GETTING STARTED WITH FENICS

DOUGLAS N. ARNOLD

1. A first program in FEniCS

1.1. **The boundary value problem.** As a first problem we consider the Neumann boundary value problem:

$$-\Delta u + u = f \text{ in } \Omega, \quad \frac{\partial u}{\partial n} = 0 \text{ on } \partial\Omega, \tag{1}$$

where Ω is a domain in $\mathbb{R}^2$. This problem is particularly simple in that it does not involve coefficient functions or essential boundary conditions, and the natural boundary conditions are homogeneous. We will get to those aspects soon enough, but its nice to ignore them the first time through. Notice that we do include a zero degree term in the PDE ($-\Delta u + u$ rather than just $-\Delta u$), since otherwise the Neumann problem would not be well-posed: any constant function would solve the problem for $f = 0$.

As always for finite elements, we start by casting the problem in weak form. Thus we seek a function u in a trial space V such that

$$b(u, v) = F(v) \quad \text{for all test functions } v \in V.$$

For the boundary value problem (1) the trial space $V = H^1(\Omega)$, and the bilinear form $b : V \times V \to \mathbb{R}$ and the linear functional $F : V \to \mathbb{R}$ are given by

$$b(u, v) = \int_\Omega (\operatorname{grad} u \cdot \operatorname{grad} v + uv)\, dx, \quad F(v) = \int_\Omega fv\, dx.$$

1.2. **A test problem.** To specify a test problem we use the *method of manufactured solutions*. This means that we start with the solution u and analytically compute the corresponding $f = -\Delta u + u$. We choose $u(x, y) = [(x^2 - 2x) \sin 2\pi y]^2$ on the unit square $\Omega = (0, 1) \times (0, 1)\Omega = (0, 1) \times (0, 1)$. This function satisfies the homogeneous Neumann boundary conditions.By direct calculation we find that

$$f(x, y) = -8[\pi(x^2 - 2x)]^2 \cos 4\pi y + (x^4 - 4x^3 - 8x^2 + 24x - 8) \sin^2 2\pi y. \tag{2}$$

1.3. **The discrete problem.** The finite element solution $u_h \in V_h$ is then determined from the Galerkin equations

$$b(u_h, v) = F(v), \quad v \in V_h,$$

where V_h is the finite element subspace we use to approximate V. We shall take V_h to be the Lagrange space of degree 2, which means that we choose a mesh (triangulation) of the domain Ω, and that V_h consists of all continuous functions on Ω which restrict to polynomials of degree at most 2 on each triangle of the triangulation.

Date: November 2017.

1.3.1. *Specifying the mesh.* Before turning to the FEniCS implementation, we discuss how the data of the problem is expressed in FEniCS. In this case, the only data we need to supply is the mesh (which implies the domain), and right-hand side function $f : \Omega \to \mathbb{R}$. There are numerous ways to define a mesh in FEniCS. It may be constructed externally to FEniCS by various different programs, then written to a file in an appropriate format and imported into FEniCS. Alternatively, there are several ways to construct a mesh inside FEniCS. We shall use the FEniCS function `UnitSquareMesh` which constructs a structured mesh of the unit square. Specifically, the code

```
mesh = UnitSquareMesh(10, 20)
```

produces the mesh obtained by dividing the unit square into 10×20 rectangles and dividing each of these into two triangles with the positively sloped diagonal, yielding this mesh of 400 elements:

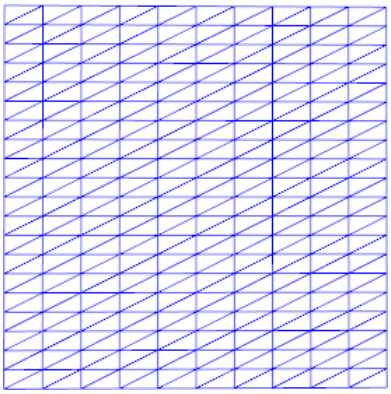

FIGURE 1. $(10 \times 20) \times 2$ mesh of the unit square.

