Lab 6: Optimization

March 3, 2017

Introduction

In this lab you will implement Newton's method for root-finding to solve the problem of computing the square root. In the second problem you will implement a general solver for a stationary non-linear partial differential equation. The last problem in this lab is a 2D source identification problem. This type of problem is important for many applications such as in cleaning groundwater pollution and in problems related to the electrical activity of the brain.

1 Newton's method to compute square root

We will use Newton's method to compute square roots, $x = \sqrt{a}$ for $a > 0$. This is equivalent to solving $x^2 = a$, or $x^2 - a = 0$. An interesting fact is that a method to compute square roots that is essentially equivalent to Newton's method was the Babylonian method which dates back to the ancient Babylonians (more than 3000 years ago). A Babylonian clay tablet from circa 1000 B.C. gives the results of the square root of 2 up to 6 decimal figures!!
Recall that Newton's method for solving $f(x) = 0$ is given by iterating,

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

until convergence. In our case $f(x) = x^2 - a$.

Exercise: Verify that using Newton's method on the square root problem results in the sequence of iterates,

$$x_{n+1} = \frac{1}{2}(x_n + \frac{a}{x_n})$$

We start with an initial guess of $x = 0.5$ and iterate. You will see that after only five iterations, Newton's method already has achieved 12 digits of precision.

```
import numpy as np

a = 2; x = 0.5; iters = 10;
for i in np.arange(iters):
   x = 0.5*(x + a/x)
   print x
```

Define the relative error $e = \frac{|x_n - \sqrt{a}|}{\sqrt{a}}$. Start with an initial guess of 30 and print the first 10 iterations of the relative error. The code for doing it is the following,

```
import numpy as np

a = 2; x = 30; iters = 10;
for i in np.arange(iters):
   x = 0.5*(x + a/x)
   relerr = np.abs(x-np.sqrt(a))/np.sqrt(a)
   print relerr
```

Exercise: By looking at the relative errors, can you deduce the rate of convergence of Newton's method? (Note that the last two errors displayed are the same because we have reached machine precision.)

2 Bratu Problem

In this section, you will implement a general solver for a stationary non-linear partial differential equation. We will use the Bratu problem as our PDE. The Bratu problem comes from a simplification of the solid fuel ignition model in combustion theory. It also has an application in the theory of the expansion of the universe. It can be described as,

$$\begin{aligned} -\nabla^2 u &= \lambda \exp(u) + f(x) && \text{in } \Omega \\ u &= 0 && \text{on } \partial\Omega \end{aligned}$$

Consider the Bratu problem in 1D with homogeneous Dirichlet boundary conditions on the interval $[0, 1]$ and $f(x) = \pi^2 \sin(\pi x) - \lambda \exp(\sin(\pi x))$. This solution has an exact solution,

$$u_{exact} = \sin(\pi x)$$

Exercise: Verify that this is an exact solution to Bratu's problem.

Newton's method for solving $F(u) = 0$, is analogously to the case described in problem 1,

$$F(u_n) + \frac{\partial F}{\partial u}\Big|_{u_n} (u_{n+1} - u_n) = 0$$

In our case, $F(u_n) = -\nabla^2 u - \lambda \exp(u) - f(x)$. Let $w = (u_{n+1} - u_n)$. Then, Let $w = (u_{n+1} - u_n)$. Then, the equation can be written as,

$$F(u_n) + \frac{\partial F}{\partial u}\Big|_{u_n} w = 0$$

After a few calculus operations regarding Gateaux derivatives, the linearized equation for Newton becomes,

$$\begin{aligned} -\nabla^2 u_n - \lambda \exp(u_n) - \nabla^2 w - \lambda \exp(u_n) w - f(x) = 0 && \text{in } \Omega \\ w = -u_n && \text{on } \partial\Omega \end{aligned}$$

Note that now the equation is a partial differential equation and the weak form would be bilinear in terms of w and v since u_n is known already as the value of the previous step (and is thus held constant). Once we have calculated w, we update $u_{n+1} = u_n + w$ as

usual, and iterate again.

If you are not familiar with Gateaux derivatives, it is not a problem. FEniCS has a function "derivative" that is able to calculate the Gateaux derivatives for us. The syntax for computing $d_w F(u_n)$ is very easy once $F(u_n)$ is defined.

```
J = derivative(F,un,w)
```

We start with an initial guess of $u_0 = -5$ everywhere, and $\lambda = 2$ on a mesh with $N = 40$.

```
from dolfin import*
import numpy as np
%matplotlib inline

lamda = 2.; N = 40
mesh = UnitIntervalMesh(N)
V = FunctionSpace(mesh, "CG", 1)
u0 = Constant(-5); un = project(u0,V)

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

uex = Expression("sin(pi*x[0])")
uex = interpolate(uex,V)
f = Expression(("pow(pi,2)*sin(pi*x[0]) - lamda*exp(sin(pi*x[0]))"), lamda =
    lamda, degree = 1)
iters = 2
err = np.zeros((iters,1))
w = TrialFunction(V)
for i in np.arange(iters):

   fm = (dot(grad(v), grad(un)) - lamda*exp(un)*v) * dx - f*v*dx
   J = derivative(fm,un,w)
   a = J
   L = -fm

   bc = DirichletBC(V, -un, lambda x, on_boundary: on_boundary)

   A,b = assemble_system(a,L,bcs = bc)
   A_mat = as_backend_type(A).mat();
   from scipy.sparse import csr_matrix
   As = csr_matrix(A_mat.getValuesCSR()[::-1], shape = A_mat.size)

   x = np.linalg.solve(As.toarray(),b.array())
   un.vector()[:] += x

import matplotlib.pyplot as plt
plt.plot(mesh.coordinates(),un.vector().array())
plt.plot(mesh.coordinates(),uex.vector().array())
plt.legend(["Numerical Solution", "Analytical Solution"])
plt.xlabel('x')
plt.ylabel('u')
```

```
plt.show()
```

Since this system is now linear, we can also use conjugate gradient to solve the linear system.

