Homework 5

Computational Math

Author:
Mikola Lysenko

May 14, 2010

1

a For the test functions, choose u, v from the space of continuous functions supported on Ω; ie supp $u \subseteq \Omega$. Now for any solution u with test function v we must have:

$$\int_{\Omega} -u_{xx}(x,y)v(x,y) - u_{yy}(x,y)v(x,y)d\Omega = \int_{\Omega} f(x,y)v(x,y)d\Omega$$

Starting on the left hand side, we work term by term:

$$\begin{aligned}
\int_{-1}^{1}\int_{-1}^{1} -u_{xx}(x,y)v(x,y)dxdy &= \int_{-1}^{1}\left(-u_x(x,y)v(x,y)|_{-1}^{1} + \int_{-1}^{1} u_x(x,y)v_x(x,y)dx\right)dy \\
&= \int_{\Omega} u_x(x,y)v_x(x,y)d\Omega \\
&= p_1(u,v)
\end{aligned}$$

By symmetry:

$$p_2(u,v) = \int_{\Omega} u_{yy}vd\Omega = \int_{\Omega} u_y(x,y)v_y(x,y)d\Omega$$

For the right hand side, we just get:

$$b(v) = \int_{\Omega} f(x,y)v(x,y)d\Omega$$

And so the weak form of the variational problem is:

$$p_1(u,v) + p_2(u,v) = b(v)$$

b Let $(x_1,y_1),(x_2,y_2),(x_3,y_3),(x_4,y_4)$ be the nodes of the element, oriented clockwise. We now solve for $\alpha_1,\alpha_2,\alpha_3,\alpha_4$ for the node (x_1,y_1). Plugging in values, we get the following linear system:

$$\begin{aligned}
\alpha_1 + \alpha_2 x_1 + \alpha_3 y_1 + \alpha_4 x_1 y_1 &= 1 \\
\alpha_1 + \alpha_2 x_2 + \alpha_3 y_2 + \alpha_4 x_1 y_2 &= 0 \\
\alpha_1 + \alpha_2 x_3 + \alpha_3 y_3 + \alpha_4 x_1 y_3 &= 0 \\
\alpha_1 + \alpha_2 x_4 + \alpha_3 y_4 + \alpha_4 x_4 y_4 &= 0
\end{aligned}$$

For the sake of simplicity, we rewrite the system in matrix form:

$$M\alpha = c$$

Where α is the vector of coefficients. Since c is a basis vector, the values for α at various nodes are just the corresponding rows of M^{-1}.

Now to construct the matrix equations for this system, we first consider the weak form from part a on a per element basis. Thus let φ^i, φ^j be two test functions on a quad element where

$$\varphi^i(x) = \alpha_1^i + \alpha_2^i x + \alpha_3^i y + \alpha_4^i xy$$

And:

$$\varphi_x^i(x) = \alpha_2^i + \alpha_4^i y$$

To integrate $p_1(\varphi^i, \varphi^j)$, we split the integral into two triangles, indexed by $\Delta(1,2,3)$ and $\Delta(1,3,4)$, then integrate in barycentric coordinates. We do this for the first triangle $\Delta(1,2,3)$ now. Let:

$$J = \begin{pmatrix} x_2 - x_1 & y_2 - y_1 \\ x_3 - x_1 & y_3 - y_1 \end{pmatrix}$$

And define the affine transformation:

$$\mathcal{T}(\lambda_1, \lambda_2) = J \begin{pmatrix} \lambda_1 \\ \lambda_2 \end{pmatrix} + \begin{pmatrix} x_1 \\ y_1 \end{pmatrix}$$

