16_Step_12

February 2, 2016

 Thanks: Gilbert Forsyth for help writing the notebooks. NSF for support via CAREER award 1149784. [@LorenaABarba](https://twitter.com/LorenaABarba)

1 12 steps to Navier-Stokes

Did you make it this far? This is the last step! How long did it take you to write your own Navier-Stokes solver in Python following this interactive module? Let us know!

1.1 Step 12: Channel Flow with Navier-Stokes

The only difference between this final step and Step 11 is that we are going to add a source term to the u-momentum equation, to mimic the effect of a pressure-driven channel flow. Here are our modified Navier-Stokes equations:

$$\frac{\partial u}{\partial t}+u\frac{\partial u}{\partial x}+v\frac{\partial u}{\partial y}=-\frac{1}{\rho}\frac{\partial p}{\partial x}+\nu\left(\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}\right)+F$$

$$\frac{\partial v}{\partial t}+u\frac{\partial v}{\partial x}+v\frac{\partial v}{\partial y}=-\frac{1}{\rho}\frac{\partial p}{\partial y}+\nu\left(\frac{\partial^2 v}{\partial x^2}+\frac{\partial^2 v}{\partial y^2}\right)$$

$$\frac{\partial^2 p}{\partial x^2}+\frac{\partial^2 p}{\partial y^2}=-\rho\left(\frac{\partial u}{\partial x}\frac{\partial u}{\partial x}+2\frac{\partial u}{\partial y}\frac{\partial v}{\partial x}+\frac{\partial v}{\partial y}\frac{\partial v}{\partial y}\right)$$

1.1.1 Discretized equations

With patience and care, we write the discretized form of the equations. It is highly recommended that you write these in your own hand, mentally following each term as you write it.

The u-momentum equation:

$$\frac{u_{i,j}^{n+1}-u_{i,j}^{n}}{\Delta t}+u_{i,j}^{n}\frac{u_{i,j}^{n}-u_{i-1,j}^{n}}{\Delta x}+v_{i,j}^{n}\frac{u_{i,j}^{n}-u_{i,j-1}^{n}}{\Delta y} \tag{1}$$

$$=-\frac{1}{\rho}\frac{p_{i+1,j}^{n}-p_{i-1,j}^{n}}{2\Delta x} \tag{2}$$

$$+\nu\left(\frac{u_{i+1,j}^{n}-2u_{i,j}^{n}+u_{i-1,j}^{n}}{\Delta x^2}+\frac{u_{i,j+1}^{n}-2u_{i,j}^{n}+u_{i,j-1}^{n}}{\Delta y^2}\right)+F_{i,j} \tag{3}$$

The v-momentum equation:

$$\frac{v_{i,j}^{n+1} - v_{i,j}^n}{\Delta t} + u_{i,j}^n \frac{v_{i,j}^n - v_{i-1,j}^n}{\Delta x} + v_{i,j}^n \frac{v_{i,j}^n - v_{i,j-1}^n}{\Delta y} \tag{4}$$

$$= -\frac{1}{\rho} \frac{p_{i,j+1}^n - p_{i,j-1}^n}{2\Delta y} \tag{5}$$

$$+\nu \left(\frac{v_{i+1,j}^n - 2v_{i,j}^n + v_{i-1,j}^n}{\Delta x^2} + \frac{v_{i,j+1}^n - 2v_{i,j}^n + v_{i,j-1}^n}{\Delta y^2} \right) \tag{6}$$

And the pressure equation:

$$\frac{p_{i+1,j}^n - 2p_{i,j}^n + p_{i-1,j}^n}{\Delta x^2} + \frac{p_{i,j+1}^n - 2 * p_{i,j}^n + p_{i,j-1}^n}{\Delta y^2} \tag{7}$$

$$= \rho \left(\frac{1}{\Delta t} \left(\frac{u_{i+1,j} - u_{i-1,j}}{2\Delta x} + \frac{v_{i,j+1} - v_{i,j-1}}{2\Delta y} \right) \right. \tag{8}$$

$$- \frac{u_{i+1,j} - u_{i-1,j}}{2\Delta x} \frac{u_{i+1,j} - u_{i-1,j}}{2\Delta x} \tag{9}$$

$$-2 \frac{u_{i,j+1} - u_{i,j-1}}{2\Delta y} \frac{v_{i+1,j} - v_{i-1,j}}{2\Delta x} \tag{10}$$

$$\left. - \frac{v_{i,j+1} - v_{i,j-1}}{2\Delta y} \frac{v_{i,j+1} - v_{i,j-1}}{2\Delta y} \right) \tag{11}$$

As always, we need to re-arrange these equations to the form we need in the code to make the iterations proceed.