1.3.2. *Specifying the right-hand side.* In order to define the functional $F(v) = \int_\Omega fv\,dx$, we have to specify the function f on Ω. The simplest way to do this is define a FEniCS `Expression`, which expresses the formula (2) for f in terms of the coordinates of the point. The coordinates are expressed as `x[0]` and `x[1]`, rather than x and y as used in (2). To the occasional surprise of FEniCS novices, the expressions are given in a string that will be parsed by a C++ compiler, and so are written in C++ rather than Python. Math expressions in the two languages are mostly identical to each other and to math expressions in other languages, but one exception is that exponentiation a^b must obtained through the `pow` function: `pow(a, b)`. The alternative syntax `a**b` which is permitted in Python is not permitted in C++, so cannot be used in an `Expression` string. Taking this into account, we get the following line of code to define f:

```
f = Expression('-8 * pow(pi*(pow(x[0], 2) - 2*x[0]), 2) * cos(4*pi*x[1]) \
     + (pow(x[0], 4) - 4*pow(x[0], 3) - 8*pow(x[0], 2) + 24*x[0] - 8) \
       * pow(sin(2*pi*x[1]), 2)', degree=4)
```

Notice the mandatory `degree=` argument to `Expression`. When FEniCS needs to compute an integral involving f, this helps it decide what quadrature rule to use.

1.4. **The FEniCS program.** We are now ready to present the full FEniCS implementation. As you can see, it is just 10 Python statements, and it follows the mathematical specification of the discrete problem closely.

```
from fenics import * # import FEniCS into Python
# Define the function f. The degree parameter is used by FEniCS to
# decide what quadrature rule to use when integrating expressions involving f.
f = Expression('-8 * pow(pi*(pow(x[0], 2) - 2*x[0]), 2) * cos(4*pi*x[1]) \
    + (pow(x[0], 4) - 4*pow(x[0], 3) - 8*pow(x[0], 2) + 24*x[0] - 8) \
    * pow(sin(2*pi*x[1]), 2)', degree=4)
# Define the mesh and the function space
mesh = UnitSquareMesh(10, 20)
Vh = FunctionSpace(mesh, 'Lagrange', 2)
# Define the bilinear form b(u, v) and the linear function F(v)
# for any trial function u in V_h, test function v in V_h
u = TrialFunction(Vh)
v = TestFunction(Vh)
b = (dot(grad(u), grad(v)) + u*v)*dx
F = f*v*dx
# Solve the Galerkin equations for the function u_h in V_h
uh = Function(Vh) # this declares u_h as finite element function in V_h
solve(b == F, uh)
```

1.5. **Processing the solution.** At this point `uh` is the finite element solution. We can use it in various ways.

1.5.1. *Plotting.* For example we can plot it using the the basic plotting capability built into FEniCS:

```
plot(uh) # use plot(uh, interactive=True) to manipulate the plot
```

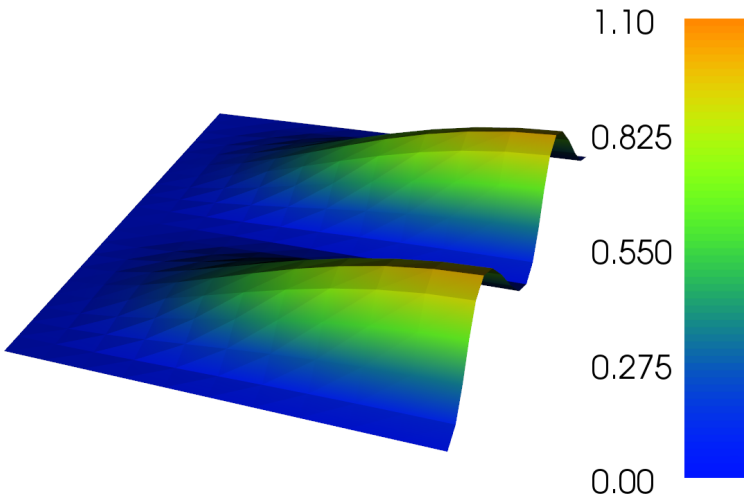

FIGURE 2. Finite element solution.

