MEK 4300
Mandatory Assignment

Andreas Slyngstad

13. mars 2015

Assignment i

In this assignment we are to verify the analytical solutions of the Hagen-Poiseuille Flow for different ducts, presented in White[7]. The ducts to be computed, are an Equilateral triangle, an Ellipse and an Eccentric annulus. I have used the gmsh software to make the different meshes to be calculated.

Triangle

White presents the analythical solution of Hagen-Poiseuille flow over the Equilateral triangle with sides a to be:

$$u(y,z) = \frac{\frac{-dp}{dx}}{2 * \sqrt{3}a\mu}\Big(z - \frac{1}{2}a\sqrt{3}\Big)(3y^2 - z^2)$$

I have constructed a Fenics program with some defined functions that takes the arguments:

- mesh point size
- a number n that says how many times we want to divide the size of the mesh point
- highest Lagrange order to be calculated.

Which are used to make the different meshes and controls the number of iterations we do. For example by choosing n to be 3 and the highest Lagrange to be 2, my program will make 2 meshes, and calculate both meshes 2 times.
I have used n = 4, highest order of Lagrange to be 3, and mesh point size to be 0.4 in my iteration.

```
h=9.17E-02 Error=3.33E-04 r=1.27
h=9.17E-02 Error=6.74E-06 r=2.37
h=9.17E-02 Error=2.48E-08 r=-2.13
h=5.90E-02 Error=1.72E-04 r=1.50
h=5.90E-02 Error=2.30E-06 r=2.44
h=5.90E-02 Error=2.17E-08 r=0.31
h=4.15E-02 Error=9.89E-05 r=1.58
h=4.15E-02 Error=9.38E-07 r=2.56
h=4.15E-02 Error=2.26E-08 r=-0.12
```

Even though I used the Listing(4) to comupte the error, from my understanding the logarithmic difference seems rather jumpy and not very stable.
The error on the other hand seems to be right, due to the descending value of mesh sizes and errors, with the increasing Lagrange degree. So it seems that the proposed analythical solution is correct.

Ellipse

For the Ellipse, the suggested analythical solution is presented as:

$$u(y,z) = \frac{1}{2\mu}\Big(-\frac{dp}{dx}\Big)\frac{a^2b^2}{a^2+b^2}\Big(1-\frac{y^2}{b^2}-\frac{z^2}{b^2}\Big)$$

For the Ellipse, my program is changed slightly by changing the makeel function to make an ellipse, rather than a triangle:

```
def makeel(filen, cl):
        file = open(filen, "w")

        file.write("cl = %s;\n" % cl)
        file.write("Point(2) = {0, 0, 0, cl};\n\
        Point(3) = {-2, 0, 0, cl};\n\
        Point(4) = {2, 0, 0, cl};\n\
        Point(5) = {0, 1, 0, cl};\n\
        Point(6) = {0, -1, 0, cl};\n\
        Ellipse(1) = {6, 2, 4, 4};\n\
        Ellipse(2) = {4, 2, 5, 5};\n\
        Ellipse(3) = {5, 2, 3, 3};\n\
        Ellipse(4) = {3, 2, 6, 6};\n\
        Line Loop(6) = {3, 4, 1, 2};\n\
        Plane Surface(6) = {6};")
        file.close
```

Again I have used n = 4, highest order of Lagrange to be 3, and mesh point size to be 0.4 in my iteration. The output yields:

```
h=8.15E-02 Error=7.48E-05 r=2.08
h=8.15E-02 Error=3.40E-05 r=2.03
h=8.15E-02 Error=3.35E-05 r=2.02
h=4.98E-02 Error=3.28E-05 r=1.67
h=4.98E-02 Error=1.54E-05 r=1.61
h=4.98E-02 Error=1.53E-05 r=1.60
h=3.85E-02 Error=1.95E-05 r=2.03
h=3.85E-02 Error=8.73E-06 r=2.20
h=3.85E-02 Error=8.68E-06 r=2.19
```