For the u- and v momentum equations, we isolate the velocity at time step `n+1`:

$$u_{i,j}^{n+1} = u_{i,j}^n - u_{i,j}^n \frac{\Delta t}{\Delta x}(u_{i,j}^n - u_{i-1,j}^n) - v_{i,j}^n \frac{\Delta t}{\Delta y}(u_{i,j}^n - u_{i,j-1}^n)$$

$$- \frac{\Delta t}{\rho 2\Delta x}(p_{i+1,j}^n - p_{i-1,j}^n) + \nu \left[\frac{\Delta t}{\Delta x^2}(u_{i+1,j}^n - 2u_{i,j}^n + u_{i-1,j}^n) \right.$$

$$\left. + \frac{\Delta t}{\Delta y^2}(u_{i,j+1}^n - 2u_{i,j}^n + u_{i,j-1}^n) \right] + F\Delta t$$

$$v_{i,j}^{n+1} = v_{i,j}^n - u_{i,j}^n \frac{\Delta t}{\Delta x}(v_{i,j}^n - v_{i-1,j}^n) - v_{i,j}^n \frac{\Delta t}{\Delta y}(v_{i,j}^n - v_{i,j-1}^n)$$

$$- \frac{\Delta t}{\rho 2\Delta y}(p_{i,j+1}^n - p_{i,j-1}^n) + \nu \left[\frac{\Delta t}{\Delta x^2}(v_{i+1,j}^n - 2v_{i,j}^n + v_{i-1,j}^n) \right.$$

$$\left. + \frac{\Delta t}{\Delta y^2}(v_{i,j+1}^n - 2v_{i,j}^n + v_{i,j-1}^n) \right]$$

And for the pressure equation, we isolate the term $p_{i,j}^n$ to iterate in pseudo-time:

$$p_{i,j}^n = \frac{(p_{i+1,j}^n + p_{i-1,j}^n)\Delta y^2 + (p_{i,j+1}^n + p_{i,j-1}^n)\Delta x^2}{2(\Delta x^2 + \Delta y^2)} - \frac{\rho \Delta x^2 \Delta y^2}{2(\Delta x^2 + \Delta y^2)} \times$$

$$\left[\frac{1}{\Delta t} \left(\frac{u_{i+1,j} - u_{i-1,j}}{2\Delta x} + \frac{v_{i,j+1} - v_{i,j-1}}{2\Delta y} \right) - \frac{u_{i+1,j} - u_{i-1,j}}{2\Delta x} \frac{u_{i+1,j} - u_{i-1,j}}{2\Delta x} \right.$$

$$\left. -2 \frac{u_{i,j+1} - u_{i,j-1}}{2\Delta y} \frac{v_{i+1,j} - v_{i-1,j}}{2\Delta x} - \frac{v_{i,j+1} - v_{i,j-1}}{2\Delta y} \frac{v_{i,j+1} - v_{i,j-1}}{2\Delta y} \right]$$

The initial condition is $u, v, p = 0$ everywhere, and at the boundary conditions are:
u, v, p are periodic on $x = 0, 2$
$u, v = 0$ at $y = 0, 2$

$\frac{\partial p}{\partial y} = 0$ at $y = 0, 2$
$F = 1$ everywhere.
Let's begin by importing our usual run of libraries:

```
In [1]: from mpl_toolkits.mplot3d import Axes3D
        from matplotlib import cm
        import matplotlib.pyplot as plt
        import numpy as np
        %matplotlib inline
```