As indicated in the comment, one may add the `interactive=True` clause to the plot call to make the plot window responsive to the mouse and keyboard. Hit the "h" key to see the possibilities. The "w" key is useful to toggle visibility of the mesh. The "c" and "v" keys toggle numbering of the cells and vertices, respectively. Hitting the "p" key will save the plot to a PNG file. Use

the "q" key to end interactivity. We also mention that you can save a plot to a file without using interactivity. The statements

```
fig = plot(uh)
fig.write_png('myfile')
```

will write the plot to myfile.png.

While plotting from FEniCS is very convenient, it is also quite limited. For more advanced plotting, one can export the solution to a PVD file, which can be imported to the ParaView visualization program for plotting:

```
File("solution.pvd") << uh
```

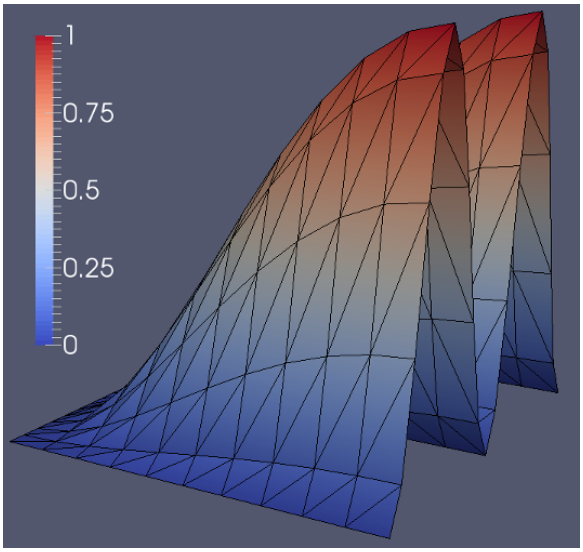

FIGURE 3. ParaView visualization of the solution.

1.5.2. *Evaluation of the solution and functionals of the solution.* We can simply evaluate the computed solution at a point. At the point $(3/4, 3/4)$, the exact solution is $u(3/4, 3/4) = 225/256 = 0.87890625$. Compare with evaluation of the computed solution:

```
value = uh(0.75, 0.75)
exact = 225./256.
error = (exact - value)/exact
print "value at (3/4, 3/4): {:10.8f}, error = {:6.4f}%".format(value, 100*error)
```

which returns

```
value at (3/4, 3/4): 0.87875670, error = 0.0170%
```

Note that requires locating the element containing the point $(3/4, 3/4)$, a relatively time-consuming process. So this approach is not ideal for evaluating the solution at many points. By contrast, the vector of DOF values of the solution, available as uh.vector().array()! returns the values of the solution at all the nodes at very little cost.

Since uh is a finite element function, we can act on it in various ways. For example, we can compute its integral over Ω using finite element assembly. Compare with the exact value which is $4/15 = 0.2666\ldots$:

```
uhint = assemble(uh*dx)
exact = 4./15.
error = (exact - uhint)/exact
print "integral of solution: {:10.8f}, error = {:6.4f}%".format(uhint, 100*error)
```

yielding

```
integral of solution: 0.2666667, error = 0.0000%
```

2. Digging a little deeper

2.1. **The import statement.** To use FEniCS we start our Python program with

```
from fenics import *
```

The name `dolfin` can be used instead of `fenics`; in this context they are interchangeable. Careful programmers may want to use `import fenics as fe` and then refer to FEniCS objects as, e.g., `fe.plot` instead of just `plot` to avoid name clashes.