And so we get the following:

$$\begin{aligned}
\int\limits_{\Delta(1,2,3)} \varphi_x^i(x,y)\varphi_x^j(x,y)dxdy &= \frac{1}{\det J}\int\limits_0^1\int\limits_0^{1-\lambda_2} \varphi_x^i(\mathcal{T}(\lambda_1,\lambda_2))\varphi_x^j(\mathcal{T}(\lambda_1,\lambda_2)))d\lambda_1 d\lambda_2 \\
&= \frac{1}{\det J}\int\limits_0^1\int\limits_0^{1-\lambda_2} \alpha_2^i\alpha_2^j + (\alpha_4^i\alpha_2^j + \alpha_2^i\alpha_4^j)(J_{2,1}\lambda_1 + J_{2,2}\lambda_2 + y_1) \\
&\quad + \alpha_4^i\alpha_4^j(J_{2,1}\lambda_1 + J_{2,2}\lambda_2 + y_1)^2 d\lambda_1 d\lambda_2
\end{aligned}$$

To simplify the expression, make the following substitutions:

$$\begin{aligned}
Q_0 &= \alpha_2^i\alpha_2^j \\
Q_1 &= \alpha_2^i\alpha_4^j + \alpha_4^i\alpha_2^j \\
Q_2 &= \alpha_4^i\alpha_4^j
\end{aligned}$$

And so we get the following quantity:

$$= \frac{1}{2\det J}\left(Q_0 + y_1\left(Q_1 + y_1 Q_2\right) + \frac{J_{2,1}+J_{2,2}}{3}\left(Q_1 + \left(2y_1 + \frac{J_{2,1}+J_{2,2}}{2}\right)Q_2\right) - \frac{J_{2,1}J_{2,2}Q_2}{6}\right)$$

We shall call this quantity T_1^1, where the upper index denotes the triangle and the lower index denotes the p_1 component of the Laplacian, thus we get:

$$A(\varphi^i, \varphi^j) = p_1(\varphi^i, \varphi^j) + p_2(\varphi^i, \varphi^j) = \sum T_1^1 + T_2^1 + T_1^2 + T_2^2$$

And so the final matrix is just formed by summing over all such values. Computing f can be done approximately by sampling at the nodal values.

c Here is the solver I wrote to implement the described method (in Python):

```
from numpy import *
from scipy import *
from scipy.linalg import *
from scipy.sparse import *
from scipy.linsolve import *

class QuadElement:
    def __init__(self, ni, nx, ny):
        self.ni = ni
        self.nx = [nx[k] for k in ni]
        self.ny = [ny[k] for k in ni]
        M = matrix([ [ 1, nx[k], ny[k], nx[k] * ny[k]] for k in ni ])
        self.alpha = inv(transpose(M))
    def laplacian(self):
        res = []
        for i in range(len(self.ni)):
            for j in range(len(self.ni)):
                ali = array(self.alpha[i,1:3]).flatten()
                alj = array(self.alpha[j,1:3]).flatten()
                ahi = self.alpha[i,3]
                ahj = self.alpha[j,3]
                Q0 = ali * alj
                Q1 = ali * ahj + alj * ahi
                Q2 = ahi * ahj
                S = 0.
                for k in range(2,4):
                    J = matrix([ [self.nx[p] - self.nx[0], self.ny[p] - self.ny[0]] for p in
                        range(k-1,k+1) ])
                    X = array([J[1,0] + J[1,1], J[0,0] + J[0,1]])
                    Y = array([self.ny[0], self.nx[0]])
                    T = Q0 + Y * (Q1 + Y * Q2) + X / 3. * (Q1 + (2 * Y + X / 2.) * Q2) - Q2 *
                        array([J[1,0]*J[1,1], J[0,0]*J[0,1]]) / 12.
                    S -= sum(T) / (2. * det(J))
                res.append(((self.ni[i], self.ni[j]), S))
        return res

def gen_regular_quad_mesh(grid):
    D, R, C = grid.shape
    def get_index(ix, iy):
        if(ix < 0 or ix >= R or iy < 0 or iy >= C):
            return -1
        idx = ix + R * iy
        return idx
    nx = grid[0,:,:].flatten()
    ny = grid[1,:,:].flatten()
    mesh = []
    for ix in range(R-1):
        for iy in range(C-1):
            mesh.append(QuadElement(     \
                [get_index(ix,    iy),    \
                 get_index(ix+1, iy),    \
                 get_index(ix+1, iy+1), \
                 get_index(ix,    iy+1)],\
                nx, ny))
    return mesh, nx, ny, R*C, R, C, get_index

def fe_solve(mesh, nx, ny, f, boundary, bvals):
    M = len(nx)
    A = dok_matrix((M, M))
    b = zeros((M))
    for e in mesh:
        for ((i,j), v) in e.laplacian():
            if(boundary[i]):
                continue
            A[i,j] += v
    for i in range(M):
        if(boundary[i]):
            b[i] = bvals[i]
            A[i,i] = 1
        else:
            b[i] = f(nx[i], ny[i])
    return spsolve(A, b), A, b
```