In step 11, we isolated a portion of our transposed equation to make it easier to parse and we're going to do the same thing here. One thing to note is that we have periodic boundary conditions throughout this grid, so we need to explicitly calculate the values at the leading and trailing edge of our `u` vector.

```
In [2]: def buildUpB(rho, dt, dx, dy, u, v):
            b = np.zeros_like(u)
            b[1:-1,1:-1]=rho*(1/dt*((u[1:-1,2:]-u[1:-1,0:-2])/(2*dx)+(v[2:,1:-1]-v[0:-2,1:-1])/(2*dy))-\
                              ((u[1:-1,2:]-u[1:-1,0:-2])/(2*dx))**2-\
                              2*((u[2:,1:-1]-u[0:-2,1:-1])/(2*dy)*(v[1:-1,2:]-v[1:-1,0:-2])/(2*dx))-\
                              ((v[2:,1:-1]-v[0:-2,1:-1])/(2*dy))**2)

            ####Periodic BC Pressure @ x = 2
            b[1:-1,-1]=rho*(1/dt*((u[1:-1,0]-u[1:-1,-2])/(2*dx)+(v[2:,-1]-v[0:-2,-1])/(2*dy))-\
                            ((u[1:-1,0]-u[1:-1,-2])/(2*dx))**2-\
                            2*((u[2:,-1]-u[0:-2,-1])/(2*dy)*(v[1:-1,0]-v[1:-1,-2])/(2*dx))-\
                            ((v[2:,-1]-v[0:-2,-1])/(2*dy))**2)

            ####Periodic BC Pressure @ x = 0
            b[1:-1,0]=rho*(1/dt*((u[1:-1,1]-u[1:-1,-1])/(2*dx)+(v[2:,0]-v[0:-2,0])/(2*dy))-\
                            ((u[1:-1,1]-u[1:-1,-1])/(2*dx))**2-\
                            2*((u[2:,0]-u[0:-2,0])/(2*dy)*(v[1:-1,1]-v[1:-1,-1])/(2*dx))-\
                            ((v[2:,0]-v[0:-2,0])/(2*dy))**2)

            return b
```

We'll also define a Pressure Poisson iterative function, again like we did in Step 11. Once more, note that we have to include the periodic boundary conditions at the leading and trailing edge. We also have to specify the boundary conditions at the top and bottom of our grid.

```
In [3]: def presPoissPeriodic(p, dx, dy):
            pn = np.empty_like(p)

            for q in range(nit):
                pn = p.copy()
                p[1:-1,1:-1] = ((pn[1:-1,2:]+pn[1:-1,0:-2])*dy**2+(pn[2:,1:-1]+pn[0:-2,1:-1])*dx**2)/\
                    (2*(dx**2+dy**2)) -\
                    dx**2*dy**2/(2*(dx**2+dy**2))*b[1:-1,1:-1]

                ####Periodic BC Pressure @ x = 2
                p[1:-1,-1] = ((pn[1:-1,0]+pn[1:-1,-2])*dy**2+(pn[2:,-1]+pn[0:-2,-1])*dx**2)/\
                    (2*(dx**2+dy**2)) -\
                    dx**2*dy**2/(2*(dx**2+dy**2))*b[1:-1,-1]

                ####Periodic BC Pressure @ x = 0
                p[1:-1,0] = ((pn[1:-1,1]+pn[1:-1,-1])*dy**2+(pn[2:,0]+pn[0:-2,0])*dx**2)/\
```

```
            (2*(dx**2+dy**2)) -\
            dx**2*dy**2/(2*(dx**2+dy**2))*b[1:-1,0]

        ####Wall boundary conditions, pressure
        p[-1,:] =p[-2,:]        ##dp/dy = 0 at y = 2
        p[0,:] = p[1,:]                 ##dp/dy = 0 at y = 0

    return p
```

Now we have our familiar list of variables and initial conditions to declare before we start.

```
In [4]: ##variable declarations
        nx = 41
        ny = 41
        nt = 10
        nit=50
        c = 1
        dx = 2.0/(nx-1)
        dy = 2.0/(ny-1)
        x = np.linspace(0,2,nx)
        y = np.linspace(0,2,ny)
        X,Y = np.meshgrid(x,y)


        ##physical variables
        rho = 1
        nu = .1
        F = 1
        dt = .01

        #initial conditions
        u = np.zeros((ny,nx)) ##create a XxY vector of 0's
        un = np.zeros((ny,nx)) ##create a XxY vector of 0's

        v = np.zeros((ny,nx)) ##create a XxY vector of 0's
        vn = np.zeros((ny,nx)) ##create a XxY vector of 0's

        p = np.ones((ny,nx)) ##create a XxY vector of 0's
        pn = np.ones((ny,nx)) ##create a XxY vector of 0's

        b = np.zeros((ny,nx))
```

For the meat of our computation, we're going to reach back to a trick we used in Step 9 for Laplace's Equation. We're interested in what our grid will look like once we've reached a near-steady state. We can either specify a number of timesteps `nt` and increment it until we're satisfied with the results, or we can tell our code to run until the difference between two consecutive iterations is very small.