2.2. **Expressions.** The next statement defines f as a FEniCS `Expression` by providing a C++ string. This becomes unwieldy for complex expressions, or if non-arithmetic logic is involved. There are other ways to define expressions. One of the most useful is by *subclassing* the `Expression` class. With this approach the `Expression` statement above would be replaced by this class definition:

```
class fex(Expression):
    def eval(self, values, x):
        t1 =-8*(pi*(x[0]**2 - 2*x[0]))**2 * cos(4*pi*x[1])
        t2 = (x[0]**4 - 4*x[0]**3 - 8*x[0]**2 + 24*x[0] - 8) * sin(2*pi*x[1])**2
        values[0] = t1 + t2
# instantiate the class, specifying degree parameter
f = fex(degree=4)
```

Note that in this approach the code defining the expression is Python, so we can use the `**` operator for exponentiation. We could also include complex code inside the evaluation routine `eval`, with loops, conditional statements, and so forth. The degree parameter must still be provided. It is used when FEniCS computes an integral involving the expression to select the quadrature rule to use (chosen so that if the expression were a polynomial of the indicated degree, then the integration rule would be exact).

We also mention that it is possible, and often very useful, to include parameters in the definition of an expression. For example, to define the function of one variable $f(x) = \exp(-ax)$ with parameter $a = 10$ we can use

```
f = Expression('exp(-a x[0])', degree=4, a=10.)
```

Later we can change the value of a to 20 with `f.a = 20.`, after which further use of the expression will evaluate $\exp(-20x)$.

2.3. **Defining a mesh.** To define a mesh of the unit square, we used `UnitSquareMesh(10, 20)`. This makes a mesh of 10×20 rectangles divided into triangles by the diagonal which increases to the right. To use the diagonal increasing to the left we would use `UnitSquareMesh(10, 20, 'left')`. To include both diagonals, use `UnitSquareMesh(10, 20, 'crossed')`. The analogous meshing routines in 1D and 3D are called `UnitIntervalMesh` and `UnitCubeMesh`. For meshing a rectangle in 2D which is not the unit square, there is an alternate routine which takes the bottom left and top right points of the rectangle as additional inputs:

```
mesh = RectangleMesh(Point(-5., -2.), Point(5., 2.), 10, 4, 'crossed')
```

This produces a mesh of a 10×4 rectangle centered at the origin. Similarly in 3 dimensions we could use

```
mesh = BoxMesh(Point(-5., -2., -3.), Point(5., 2., 3.), 10, 4, 20)
```

In 1 dimension, the syntax is a bit different:

```
mesh = IntervalMesh(100, 2., 3.)
```

gives a uniform mesh of 100 subintervals on the interval $[2, 3]$.

For meshing more complicated shapes, FEniCS includes a module called mshr, which implements constructive geometry. The mshr module must be explicitly loaded: `from mshr import *`. For example the mesh in Figure 4 is produced by

```
disc1 = Circle(Point(0., 0.), 3.)
disc2 = Circle(Point(1., 0.), 1.)
annulus = disc1 - disc2
mesh = generate_mesh(annulus, 20)
```

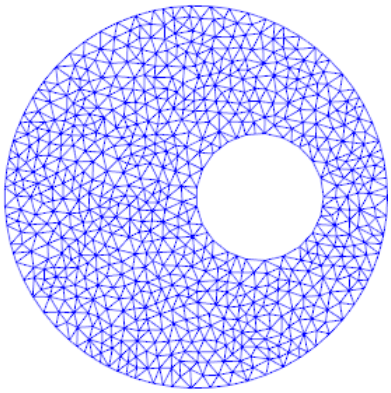

FIGURE 4. A mesh produced by mshr.

An alternative way to obtain meshes in FEniCS is to use an external package, such as Gmsh, to generate the mesh, and then import it into FEniCS.