In this run, I'm more satisfied with the print due to the logarithmic difference is more or less stable. At the same time we observe the error decreases with the mesh size, and the increasing degree of the Lagrange.

```
from dolfin import *
import os
import sys
from math import *

#Setting constants, runs smoother in C++
mu = 0.2
dpdx = -1.0
a = 1.0

#Function to make triangle with a certain mesh size
def makeel(filen, cl):
        file = open(filen, "w")

        file.write("cl = %s;\n" % cl)
        file.write("Point(1) = {0, 0, 0, cl};\n\
                                Point(2) = {-0.5, Sqrt(0.75), 0, cl};\n\
                                Point(3) = {0.5, Sqrt(0.75), 0, cl};\n\
                                Line(1) = {1, 3};\n\
                                Line(2) = {3, 2};\n\
                                Line(3) = {2, 1};\n\
                                Line Loop(4) = {3, 1, 2};\n\
                                Plane Surface(5) = {4};")
        file.close

#Analythical solution

u_e = Expression('-dpdx/(2.0*sqrt(3.0)*a*mu)*(x[1]-0.5*a*sqrt\
(3.0))*(3.0*x[0]*x[0]-x[1]*x[1])', mu=mu, a=a, dpdx=dpdx)

#Defining lists
error = [] #Error
h = []  #
cls = [] #Size of mesh points

#IMPORTANT n is NUMBER OF MESHES MADE, FIXED TO THE FUNCTIONS AND LOOPS

def divide(cl, n):
        #Reducing the mesh size cl=0.5 with a half, n times
        #Remember that the first calculation is NOT cl, but cl
        #divided in two

        for i in range(1,n+1):
                calc = cl/(2.0*i)
                cls.append(calc)
                inp = str(i)
                filename = 'triangle_mesh%s' % inp
                makeel('%s.geo' % filename, calc)
                os.system('gmsh -2 %s.geo' % filename)
                os.system('dolfin-convert %s.msh %s.xml' % (filename, filename))


def main(deg, mesh):
        #Define variational problems
        mesh = mesh
        V = FunctionSpace(mesh, 'CG', deg)
```

```
        u = TrialFunction(V)
        v = TestFunction(V)
        F = inner(grad(u),grad(v))*dx + 1.0/mu*dpdx*v*dx
        bc = DirichletBC(V, Constant(0), DomainBoundary())

        #Solving the problem
        u_ = Function(V)
        solve(lhs(F) == rhs(F), u_, bc)

        u_er = project(u_e, V)

        #Calculating Error
        u_er = project(u_e, V)
        err = errornorm(u_er, u_, degree_rise=0)
        error.append(err)



def calc(n, highd):
        for k in range(1,n+1):
                mesh = Mesh('triangle_mesh%d.xml' % k)  #Currrent mesh size to ca
                hi = mesh.hmin()
                h.append(hi)
                for j in range(1, highd+1):
                        main(j, mesh)


def compare(highp, n):
        for j in range(n-1):
                for i in range(highp):
                                r = log( error[i+highp*(j+1)] / error[i+highp*(j)]

                                print "h=%2.2E Error=%2.2E r=%.2f" % (h[j+1], err

n=4                             #Number of times to divide cl in half
cl = 0.4                    #Mesh point size
deg = 3                         #Highest Lagrange degree to be calculated

divide(cl, n)       #Read as divide cl, n times
calc(n, deg)        #Read as calcualte n meshes, with lagrange degree up to deg
compare(deg, n)         #Read as compare the results with lagrange degree from 1
```

Assigment ii

In this assigment we are to solve to dimentionless equations, one for plane stagnation flow and one for Axisymmetric stagnation flow.

Stagnation flow

We obtain the dimensionless equation (146):

$$F''' + FF'' + 1 - F'^2 = 0$$

By defining $H = F'$ and applying this on the equation we end up with a solveable system.

$$H - F' + H'' + FH' + 1 - H^2 = 0$$

To variational form

$$\int_\Omega (Hv_f - F'v_f + H''v_n + FH'v_n + v_n - H^2 v_n)dx$$

The surface integrals goes to zero, hence:

$$\int_\Omega Hv_f dx - \int_\Omega \nabla F v_f dx - \int_\Omega \nabla h \cdot \nabla v_h dx + \int_\Omega F \nabla H v_n dx + \int_\Omega v_n dx - \int_\Omega H^2 v_n dx = 0$$

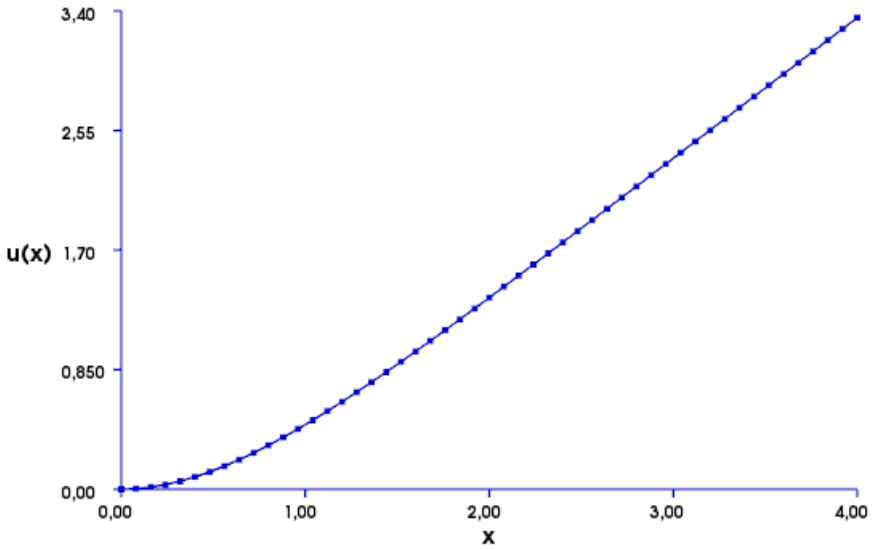
We solve the non-linear equation with Newton and Piccard iterations. The Result har show below for the