We also have to manage **8** separate boundary conditions for each iteration. The code below writes each of them out explicitly. If you're interested in a challenge, you can try to write a function which can handle some or all of these boundary conditions. If you're interested in tackling that, you should probably read up on Python dictionaries.

```
In [5]: udiff = 1
        stepcount = 0

        while udiff > .001:
```

```
un = u.copy()
vn = v.copy()

b = buildUpB(rho, dt, dx, dy, u, v)
p = presPoissPeriodic(p, dx, dy)

u[1:-1,1:-1] = un[1:-1,1:-1]-\
    un[1:-1,1:-1]*dt/dx*(un[1:-1,1:-1]-un[1:-1,0:-2])-\
    vn[1:-1,1:-1]*dt/dy*(un[1:-1,1:-1]-un[0:-2,1:-1])-\
    dt/(2*rho*dx)*(p[1:-1,2:]-p[1:-1,0:-2])+\
    nu*(dt/dx**2*(un[1:-1,2:]-2*un[1:-1,1:-1]+un[1:-1,0:-2])+\
    dt/dy**2*(un[2:,1:-1]-2*un[1:-1,1:-1]+un[0:-2,1:-1]))+F*dt

v[1:-1,1:-1] = vn[1:-1,1:-1]-\
    un[1:-1,1:-1]*dt/dx*(vn[1:-1,1:-1]-vn[1:-1,0:-2])-\
    vn[1:-1,1:-1]*dt/dy*(vn[1:-1,1:-1]-vn[0:-2,1:-1])-\
    dt/(2*rho*dy)*(p[2:,1:-1]-p[0:-2,1:-1])+\
    nu*(dt/dx**2*(vn[1:-1,2:]-2*vn[1:-1,1:-1]+vn[1:-1,0:-2])+\
    (dt/dy**2*(vn[2:,1:-1]-2*vn[1:-1,1:-1]+vn[0:-2,1:-1])))

####Periodic BC u @ x = 2
u[1:-1,-1] = un[1:-1,-1]-\
    un[1:-1,-1]*dt/dx*(un[1:-1,-1]-un[1:-1,-2])-\
    vn[1:-1,-1]*dt/dy*(un[1:-1,-1]-un[0:-2,-1])-\
    dt/(2*rho*dx)*(p[1:-1,0]-p[1:-1,-2])+\
    nu*(dt/dx**2*(un[1:-1,0]-2*un[1:-1,-1]+un[1:-1,-2])+\
    dt/dy**2*(un[2:,-1]-2*un[1:-1,-1]+un[0:-2,-1]))+F*dt

####Periodic BC u @ x = 0
u[1:-1,0] = un[1:-1,0]-\
    un[1:-1,0]*dt/dx*(un[1:-1,0]-un[1:-1,-1])-\
    vn[1:-1,0]*dt/dy*(un[1:-1,0]-un[0:-2,0])-\
    dt/(2*rho*dx)*(p[1:-1,1]-p[1:-1,-1])+\
    nu*(dt/dx**2*(un[1:-1,1]-2*un[1:-1,0]+un[1:-1,-1])+\
    dt/dy**2*(un[2:,0]-2*un[1:-1,0]+un[0:-2,0]))+F*dt

####Periodic BC v @ x = 2
v[1:-1,-1] = vn[1:-1,-1]-\
un[1:-1,-1]*dt/dx*(vn[1:-1,-1]-vn[1:-1,-2])-\
    vn[1:-1,-1]*dt/dy*(vn[1:-1,-1]-vn[0:-2,-1])-\
    dt/(2*rho*dy)*(p[2:,-1]-p[0:-2,-1])+\
    nu*(dt/dx**2*(vn[1:-1,0]-2*vn[1:-1,-1]+vn[1:-1,-2])+\
    (dt/dy**2*(vn[2:,-1]-2*vn[1:-1,-1]+vn[0:-2,-1])))

####Periodic BC v @ x = 0
v[1:-1,0] = vn[1:-1,0]-\
    un[1:-1,0]*dt/dx*(vn[1:-1,0]-vn[1:-1,-1])-\
    vn[1:-1,0]*dt/dy*(vn[1:-1,0]-vn[0:-2,0])-\
    dt/(2*rho*dy)*(p[2:,0]-p[0:-2,0])+\
    nu*(dt/dx**2*(vn[1:-1,1]-2*vn[1:-1,0]+vn[1:-1,-1])+\
    (dt/dy**2*(vn[2:,0]-2*vn[1:-1,0]+vn[0:-2,0])))


####Wall BC: u,v = 0 @ y = 0,2
```

```
u[0,:] = 0
u[-1,:] = 0
v[0,:] = 0
v[-1,:]=0

udiff = (np.sum(u)-np.sum(un))/np.sum(u)
stepcount += 1
```

You can see that we've also included a variable `stepcount` to see how many iterations our loop went through before our stop condition was met.

