Scripting for Computational Science*. Springer, third edition, 2009.
12. M. Lutz. *Programming Python*. O'Reilly, third edition, 2006.
13. M. Lutz. *Learning Python*. O'Reilly, third edition, 2007.
14. A. Martelli. *Python in a Nutshell*. O'Reilly, second edition, 2006.
15. A. Martelli and D. Ascher. *Python Cookbook*. O'Reilly, second edition, 2005.
16. A. Quarteroni and A. Valli. *Numerical Approximation of Partial Differential Equations*. Springer Series in Computational Mathematics. Springer, 1994.
17. The python tutorial. `http://docs.python.org/tutorial/`.