Axisymmetric stagnation flow

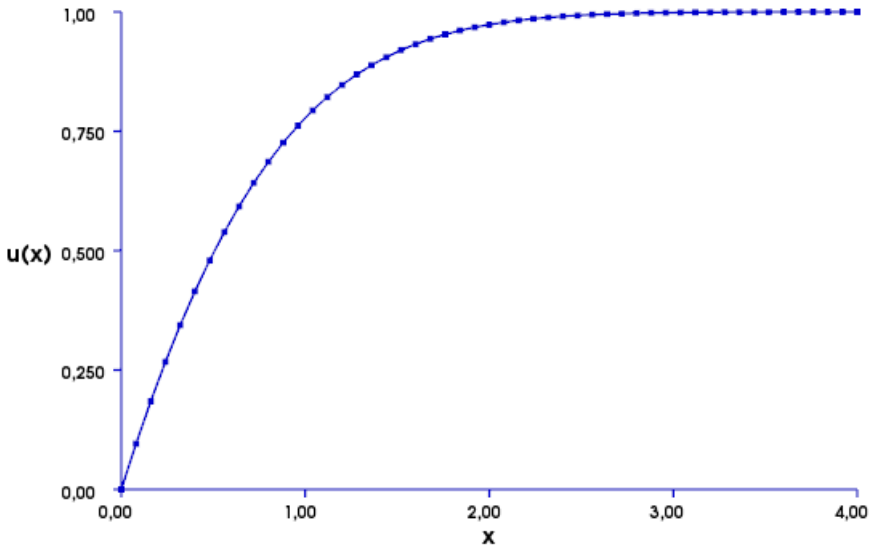
In this situation we obtain the almoust same dimensionless equation, except for the second term which is doubled. Hence we should not get to different plots from this situation, as the graphs show us below. I will not add the code to this sub-exercise due to it's almoust identicall to the stagnation flow.