2.4. **Defining a finite element space.** We defined the finite element space V_h with

```
FunctionSpace(mesh, 'Lagrange', 2)
```

This specifies the Lagrange family of finite elements of degree 2. The spatial dimension is 2 as FEniCS can infer from the mesh. Alternatively we can separate the specification of the finite element—Lagrange triangle of degree 2—and the specification of the space from the mesh and the finite element:

```
P2 = FiniteElement('Lagrange', triangle, 2)
Vh = FunctionSpace(mesh, P2)
```

As synonyms to `'Lagrange'` we can use `'CG'` (Continuous Galerkin). The degree, here 2, can be any positive integer. Another family of finite elements is the `'DG'` or Discontinuous Galerkin family, defined for any nonnegative degree. These elements have no interelement continuity and are mostly used to discretize $L^2(\Omega)$. Most of the other finite elements in FEniCS are for vector-valued function spaces.

2.5. **Defining the bilinear and linear forms.** The declarations `u = TrialFunction(Vh)` and `v = TestFunction(Vh)` defines two arguments that can be used as variables to define the bilinear form (`TrialFunction` just means the first of the two arguments to the bilinear form and `TestFunction` the second). The statement

```
b = (dot(grad(u), grad(v)) + u*v)*dx
F = f*v*dx
```

are written in FEniCS's Unified Form Language (UFL), and are fairly self-explanatory. Note that multiplication by the "measure" `dx` signals integration over Ω, i.e., the sum of the integrals over the elements of the mesh associated to V_h. There are also numerous other operators, beyond `grad` and `dot`, that can be used to define UFL expressions, and there is a way of defining a subset of the triangles and integrating only over the subset using a notation like `dx(1)`. Integration over the boundary (that is the edges contained in the boundary) is obtained with `ds`, and integration over the edges that are *not* contained in the boundary uses `dS`.

2.6. **Solving the boundary value problem.** The statement `uh = Function(Vh)` declares `uh` to be a function in the finite element space V_h. Such a function is characterized by a vector of length $\dim V_h$ giving the values of its degrees of freedom. One may obtain the vector as `uh.vector()`. This has the type of a FEniCS `Vector`, which has associated to it a (numpy) array with the actual values of the DOFs: `uh.vector().array()`. The array values are set to zero when we declare `uh`.

The most powerful statement in the program is the final solve:

```
solve(b == F, uh)
```

This triggers several actions in FEniCS. First the bilinear form b is assembled into a sparse matrix A and the linear functional F into a vector y. Next the linear algebra problem $Ax = y$ is solved for the vector x, which is stored as the DOFs of the function u_h.

These steps can be separated for more control:

```
A = assemble(b)
y = assemble(F)
uh = Function(Vh)
x = uh.vector()
solve(A, x, y)
```

The assembly command `A = assemble(b)` produces a FEniCS `Matrix` object. This encapsulates a sparse matrix implemented using some backend linear algebra package, most commonly PETSc, a highly tuned suite of datastructures and routines for working with sparse matrices. One could examine the matrix in various ways. E.g., `A.nnz()` will report the number of nonzeros stored for the matrix, and `A.array()` will convert A to a full matrix, stored as a numpy array. FEniCS (via PETSc) also offers many ways to control the algorithm used to solve the matrix equation.

3. More general problems

3.1. **Including coefficients.** Suppose we had the more general differential equation

$$-\operatorname{div}(a \operatorname{grad} u) + p \cdot \operatorname{grad} u + cuv = f$$

with coefficients a, p, and c, either variable or constant. We would then define each coefficient as a FEniCS `Expression`, just as we did for f. Since the coefficient p is vector-valued, its expression would be given by a pair of strings, like

```
p = Expression(('x[0] + 2*x[1]', '1.'), degree=1)
```

For constant coefficients, we can use `Constant` instead of `Expression`:

```
p = Constant((3., 1.))
```

In either case, we would define the bilinear form using UFL similarly to before:

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

We could also have a matrix-valued coefficient in place of a:

```
A = Expression((('2.', '.5*x[0]'), ('.5*x[0]', '4. + x[0]*x[1]')), degree=2)
b = (dot(dot(A, grad(u)), grad(v)) + dot(p, grad(u))*v + c*u*v)*dx
```