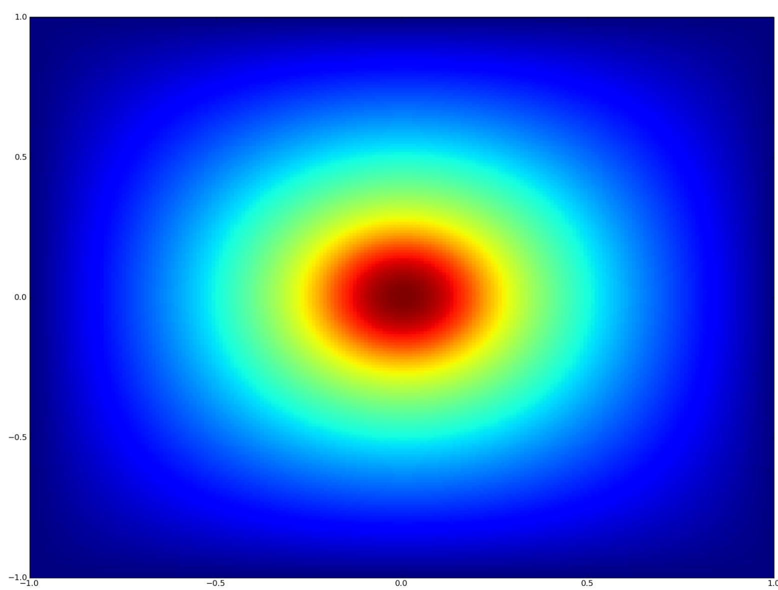
solver1.py

And here is the script I wrote to test it on the prescribed problem:

```
from numpy import *
from pylab import *
from solver1 import *

#Do the mesh generation
hx = 0.01
hy = 0.01
grid  = mgrid[-1:1+hx:hx,-1:1+hy:hy]
mesh, nx, ny, M, R, C, get_idx = gen_regular_quad_mesh(grid)

#Compute boundary conditions
boundary = zeros((M), 'bool')
for ix in range(C):
    boundary[get_idx(ix,0)] = True
    boundary[get_idx(ix,R-1)] = True
for iy in range(R):
    boundary[get_idx(0,iy)] = True
    boundary[get_idx(C-1,iy)] = True
bvals = zeros((M))

#Construct f
def f(x,y):
    if(sqrt(x*x + y*y) < 0.2):
        return 100.
    return 1.

#Solve problem
u, A, b = fe_solve(mesh, nx, ny, f, boundary, bvals)

#Display result
X = nx.reshape(R, C)
Y = ny.reshape(R, C)
U = u.reshape(R, C)
pcolor(X, Y, U)
savefig("prob1_result.png")
show()
```

prob1.py

This is a heatmap plot of the resulting distribution:



2 Since I am using python, I couldn't get the meshgen code to work. Instead, I downloaded a generic wrapper for QHull and wrote my own mesher based on Delaunay triangulation with a few refinements. First, the condition that points stay within the set is rewritten as a single unilateral constraint, which can be achieved using R-functions. Similarly, the points near the boundary can be clamped turning this constraint into an optimization problem. The starting points for the mesh are taken from a uniform grid, and then jittered a bit. As a post process, any elements with a sufficiently small initial angle are collapsed. The removal of these bad quality elements is iterated until all elements have an acceptable quality threshold. Here is the code I wrote:

```
from numpy import *
from numpy.random import uniform
from scipy import *
from math import atan2
from scipy.linalg import *
from scipy.sparse import *
from scipy.linsolve import *
import scipy.optimize as opt
from delaunay import Triangulation
from solver2 import TriElement
from pylab import *
import sympy as sp
import sympy.abc as abc

'''
Generates a uniform triangular grid of sample points
'''
def gen_tri_grid(xmin, xmax, xstep):
    g = mgrid[xmin[0]:xmax[0]+xstep[0]:xstep[0],xmin[1]:xmax[1]+xstep[1]:xstep[1]]
    nx = g[0,:,:].flatten()
    ny = g[1,:,:].flatten()
    b0 = matrix([[cos(pi/3.)], [sin(pi/3.)]])
    b1 = matrix([[1.], [0]])
    return transpose(array(b0 * nx + b1 * ny))

'''
Generates lambdas for the semianalytic constraint
'''
def make_lambdas(f_expr, X, Y):
    #Construct base lambda
    fn = sp.lambdify((X,Y), f_expr)
    f  = lambda x : fn(x[0], x[1])

    #Comput lagrange multiplier form/derivatives
    f_sq = f_expr**2
    f_dx = sp.diff(f_sq, X)
    f_dy = sp.diff(f_sq, Y)

    #Construct numerical functions
    fn_sq = sp.lambdify((X,Y), f_sq)
    fn_dx = sp.lambdify((X,Y), f_dx)
    fn_dy = sp.lambdify((X,Y), f_dy)

    fv      = lambda x : fn_sq(x[0], x[1])
    grad_fv = lambda x : -array([ \
        fn_dx(x[0], x[1]), \
        fn_dy(x[0], x[1])])

    return f, fv, grad_fv

'''
Pushes the point p to the boundary
'''
def push_to_boundary(p, fv, grad_fv):
    return opt.fmin_ncg(fv, p, grad_fv, maxiter=400, disp=0)

'''
Filters the points to lie within a semianalytic set defined by the function f. Points 'near' the
    boundary but exterior to f are pushed exactly onto the boundary by nonlinear optimization.
'''
def filter_points(pts, f, fv, grad_fv, cutoff):
    pz = zip(map(f, pts), pts)
    samples = [ x[1] for x in pz if x[0] <= 0 ]
    bstart = len(samples)
    for p in [ x[1] for x in pz if (x[0] > 0 and x[0] <= cutoff) ]:
```

```
        v = push_to_boundary(p, fv, grad_fv)
        if(abs(f(v)) < 1e-8):
            samples.append(v)
    return array(samples)

'''
Generates a mesh from a set of base sample points using delaunay triangulation.  Removes edges
    which cross outside boundary.  Low quality elements near the boundary are also killed
'''
def make_tri_mesh(pts, f, cutoff = 0.):
    M = pts.shape[0]
    dtri = Triangulation(pts, 2)
    mesh = []
    for t in dtri.get_elements_indices():
        edges = [ array([pts[t[k]], pts[t[(k+1)%3]]]) for k in range(3) ]
        #Check for edges that cross outside region
        good = True
        for e in edges:
            m = .5 * (e[0] + e[1])
            if(f(m) >= cutoff):
                good = False
                break
        if(not good):
            continue

        #Add element
        mesh.append(TriElement(t, pts[:,0], pts[:,1]))
    return mesh

'''
Removes bad elements
'''
def refine_mesh(pts, mesh, f, fv, grad_fv, qcutoff):
    M = pts.shape[0]
    bad_pt = zeros((M))
    npts = []
    for t in mesh:
        if(any([ bad_pt[k] for k in t.ni ])):
            continue
        if(t.quality() < qcutoff):
            #Find smallest edge
            edges = [ [t.ni[k], t.ni[(k+1)%3]] for k in range(3) ]
            edge_len = [ norm(pts[e[0]] - pts[e[1]]) for e in edges ]
            emin = min(zip(edge_len, edges))
            #collapse edge
            e = emin[1]
            bad_pt[e[0]] = 1
            bad_pt[e[1]] = 1
            v = .5 * (pts[e[0]] + pts[e[1]])
            if(any([ abs(f(pts[k])) < 1e-8 for k in e ])):
                v = push_to_boundary(v, fv, grad_fv)
            npts.append(v)
    npts.extend([ p for (i,p) in enumerate(pts) if not bad_pt[i] ])
    return array(npts), sum(bad_pt)

'''
Draw a wire frame of a triangle mesh
'''
def wire_plot_mesh(mesh, pts, color='#000000'):
    for poly in mesh:
        v = array([ pts[k] for k in poly.ni ])
        edges = [ array([ v[k], v[k-1] ]) for k in range(3) ]
        for e in edges:
            plot(e[:,0], e[:,1], color=color)
```