$$F''' + 2FF'' + 1 - F'^2 = 0$$

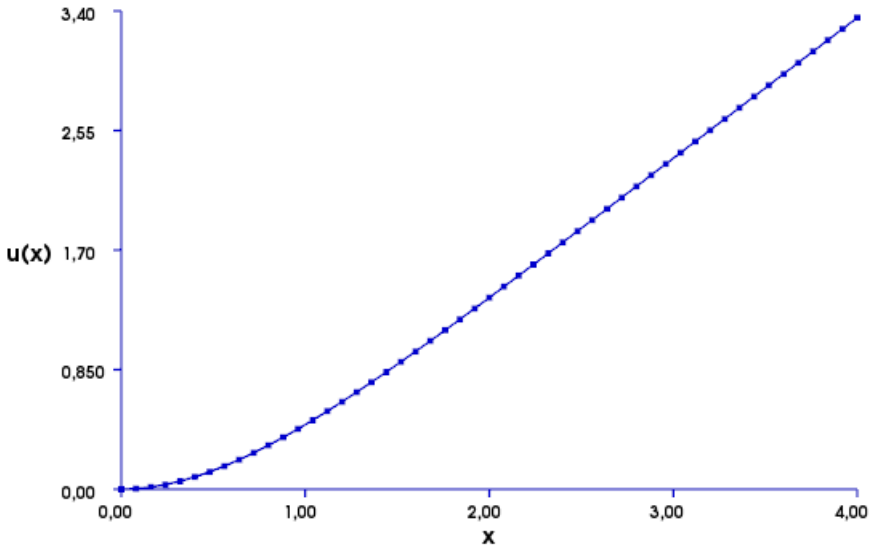
Equation 146 Newton F



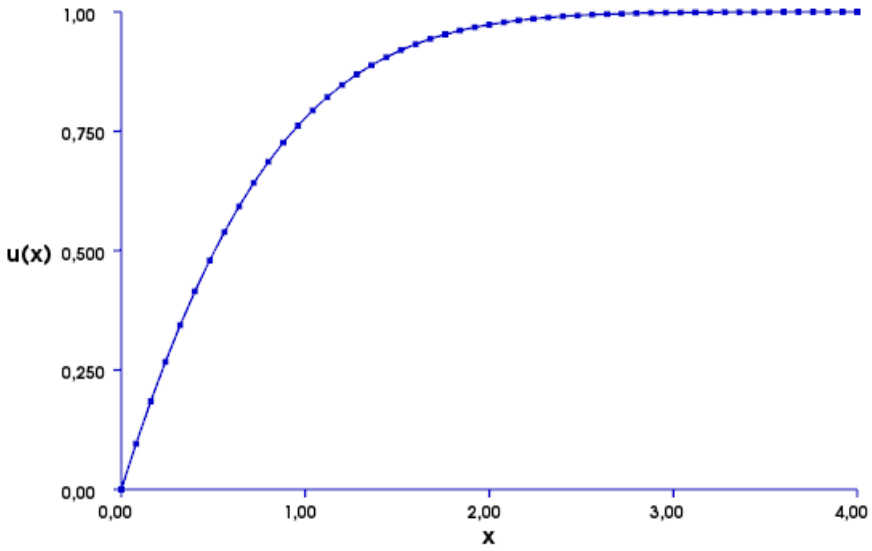
Equation 146 Newtin H



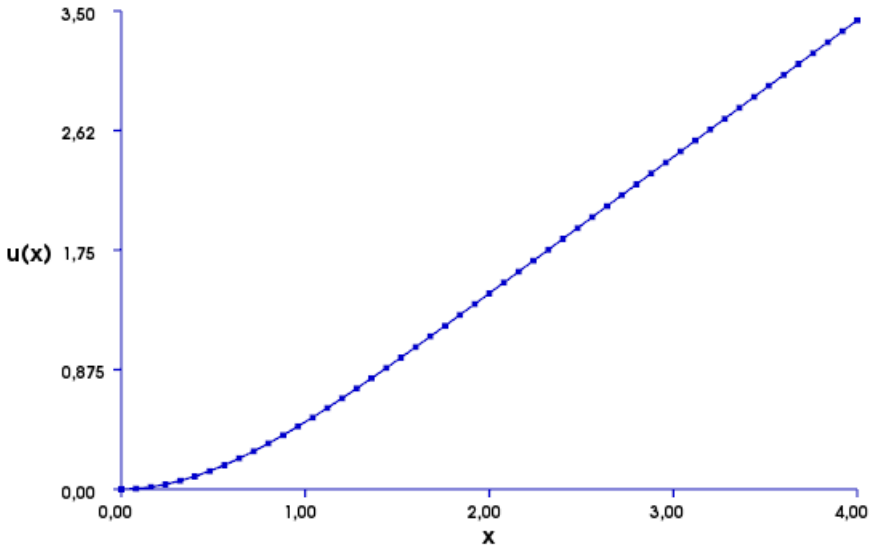
Equation 146 Piccard F



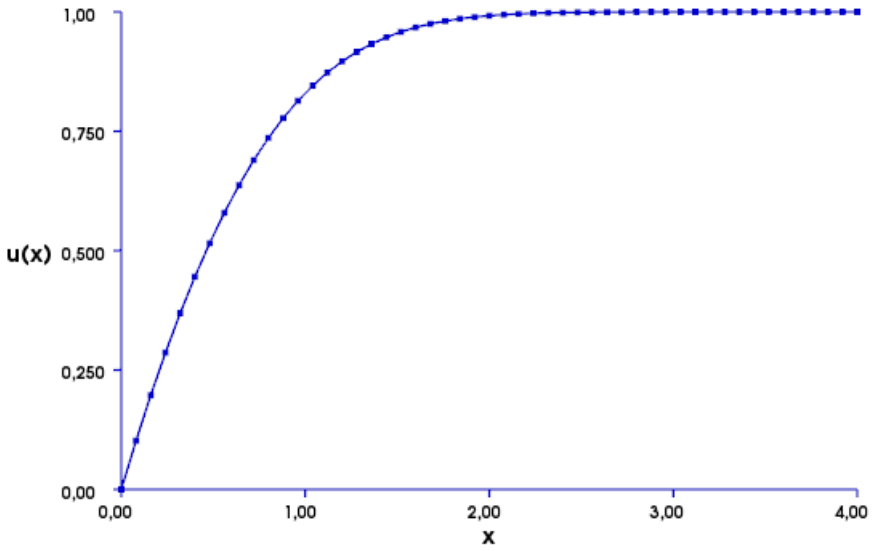
Equation 146 Piccard H



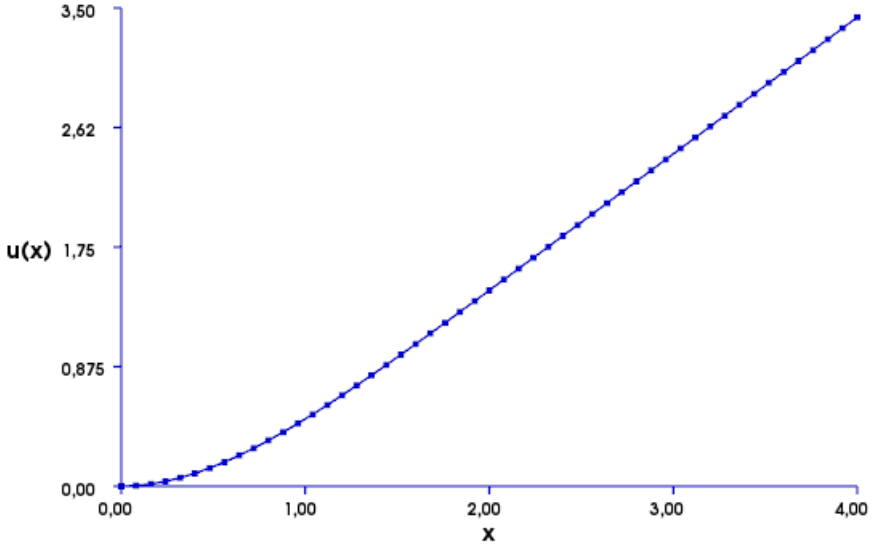
Equation 148 Newton F



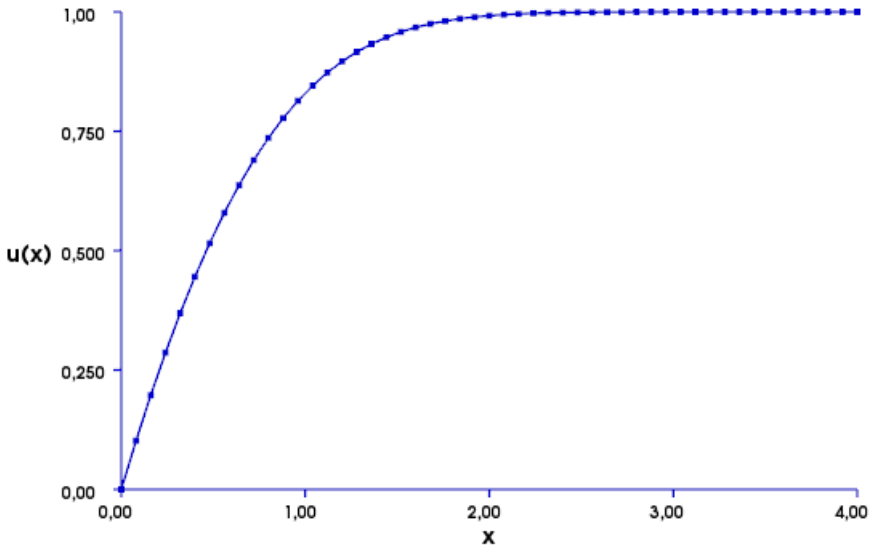
Equation 148 Newton H



Equation 148 Piccard F



Equation 148 Newton H



```
from dolfin import *
L = 4
mesh = IntervalMesh(50, 0, L)
V = FunctionSpace(mesh, 'CG', 1)
VV = V*V #Making mixed function space
vf, vh = TestFunctions(VV) #different testfunc for both equations
fh = TrialFunction(VV) #To the picard
f, h = split(fh)

def newton():
        #Mind Constant((0,0)) due to mixed function space
        bc0 = DirichletBC(VV, Constant((0,0)), "std::abs(x[0]) < 1e-10")

        #Fordi jeg ikke tar med sub sa gjelder bc for begge
        bc1 = DirichletBC(VV.sub(1), Constant(1), "std::abs(x[0]-%d) < 1e-10" % L
        #Mind sub(1) cause this only yields for F' (H)

        #First "guess" of the solution
        fh_ = interpolate(Expression(("x[0]","x[0]")), VV)
        f_, h_ = split(fh_)

        F = h_*vf*dx -f_.dx(0)*vf*dx - inner(grad(h_),grad(vh))*dx + f_*h_.dx(0)*