3.2. **Inhomogeneous Neumann boundary conditions.** The boundary value problem (1) has the simplest boundary conditions for finite element solution, namely homogeneous Neumann conditions. A very slight generalization is to take the inhomogeneous Neumann condition

$$\frac{\partial u}{\partial n} = g \text{ on } \partial\Omega. \tag{3}$$

The only change this entails to the weak formulation is the addition of a second term to the linear function $F(v)$:

$$F(v) = \int_\Omega fv\,dx + \int_{\partial\Omega} gv\,ds,$$

represented in FEniCS, of course, by

$$F = f * v * dx + g * v * ds,$$

where g is an `Expression` object.

3.3. **Dirichlet boundary conditions.** Next suppose that, instead of the Neumann boundary condition in (1) we wish to impose the Dirichlet boundary condition

$$u = g \text{ on } \partial\Omega.$$

This is an essential boundary condition, so that it should be imposed as a constraint on the space V rather than by changing the bilinear form b or the linear functional F. The weak formulation is now to find $u \in V = H^1(\Omega)$ such that $u = g$ on $\partial\Omega$ and

$$b(u, v) = \int_\Omega fv\,dx \quad \text{for all } v \in V \text{ such that } v = 0 \text{ on } \partial\Omega.$$

On the discrete level, u_h is restricted to elements of V_h which interpolate g on the boundary, i.e., any DOFs associated to vertices or edges on the boundary must give the same value when applied to u_h as when applied to g. We say that the DOFs of u_h on the boundary are constrained to g. In the case of Lagrange elements of degree 2, for instance, this means that u_h must equal g at the vertices on the boundary and the midpoints of edges on the boundary. Similarly the DOFs of the test function v on the boundary are constrained to be zero. FEniCS imposes these constraints this after assembling the stiffness matrix and load vector, by modifying the matrix and vector entries associated to boundary DOFs. To accomplish this, we need to specify to FEniCS which DOFs are to be constrained (in this case all the DOFs on the boundary), and what value they are to be constrained to (namely to g). This is accomplished by invoking the procedure

```
DirichletBC(Vh, g, bdry)
```

where the three arguments are the function space to constrain, the function to constrain to, and the portion of the domain where DOFs are to be constrained. The function g can be defined as an `Expression` just as for the Neumann boundary condition. When the Dirichlet boundary condition is to be imposed on the whole boundary of Ω, we can simply take specify `bdry` as `DomainBoundary()`. The relevant code is then:

```
g = Expression('...')
bc = DirichletBC(Vh, g, DomainBoundary())
uh = Function(Vh)
solve(b == F, uh, bc)
```

Note that the solve call now takes a third argument, specifying the essential boundary condition.

3.4. **Mixed boundary conditions.** Sometimes we have different boundary conditions on different, complementary, parts of the boundary. For example, let Γ_D denote the bottom side of the unit square, and Γ_N the union of the remaining three sides, and consider the solution of the boundary value problem

$$\Delta u = f \text{ in } \Omega, \quad u = g \text{ on } \Gamma_D, \quad \frac{\partial u}{\partial n} = k \text{ on } \Gamma_N.$$

The weak formulation then seeks $u \in V = H^1(\Omega)$ constrained to equal g on Γ_D such that

$$\int_\Omega \operatorname{grad} u \cdot \operatorname{grad} v \, dx = \int_\Omega f v \, dx + \int_{\partial\Omega} k v \, ds$$