Exercise: Change the number of iterations until you observe convergence (visual examination is okay).
Exercise: Use the code you had written for conjugate gradient in the first assignment to solve this system, instead of the $x =$ np.linalg.solve() command.

3 Optimization (Location of a source)

In this part of the lab, we will solve an optimization problem to locate a Gaussian source. More specifically, the problem we are interested in is the 2D Poisson equation $-\nabla^2 u = f$ on $[-1, 1]$, the same problem as in Lab1. Assume we have a Gaussian source, i.e.

$$f = 2\exp\left(-\left(\frac{x - s_0}{2\sigma}\right)^2 - \left(\frac{y - s_1}{2\sigma}\right)^2\right)$$

Below are examples of three gaussians with different centers (x_0, y_0).

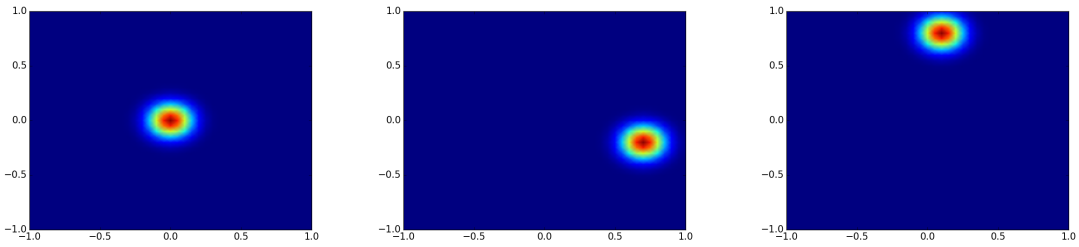

Figure 1: Example Gaussians with varying (s_0, s_1)

Suppose we do not know what s_0 or s_1 are and we are trying to estimate it using data on the values of u and the positions along the grid where u is measured. We are given a dataset "data_mat.mat" with the locations and values of u at the locations shown in Figure 2.
To load the data, we use the following:

```
import scipy.io as sio
A = sio.loadmat('data_mat')
uvalues = A['values']
pos = A['posx']
```

We first need to define the function that we are trying to optimize. In particular, we want to minimize the error between our solution u at the locations of interest with the predicted (x_0, y_0) and the u-data we are given, i.e. uvalues. In mathematical notation, this is equivalent to,

$$\underset{x_0, y_0}{\text{minimize}} \ \|u(\text{pos}) - \text{uvalues}\|_2$$

The first step is to define our function,

$$f(s) = \|u(\text{pos}) - \text{uvalues}\|_2$$

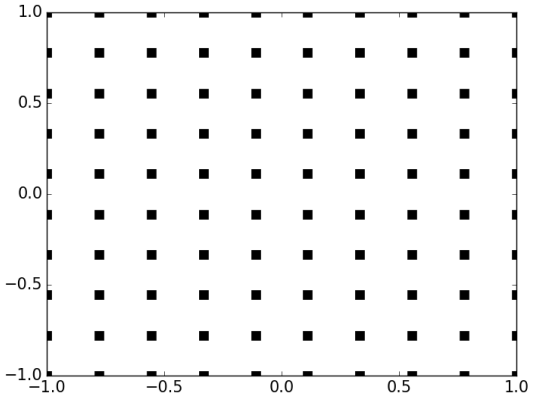

Figure 2: Locations at which we are given the values of u.

```
def f(s, pos, uvalues,V):

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

   #Parameters
   s0 = s[0]
   s1 = s[1]
   sigma = 0.1
   f = Expression('2.*exp(-0.5*(pow((x[0] - s0)/sigma, 2)) '
           '     - 0.5*(pow((x[1] - s1)/sigma, 2)))',
                    sigma=sigma,s0 = s0, s1 = s1)
   a = inner(grad(u), grad(v))*dx
   fproj = interpolate(f,V)
   L = f*v*dx
   # Assemble system
   A,b = assemble_system(a,L,bcs)
   A_mat = as_backend_type(A).mat();
   from scipy.sparse import csr_matrix
   As = csr_matrix(A_mat.getValuesCSR()[::-1], shape = A_mat.size)
   u = np.linalg.solve(As.toarray(),b.array())

   return np.linalg.norm(u[pos] - uvalues)
```

We first use a derivative free method, scipy has Nelder-Mead implemented, we start with the initial guess of $[0, 0]$.

```
   import scipy.optimize as optimize
   import scipy.io as sio
   A = sio.loadmat('data_mat')
   uvalues = A['values']
   pos = A['posx']

   N = 40
   mesh = UnitSquareMesh(N, N)
```

```
mesh.coordinates()[:,0] = mesh.coordinates()[:,0]*2.0-1.0
mesh.coordinates()[:,1] = mesh.coordinates()[:,1]*2.0-1.0

def boundary(x):
   return x[0] < -1 + DOLFIN_EPS or x[1]< -1 + DOLFIN_EPS or x[0]>1 -
       DOLFIN_EPS or x[1]>1 - DOLFIN_EPS

V = FunctionSpace(mesh, "CG", 1)
bcs = [DirichletBC(V,Constant(0.0), boundary)]
# Define variational problem
result = optimize.minimize(f, [0.,0.], args = (pos,uvalues,V),
    method='Nelder-Mead', tol=1e-6)
```

Once the code has finished, just print result and you should get the values of the source it has found. You should get something close to $0.5, 0.5$, which is the correct value of the Gaussian source.