        solve(F==0, fh_, [bc0, bc1])


        viz = plot(f_)
        viz.write_png('146newton1')
        viz = plot(h_)
        viz.write_png('146newton2')
        interactive()

def picard():
        #Mind Constant((0,0)) due to mixed function space
        bc0 = DirichletBC(VV, Constant((0,0)), "std::abs(x[0]) < 1e-10")

        #Mind sub(1) cause this only yields for F' (H)
        bc1 = DirichletBC(VV.sub(1), Constant(1), "std::abs(x[0]-%d) < 1e-10" % L

        #Previous known functionvalue
        fh_ = interpolate(Expression(("x[0]","x[0]")), VV)
        f_, h_ = split(fh_)

        F = h*vf*dx - f.dx(0)*vf*dx - inner(grad(h),grad(vh))*dx + f_*h.dx(0)*vh*
        #h*h_ is the linearsation of

        k = 0
        fh_1 = Function(VV) #The new unknown function
        error = 1

        while k < 100 and error > 1e-12:
                solve(lhs(F) == rhs(F), fh_, [bc0, bc1])
                #solve godtar ikke vf*dx arg..
                error = errornorm(fh_, fh_1)
                fh_1.assign(fh_) #updates the unknown for next it.
                print "Error = ", k, error
```

```
                k += 1

        viz = plot(f_)
        viz.write_png('146picard1')
        viz = plot(h_)
        viz.write_png('146picard2')
        interactive()

newton()
picard()
```