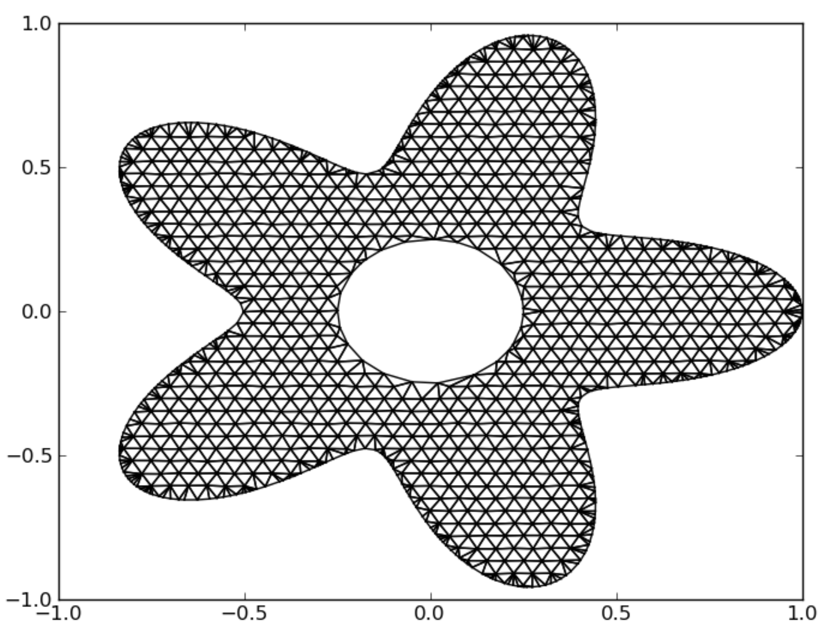
trimesh.py

```
from trimesh import *
import sympy as sp
from sympy.abc import X, Y

#Construct semianalytic variety f using R functions to minimize singularities
def F(X, Y):
    rs = X * X + Y * Y
    r = sp.sqrt(rs)
    c = X / (r + 0.001)
```

```
    s = Y / (r + 0.001)
    fo = r - .75 - .25 * (5 * s**4 * c - 10 * c**3 * s**2 + c**5)
    fi = 2. * ((.25)**2 - rs)
    return fo + fi + sp.sqrt(fo**2 + fi**2)

#Make lambdas
f, fv, grad_fv = make_lambdas(F(X, Y), X, Y)

#Make grid
pts = gen_tri_grid([-3,-3],[3,3],[0.05,0.05])
pts += uniform(-0.002, 0.002, pts.shape) #jitter points a bit
pts = filter_points(pts, f, fv, grad_fv, 1.2)

#Generate mesh
nbad_elements = 1
while(nbad_elements > 0):
    mesh = make_tri_mesh(pts, f, 0.0052)
    pts, nbad_elements = refine_mesh(pts, mesh, f, fv, grad_fv, pi/13.)

#Mark all boundary points
boundary = array([ abs(f(v)) < 1e-8 for v in pts ])

#Save the mesh/data points to file
import pickle
fout = open("mesh.pkl", "wb")
pickle.dump(mesh, fout)
pickle.dump(pts, fout)
pickle.dump(boundary, fout)
fout.close()

#Plot result
wire_plot_mesh(mesh, pts)
savefig('prob2_result.png')
show()
```

