Shape Optimization Tutorial

By Stephan Schmidt

Exercise 1. The first exercise is to familiarise with Python and FEniCS. We can use the built-in FEniCS tutorial to implement a small solver for the Laplacian

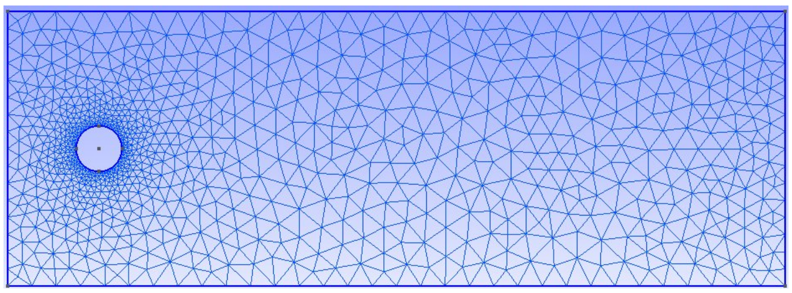
$$-\Delta u = f$$

on the unit square. Use for example $f = 1.0$ and Dirichlet boundary conditions $u = 0$ on all boundaries. An example implementation is below:

```
 # -*- coding: utf-8 -*-
from dolfin import *

#omega = UnitSquare(6,4)
omega = UnitSquareMesh(6,4)

W = FunctionSpace(omega, "CG", 1)
u = TrialFunction(W)
v = TestFunction(W)

f = Constant((1.0))
a = inner(nabla_grad(u),nabla_grad(v))*dx
L = inner(f,v)*dx

bc = DirichletBC(W, Constant((0.0)), "on_boundary")

u = Function(W)
solve (a == L, u, bc)

print "Done, plotting"
plot(u, interactive=True)
```

Exercise 2. We now modify this code to operate on a domain, such as the one given by “stationary_circle.xml”. That file contains a domain like this. The file “stationary_circle_facet_region.xml” contains information about which facet lies on which boundary. In our situation here, we have

- Inflow: 1
- Outflow: 2
- Outer Walls: 3



- Obstacle: 4

Modify your code to solve the Laplacian on this domain. Use Dirichlet boundary conditions $u = 0$ everywhere, except on boundary 1, where we want $u = 1$. You will find these new commands useful:

```
omega = Mesh('stationary_circle.xml')
mf=MeshFunction('size_t', omega,
                      'stationary_circle_facet_region.xml')
```

and

```
bc1 = DirichletBC(W, 0.0, mf, 3)
bc2 = DirichletBC(W, 0.0, mf, 4)
#inflow
bc3 = DirichletBC(W, 1.0, mf, 1)
#obstacle
bc4 = DirichletBC(W, 0.0, mf, 2)
bc = [bc1, bc2, bc3, bc4]
```

Exercise 3. Modify your code such that it solves two Laplace problems simultaneously. Find $w = (u, p)$ in $W = V \times Q$ such that

$$\int_\Omega (f_1, v)\, dx + (f_2, q)\, dx = \int_\Omega (\nabla u, \nabla v) + (\nabla p, \nabla q)\, dx$$

for all $(v, q) \in V \times Q$. Use Dirichlet boundary conditions of your liking for each. Products of functions spaces are created in FEniCS using

```
V = VectorFunctionSpace(omega, "CG", 1)
Q = FunctionSpace(omega, "CG", 1)
W = V*Q
```

Dirichlet conditions for the respective component are designated by

```
 bc1 = DirichletBC(W.sub(0), 0.0, mf, 3)
```

Any function $w \in W$ can be decomposed into $w = (u, p)$ with the command

```
(u,p) = w.split(True)
```

Exercise 4. Change your program to solve Stokes's equation

$$\int_\Omega (\nabla u, \nabla v)\ dx + p \operatorname{div} v\ dx - (\operatorname{div} u) q\ dx$$

using the notation as above. Note that now u is a vector valued function. Thus, your Dirichlet boundary values will need to be vectors and you will need to disable the Dirichlet boundary condition on the outflow boundary marker 2. For LBB-stability, chose u to be 2nd order ansatz functions and p first order.