Assignment iii

In this assignment we are to work with Stokes flow. Assuming low Reynoldnumber the navier-stokes and continuity equations are reduced to:

$$\mu\nabla^2\mathbf{u} = \nabla p$$
$$\nabla\cdot\mathbf{u} = 0$$

By applying two test functions v and q, we end up with the variational forms for the equation:

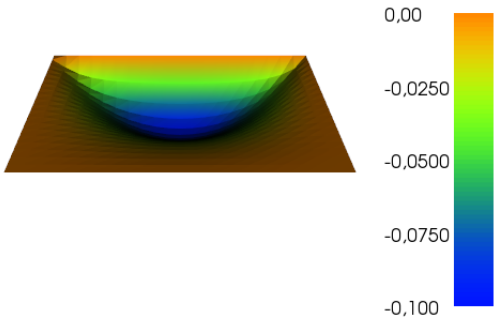
$$\int_\Gamma \mathbf{v}\cdot(\nabla\mathbf{u}\cdot\mathbf{n})ds - \mu\int_\Omega \nabla\mathbf{v}:\nabla\mathbf{u}dx = \int_\Gamma \mathbf{v}\cdot p\mathbf{n}ds - \int_\Omega p\nabla\cdot\mathbf{v}dx$$
$$\int_\Omega q\nabla\cdot\mathbf{u}dx = 0$$

By summing up the surface integral we get $\int_\Omega \mathbf{v}\cdot(p\mathbf{I}-\mu\nabla\mathbf{u})\cdot nds$ which we recognize as the total surface stress. In this case, the surface stress is zero due to pseudo-traction condition. Running my program I get the following results.

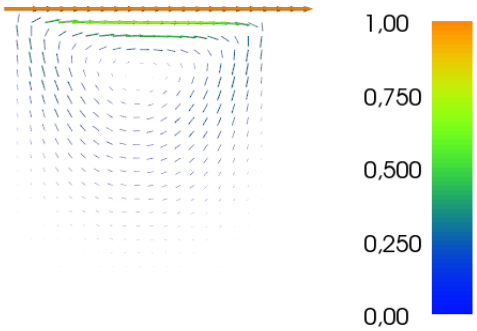
```
Solving linear variational problem.
[ 0.2  0.3]
```

I have really made an effort to correct my code to get the right coordinates for the vortex. It's clear that this has to be wrong according to my plot, but the plot itself is right.