for all $v \in V$ constrained to zero on Γ_D. (Note that the last integral is really taken only over Γ_N, since v is constrained to zero on Γ_D.) In this case we cannot use `DomainBoundary()` to specify the DOFs to constrain, but must define the subset Γ_D of the boundary. This is done by defining a Python function and passing it as the third argument of DirichletBC. We might choose to call the function `GammaD`. It takes two arguments, a point `x`, and a boolean `on_boundary`. When FEniCS needs to identify whether some point of the domain belongs to Γ_D it will call `GammaD(x, on_boundary)` with `x` set to the point (the array of its coordinates `x[0]` and `x[1]`), and `on_boundary` set to `True` or `False` according to whether the point x belongs to $\partial\Omega$ or not. Our function must return a boolean value true or false according to whether the point x belongs to Γ_D or not. Since Γ_D is the bottom edge of the square, the test is simply whether the second coordinate is equal to zero. However one should never test floating point numbers for exact equality, since they may suffer from small round off errors. For such purposes, FEniCS provides the routine `near(x[1], 0.)` which is true if `x[1]` is within round off error of zero. Thus we can specify Γ_D as follows:

```
def GammaD(x, on_boundary):
    return near(x[1], 0.) and on_boundary
```

which says that the points which should be constrained are those on the boundary of the domain for which the y component is within round-off error of 0.[1] From this point on, the code is similar to that discussed above.

```
f = Expression('...')
g = Expression('...')
k = Expression('...')
b = dot(grad(u), grad(v))*dx
F = f*v*dx + k*v*ds
bc = DirichletBC(Vh, g, GammaD)
uh = Function(Vh)
solve(b == F, uh, bc)
```

[1]Actually, here we could drop the `and on_boundary`, since if $y = 0$ the point must belong to $\partial\Omega$. But it is wise to include it. If we were describing the portion of the boundary where $y \geq .5$, for example, we would need to include the restriction to the boundary so that we do not constrain DOFs at points inside the domain.

Note that on the right hand side we have included $\int_{\partial\Omega} kv\,ds$. Actually, this integral is over Γ_N, the complement of Γ_D on the boundary, because it is only applied to v which are constrained to 0 on Γ_D.

Remark. People are often confused about what to do at the interface where the boundary conditions switch from Neumann to Dirichlet. In our example, this would consist of the two points $(0,0)$ and $(1,0)$, the end points of the bottom side Γ_D. Should the DOFs associated with these points be constrained, as for a Dirichlet BC or left unconstrained as for a Neumann BC? The answer is that *they should be constrained*. The constraint $u = g$ is a continuous condition: if it holds on a subset of $\partial\Omega$, it should hold on the closure of that set. In the case we have treated, where the Dirichlet boundary is the bottom edge of the square, this was easy to code: `near(x[0], 0.)` includes the end points. But consider the case where Γ_D is the union of the bottom, top, and right sides of the square. One might be tempted to define it by

```
def GammaD(x, on_boundary):
    return x[0] > 1.e-15 and on_boundary
```

using the condition $y > 1.e-15$ to include all points that are not on the left edge $y = 0$ (up to round-off error). But this is an error, because it excludes the points $(0,0)$ and $(0,1)$, which must be included in Γ_D. A correct definition of the boundary is

```
def GammaD(x, on_boundary):
    return (near(x[0], 0.) or near(x[1], 0.) or near(x[1], 1.)) and on_boundary
```

3.5. **Multiple Dirichlet boundary conditions.** It is possible to have multiple Dirichlet boundary conditions, say $u = g_1$ on the bottom side and $u = g_2$ on the top side. We would then define two different `DirichletBC`, each just as above, say `bc1` and `bc2`, and pass both of them to the solve routine:

```
solve(b == F, uh, [bc1, bc2])
```

3.6. **Robin boundary conditions.** Robin boundary conditions take the form $\partial u/\partial n + qu = g$, i.e., a linear combination of Dirichlet and Neumann boundary conditions. These can be treated much as for Neumann boundary conditions, in that they are natural, not essential. In the case of Robin boundary conditions, not only is the functional F modified, but also the bilinear form b.

3.7. **Periodic boundary conditions.** In the case of periodic boundary conditions, the boundary condition does not constrain individual points on the boundary, but relates the values at two different points, e.g., $u(0,y) = u(1,y)$ for periodicity in the x-direction on the unit square. We will not discuss this further except to say that this possibility is also available in FEniCS.