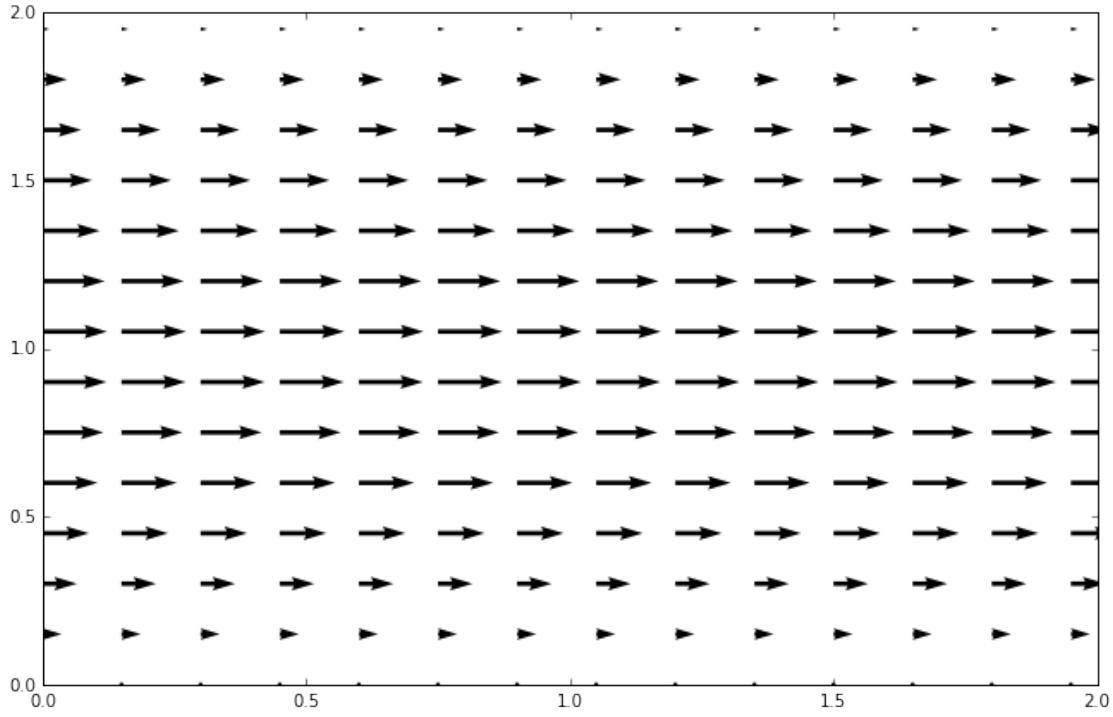
```
In [6]: print stepcount
```

```
499
```

If you want to see how the number of iterations increases as our `udiff` condition gets smaller and smaller, try defining a function to perform the `while` loop written above that takes an input `udiff` and outputs the number of iterations that the function runs.