Be aware that FEniCS distinguishes between "Function" and "Functions", so that

```
V = VectorFunctionSpace(omega, "CG", 2)
```

and

```
(u, p) = TrialFunctions(W)
```

Finally, evaluate the objective function

$$J = \frac{1}{2} \int_\Omega (\nabla u, \nabla u)\ dx$$

The FEniCS command to integrate is "assemble".

Exercise 5. Clean up your code and create a subroutine that takes the domain Ω and the boundary marker "mf" and returns the objective function value J and the two PDE states u and p. A function in Python is defined by

```
def FunctionName("arguments"):
    CODE
    return "values"
```

Note that in Python, the level of indentation indicates where the declaration of a function or loop starts and ends.

Exercise 6. We now start with the first components specifically needed for shape optimization. Unfortunately, FEniCS leaks some pre-made functions that would simplify this task, so we have to code custom functions.

This is a very technical task that cannot be solved as elegant as the previous ones, because it requires some lower level manipulations of the code. It might help to have a look at the demo solutions sooner. Because of the low level index multiplications, you may need (depending on the FEniCS version you are using) to set the following parameters at the beginning of your main file to make the demo routine work:

```
dolfin.parameters.reorder_dofs_serial = False
dolfin.parameters.allow_extrapolation = True
```

- Within integrals, FEniCS understands the local normal *n* only on boundaries. With respect to elements in the domain, FEniCS knows a face normal, but these are not useable in integrals and either way, FEniCS has no functionality for normals at vertices, which we need because we want to move points and thus we need the normal at points on the boundary.

  Some useful functions

  ```
  ver = Vertex(mesh, i)
  n = Point(0.0, 0.0)
  div = 0.0
  for fac in entities(ver, mesh.geometry().dim()-1):
      f = Facet(mesh, fac.index())
      if f.exterior()==True:
      ...
  ```

  This gets the vertex with index *i* from the mesh, afterwards generates an empty vector and then loops over all facets around the vertex *i*. You can use the div-counter to average the facet normals into a vertex normal.

- As a second step, we now want to generate a vector field *N* over Ω that contains the vertex normals at the respective boundaries and some values (maybe zeros?) in the domain. The idea is to iterate over vertex indices, check if they are on a boundary and then call the above function to calculate the vertex normal. Afterwards, we store the respective result into the right component of *N*. Commands you might find useful:

  ```
  gamma = BoundaryMesh(omega, "exterior")
  mapa = gamma.entity_map(0)
  ```

extracts the outer boundary mesh. Also “mapa” contains the respective index map, that is “mapa(i)” will give us the volume ID of the *i*-th boundary node. You can use a command like this

```
solve(inner(normal_field[0], x)*dx
  + inner(normal_field[1], y)*dx
  - inner(normal_x, x)*dx
  - inner(normal_y, y)*dx == 0, normal_field)
```

to turn two numbers into a vector field.

Exercise 7. Modify your routine that computes a vertex normal everywhere (or the demo “compute_normal_field”) from the previous step to now compute the shape gradient in normal direction on the boundary and smoothly extended into the domain. Instead of using just the vertex normal as in the previous step, you can now compute

$$(\nabla u, \nabla u) \cdot n.$$

Next, create a routine “update_domain”, that takes a small number and your deformation field and moves every mesh node accordingly. You can access all *x*-components of all vertices in the mesh simultaneously using the command

```
omega.coordinates()[:,0] += rho*u_x.vector().array()
```

You are now ready to put a loop around these steps and perform a simple steepest descent! Note that because the gradient is positive everywhere, the shape will shrink. For an actual physical problem, a volume constraint would be necessary. Otherwise, no flow obstacle at all has of course the least drag.

Exercise 8. Add in a volume constraint. Although the proper way of doing this would be to use a constrained optimization algorithm, we could also just step into vertex normal direction (the direction of the shape gradient of the volume) until we have recovered the original volume. The volume of the mesh can for example be calculating by summation of the volume of every cell.

```
for c in cells(omega):
  volume += c.volume()
```