prob2.py

And here is the resulting mesh:



It is not as good as the meshgen results, but given time limitations it is the best I could come up with.

3 To set up the weak form of the variational problem, it is sufficient to derive the weak form for the Laplacian operator. Again, the space of test functions are smooth bumps supported on the domain. As before, the basis coefficients per element are described a linear system. Without loss of generality, consider a single element with vertices $(x_1, y_1), (x_2, y_2), (x_3, y_3)$. Let the basis function for vertex i of this element be given by the following affine function:

$$\varphi^i(x, y) = \alpha_1^i + \alpha_2^i x + \alpha_3^i y$$

Subject to the constraint $\varphi^i(x_j, y_j) = \delta_{i,j}$. As before, this gives a linear system:

$$\begin{aligned}
\alpha_1^i + \alpha_2^i x_1 + \alpha_3^i y_1 &= \delta_{i,1} \\
\alpha_1^i + \alpha_2^i x_2 + \alpha_3^i y_2 &= \delta_{i,2} \\
\alpha_1^i + \alpha_2^i x_3 + \alpha_3^i y_3 &= \delta_{i,3}
\end{aligned}$$

Which we rewrite as a matrix equation, $M\alpha_i = e_i$, and so the coefficients for the i^{th} element are just the i^{th} row of M^{-1}.

Now to integrate over the elements, use Barycentric coordinates again. Define:

$$J = \begin{pmatrix} x_2 - x_1 & y_2 - y_1 \\ x_3 - x_1 & y_3 - y_1 \end{pmatrix}$$

$$\mathcal{T}(\lambda_1, \lambda_2) = J \begin{pmatrix} \lambda_1 \\ \lambda_2 \end{pmatrix} + \begin{pmatrix} x_1 \\ y_1 \end{pmatrix}$$

Though in this case, we must sum over only one triangle. To determine the x-component of the Laplacian, we do the following:

$$\begin{aligned}
\int_{\Delta} \varphi_x^i(x, y)\varphi_x^j(x, y)dxdy &= \frac{1}{\det J} \int_0^1 \int_0^{1-\lambda_2} \varphi_x^i(\mathcal{T}(\lambda_1, \lambda_2))\varphi_x^j(\mathcal{T}(\lambda_1, \lambda_2))d\lambda_1 d\lambda_2 \\
&= \frac{1}{\det J} \int_0^1 \int_0^{1-\lambda_2} \alpha_2^i \alpha_2^j d\lambda_1 d\lambda_2 \\
&= \frac{\alpha_2^i \alpha_2^j}{2 \det J}
\end{aligned}$$

And thus the weights for the total Laplacian for element i to j are:

$$p_2(\varphi^i, \varphi^j) = \frac{\alpha_2^i \alpha_2^j + \alpha_3^i \alpha_3^j}{2 \det J}$$

Using this method, I implemented the following modified element type, which reuses my solver from part 1:

```
from numpy import *
from scipy import *
from math import acos
from scipy.linalg import *
from scipy.sparse import *
from scipy.linsolve import *
from pylab import *

class TriElement:
    def __init__(self, ni, nx, ny):
        self.ni = ni
        self.nx = [nx[k] for k in ni]
        self.ny = [ny[k] for k in ni]
        M = matrix([ [ 1, nx[k], ny[k] ] for k in ni ])
        self.alpha = inv(transpose(M))
```