Equation Stokes Flow



Equation Stokes Flow



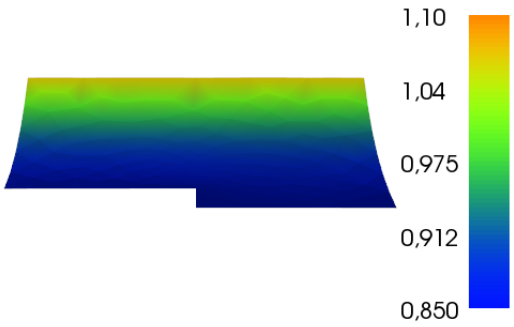

```
from dolfin import *

mu = 100
mesh = UnitSquareMesh(20,20)

#Different degree, due to unstabiity if the degree is the same
V = VectorFunctionSpace(mesh, 'CG', 2)
Q = FunctionSpace(mesh, 'CG', 1)

VQ = V*Q
u, p = TrialFunctions(VQ)
v, q = TestFunctions(VQ)

F = mu*inner(grad(v),grad(u))*dx - inner(div(v), p)*dx - inner(q, div(u))*dx
#Negative velocity divergence due to sym coefficient matrix, better iteration
def top(x, on_boundary):
        return x[1] > (1-DOLFIN_EPS)

bc0 = DirichletBC(VQ.sub(0), Constant((0.0,0)), DomainBoundary())
bc1 = DirichletBC(VQ.sub(0), Constant((1.0,0)), top)

up_ = Function(VQ)
solve(lhs(F) == rhs(F), up_, [bc0, bc1])
u_, p_ = up_.split()

u, v = u_.split()

w = v.dx(0) - u.dx(1) #Equation 136
psi = TrialFunction(Q)
psi_v = TestFunction(Q)

lhs = -inner(grad(psi_v), grad(psi))*dx #Equation 162
rhs = -dot(psi_v, w)*dx

bc = DirichletBC(Q, Constant(0), DomainBoundary())
psi_s = Function(Q)

solve(lhs == rhs, psi_s, bc)
```

```
vortex = psi_s.vector().array().argmin() #Lowest point
print(mesh.coordinates()[vortex])

viz = plot(psi_s)
viz.write_png('stokes')

interactive()

viz = plot(u_)
viz.write_png('stokes1')
#interactive()
```

Assignment iv

In this assignment we are also looking at Stokes flow. This time we have another mesh as shown in Figure(9). We also apply boundary conditions weakly, the shear stress is added in the variational formulation. The code is more or less the same as the assignment (iii), but with different mesh and variational form. We end up with the following output, code is provided.

Streamfunction



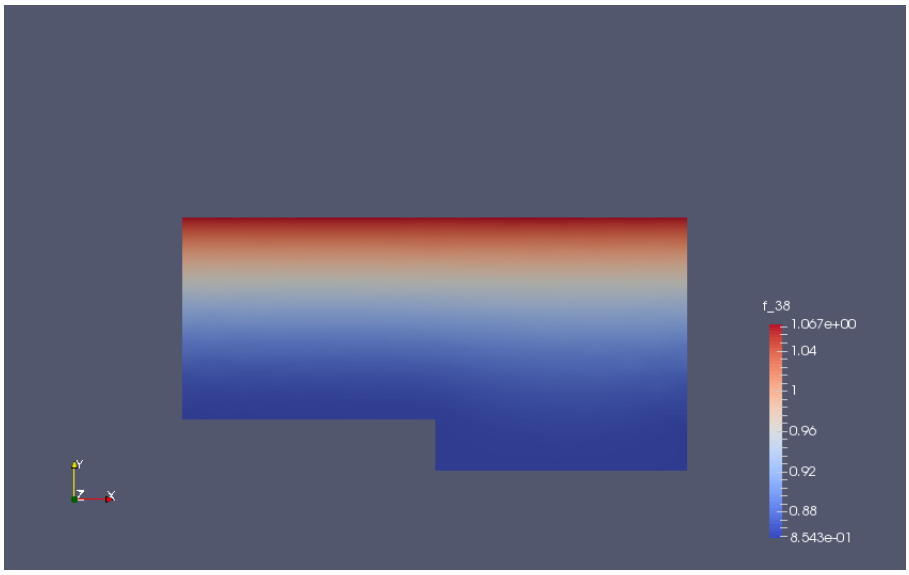

```
[[ 0.93508224  0.05647593]]
```

Again, clearly my location of the vortex is wrong. It should have a coordinate around 0.52 and 0.1 something. For some reason the code does not provide me the right returned values, so I guess it has something to do with my version.

b)

Using paraview I get the following plot

Streamfunction



c)

By using the provided code, I'm able to compute the flow through the left and right boundaries.

```
The flux on the left boundary is -1.165734e-15
The flux on the right boundary is 1.165734e-15
```

It seems very well that the mass is being conserved in this system.

d)

Here we are supposed to reverse the direction of flow in the top boundary to $\mathbf{u} = (-1, 0)$. Again since I had some trouble with getting the right coordinates for the vortex center, I'm sure that my code is somewhat right due to I get the following results:

```
[[ 0.93508224  0.05647593]]
```

So even if my point is wrong, it doesn't change. And so due to my results and also by some intuition, the vortex center doesn't change as the top boundary condition change direction.

e)

Since we applied weak boundary condition, the shear stress on the boundary is not zero. We compute:

$$\int_\Gamma (\gamma \cdot \mathbf{n}) \cdot \mathbf{n} ds$$

```
tau = -p_*Identity(2) + mu*(grad(u_) + grad(u_).T)

Bottom = AutoSubDomain(bottom)
mf.set_all(0)
Bottom.mark(mf, 1)

shear = assemble(dot(dot(tau, n), n)*ds, exterior_facet_domains=mf)
print shear
```