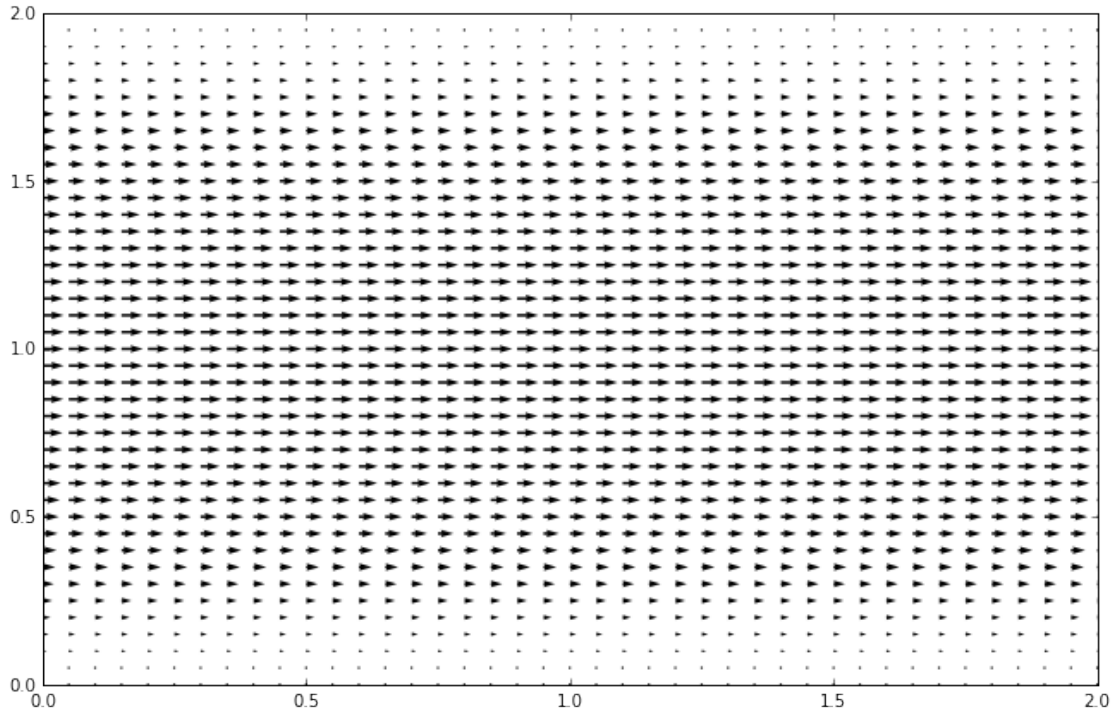
For now, let's look at our results. We've used the quiver function to look at the cavity flow results and it works well for channel flow, too.

```
In [7]: fig = plt.figure(figsize = (11,7), dpi=100)
        plt.quiver(X[::3, ::3], Y[::3, ::3], u[::3, ::3], v[::3, ::3]);
```



The structures in the `quiver` command that look like `[::3, ::3]` are useful when dealing with large amounts of data that you want to visualize. The one used above tells `matplotlib` to only plot every 3rd data point. If we leave it out, you can see that the results can appear a little crowded.

```
In [8]: fig = plt.figure(figsize = (11,7), dpi=100)
        plt.quiver(X, Y, u, v);
```



1.2 Learn more

What is the meaning of the F term? Step 12 is an exercise demonstrating the problem of flow in a channel or pipe. If you recall from your fluid mechanics class, a specified pressure gradient is what drives Poisseulle flow.

Recall the x-momentum equation:

$$\frac{\partial u}{\partial t} + u \cdot \nabla u = -\frac{\partial p}{\partial x} + \nu \nabla^2 u$$

What we actually do in Step 12 is split the pressure into steady and unsteady components $p = P + p'$. The applied steady pressure gradient is the constant $-\frac{\partial P}{\partial x} = F$ (interpreted as a source term), and the unsteady component is $\frac{\partial p'}{\partial x}$. So the pressure that we solve for in Step 12 is actually p', which for a steady flow is in fact equal to zero everywhere.

Why did we do this?

Note that we use periodic boundary conditions for this flow. For a flow with a constant pressure gradient, the value of pressure on the left edge of the domain must be different from the pressure at the right edge. So we cannot apply periodic boundary conditions on the pressure directly. It is easier to fix the gradient and then solve for the perturbations in pressure.

Shouldn't we always expect a uniform/constant p' then?

That's true only in the case of steady laminar flows. At high Reynolds numbers, flows in channels can become turbulent, and we will see unsteady fluctuations in the pressure, which will result in non-zero values for p'.

In step 12, note that the pressure field itself is not constant, but it's the pressure perturbation field that is. The pressure field varies linearly along the channel with slope equal to the pressure gradient. Also, for incompressible flows, the absolute value of the pressure is inconsequential.

And explore more CFD materials online The interactive module **12 steps to Navier-Stokes** is one of several components of the Computational Fluid Dynamics class taught by Prof. Lorena A. Barba in Boston University between 2009 and 2013.

For a sample of what the othe components of this class are, you can explore the **Resources** section of the Spring 2013 version of the course's Piazza site.

```
In [9]: from IPython.core.display import HTML
        def css_styling():
            styles = open("../styles/custom.css", "r").read()
            return HTML(styles)
        css_styling()
```

```
Out[9]: <IPython.core.display.HTML at 0x7f9ed121c590>
```

(The cell above executes the style for this notebook.)