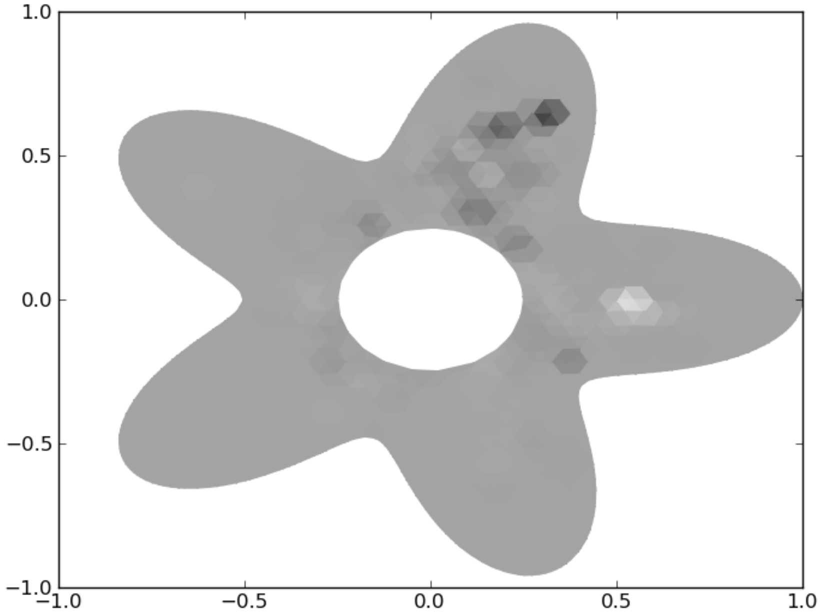
```
    def quality(self):
        theta = []
        for k in range(3):
            d = [array([self.nx[(k+p)%3] - self.nx[k], \
                        self.ny[(k+p)%3] - self.ny[k]]) for p in range(1,3)]
            d = [v / norm(v) for v in d]
            theta.append(acos(sum(d[0] * d[1])))
        return min(theta)


    def laplacian(self):
        res = []
        for i in range(len(self.ni)):
            for j in range(len(self.ni)):
                ali = array(self.alpha[i,1:3]).flatten()
                alj = array(self.alpha[j,1:3]).flatten()
                J = matrix([ [self.nx[p] - self.nx[0], self.ny[p] - self.ny[0]] for p in
                    range(1,3) ])
                S = -sum(ali * alj) / (2. * det(J))
                print S
                res.append(((self.ni[i], self.ni[j]), S))
        return res


'''
Draws the mesh (not very good right now)
'''
def plot_mesh(mesh, U):
    umin = min(U)
    umax = max(U)
    s = 1. / (umax - umin)
    def get_color(v):
        return str((v - umin) * s)
    for ele in mesh:
        fill(ele.nx, ele.ny, color=get_color(sum([ U[k] for k in ele.ni ])/3.) )
```

solver2.py

```
from numpy import *
from math import atan2
from pylab import *
from solver1 import *
from solver2 import *
import pickle

#Load mesh
fin = open("mesh.pkl", "rb")
mesh     = pickle.load(fin)
pts      = pickle.load(fin)
boundary = pickle.load(fin)
fin.close()

#Compute boundary conditions
def BC(x):
    r = norm(x)
    theta = atan2(x[1], x[0])
    if(r <= .251):
        return sin(5. * theta)
    return 0
bvals = array([ BC(v) for v in pts ])

#Set up RHS
def f(x,y):
    return 0

#Solve system
U, A, b = fe_solve(mesh, pts[:,0], pts[:,1], f, boundary, bvals)

#Solve the system and plot results
plot_mesh(mesh, U)
savefig("prob3_result.png")
show()
```

prob3.py

Here are some results:



There appear to be some issues with the set up for the problem. I would have this working except, I spent too much time trying to get the mesh to draw properly using pylab. I think that given a few more hours I could get this working.

4