Which gives us the result:

```
122.635483966
```

By turning the direction of the flow, the result turns negative, so the value of the stress stays the same, but the direction changes.

```
from dolfin import *
import numpy as np
mu = 100.0

mesh = Mesh('figure.xml')

#Different degree, due to unstabiity if the degree is the same
V = VectorFunctionSpace(mesh, 'CG', 2)
Q = FunctionSpace(mesh, 'CG', 1)

VQ = V*Q
u, p = TrialFunctions(VQ)
v, q = TestFunctions(VQ)

Fu = mu*inner(grad(v),grad(u))*dx - inner(div(v), p)*dx - inner(q, div(u))*dx

#Negative velocity divergence due to sym coefficient matrix, better iteration
def top(x, on_boundary):
        return x[1] > (0.5-DOLFIN_EPS) and on_boundary

def bottom(x, on_boundary):
        return (x[1] < (0.1+DOLFIN_EPS) or abs(x[0]-0.5) < DOLFIN_EPS or x[1] < D

bc0 = DirichletBC(VQ.sub(0), Constant((0,0)), bottom)
bc1 = DirichletBC(VQ.sub(0), Constant((1,0)), top)

up_ = Function(VQ)
solve(lhs(Fu) == rhs(Fu), up_, [bc0, bc1])
u_, p_ = up_.split()

#plot(u_)
#interactive()

u, v = u_.split()

w = v.dx(0) - u.dx(1) #Equation 136

psi = TrialFunction(Q)
psi_v = TestFunction(Q)

#Fetching the normal of the mesh
n = FacetNormal(mesh)
grad_psi = as_vector((-v,u))

#Equation 162 WITH weak boundary
F = -inner(grad(psi_v), grad(psi))*dx + inner(psi_v*grad_psi, n)*ds + psi_v*w*dx

psi_s = Function(Q)

solve(lhs(F) == rhs(F), psi_s)

#viz = plot(psi_s)
#viz.write_png('stokesa')
#interactive()
```

```
bc = DirichletBC(Q, Constant(100), DomainBoundary())
bc.apply(psi_s.vector())
val = abs(psi_s.compute_vertex_values())

loc = np.where(val == val.min())
#print mesh.coordinates()[loc]


#Task B
#f = File("psi_s.pvd")
#f << psi_s

#Task C
def left(x, on_boundary):
        return x[0] < 1e-12 and on_boundary
def right(x, on_boundary):
        return x[0] > 1-DOLFIN_EPS and on_boundary

Left = AutoSubDomain(left)
mf = FacetFunction("size_t", mesh)
mf.set_all(0)
Left.mark(mf, 1)

Right = AutoSubDomain(right)
mf.set_all(0)
Right.mark(mf, 1)

Leftintegral = assemble(dot(u_, -n)*ds, exterior_facet_domains=mf)
Rightintegral = assemble(dot(u_, n)*ds, exterior_facet_domains=mf)
#print "The flux on the left boundary is %e" % Leftintegral
#print "The flux on the right boundary is %e" % Rightintegral

#Task D
def reverse():
        bc1 = DirichletBC(VQ.sub(0), Constant((-1,0)), top)

        up_ = Function(VQ)
        solve(lhs(Fu) == rhs(Fu), up_, [bc0, bc1])
        u_, p_ = up_.split()

        u, v = u_.split()

        w = v.dx(0) - u.dx(1) #Equation 136

        psi = TrialFunction(Q)
        psi_v = TestFunction(Q)

        #Fetching the normal of the mesh
        n = FacetNormal(mesh)
        grad_psi = as_vector((-v,u))

        #Equation 162 WITH weak boundary
        F = -inner(grad(psi_v), grad(psi))*dx + inner(psi_v*grad_psi, n)*ds + psi

        psi_s = Function(Q)
```

```
        solve(lhs(F) == rhs(F), psi_s)
        #plot(psi_s)
        #interactive()

        bc = DirichletBC(Q, Constant(100), DomainBoundary())
        bc.apply(psi_s.vector())
        val = abs(psi_s.compute_vertex_values())

        loc = np.where(val == val.min())
        print mesh.coordinates()[loc]

reverse()

#Task E

tau = -p_*Identity(2) + mu*(grad(u_) + grad(u_).T)

Bottom = AutoSubDomain(bottom)
mf.set_all(0)
Bottom.mark(mf, 1)

shear = assemble(dot(dot(tau, n), n)*ds, exterior_facet_domains=mf)
print shear
```