# 12_Step_9

January 19, 2016

## 1 12 steps to Navier-Stokes

---

In the previous step, we solved the 2D Burgers' equation: an important equation in the study of fluid mechanics because it contains the full convective nonlinearity of the flow equations. With that exercise, we also build the experience to incrementatlly code a Navier-Stokes solver.

In the next two steps, we will solve Laplace and then Poisson equation. We will then put it all together!

### 1.1 Step 9: 2D Laplace Equation

---

Here is Laplace's equation in 2D:

$$\frac{\partial^2 p}{\partial x^2} + \frac{\partial^2 p}{\partial y^2} = 0$$

We know how to discretize a 2nd order derivative. But think about this for a minute — Laplace's equation has the features typical of diffusion phenomena. For this reason, it has to be discretized with central differences, so that the discretization is consistent with the physics we want to simulate.

The discretized equation is:

$$\frac{p_{i+1,j}^n - 2p_{i,j}^n + p_{i-1,j}^n}{\Delta x^2} + \frac{p_{i,j+1}^n - 2p_{i,j}^n + p_{i,j-1}^n}{\Delta y^2} = 0$$

Notice that the Laplace Equation does not have a time dependence — there is no $p^{n+1}$. Instead of tracking a wave through time (like in the previous steps), the Laplace equation calculates the equilibrium state of a system under the supplied boundary conditions.

If you have taken coursework in Heat Transfer, you will recognize the Laplace Equation as the steady-state heat equation.

Instead of calculating where the system will be at some time $t$, we will iteratively solve for $p_{i,j}^n$ until it meets a condition that we specify. The system will reach equilibrium only as the number of iterations tends to $\infty$, but we can approximate the equilibrium state by iterating until the change between one iteration and the next is very small.

Let's rearrange the discretized equation, solving for $p_{i,j}^n$:

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

Using second-order central-difference schemes in both directions is the most widely applied method for the Laplace operator. It is also known as the **five-point difference operator**, alluding to its stencil.

We are going to solve Laplace's equation numerically by assuming an initial state of $p = 0$ everywhere. Then we add boundary conditions as follows:

$p = 0$ at $x = 0$

$p = y$ at $x = 2$

$\frac{\partial p}{\partial y} = 0$ at $y = 0,\ 1$

Under these conditions, there is an analytical solution for Laplace's equation:

$$p(x, y) = \frac{x}{4} - 4 \sum_{n=1,odd}^{\infty} \frac{1}{(n\pi)^2 \sinh 2n\pi} \sinh n\pi x \cos n\pi y$$

**Exercise** Write your own code to solve Poisson's equation using loops, in the style of coding used in our first lessons. Then, consider the demonstration of how to write it using functions (below) and modify your code in that style. Can you think of reasons to abandon the old style and adopt modular coding?

Other tips:

- Visualize each step of the iterative process
- Think about what the boundary conditions are doing
- Think about what the PDE is doing

### 1.1.1 Using functions

Remember the lesson on writing functions with Python? We will use that style of code in this exercise.

We're going to define two functions: one that plots our data in a 3D projection plot and the other that iterates to solve for $p$ until the change in the L1 Norm of $p$ is less than a specified value.

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

```
In [2]: def plot2D(x, y, p):
            fig = plt.figure(figsize=(11,7), dpi=100)
            ax = fig.gca(projection='3d')
            X,Y = np.meshgrid(x,y)
            surf = ax.plot_surface(X,Y,p[:], rstride=1, cstride=1, cmap=cm.coolwarm,
                    linewidth=0, antialiased=False)
            ax.set_xlim(0,2)
            ax.set_ylim(0,1)
            ax.view_init(30,225)
```

The function `plot2D` takes three arguments, an x-vector, a y-vector and our p matrix. Given these three values, it produces a 3D projection plot, sets the plot limits and gives us a nice viewing angle.

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

```
In [3]: def laplace2d(p, y, dx, dy, l1norm_target):
            l1norm = 1
            pn = np.empty_like(p)

            while l1norm > l1norm_target:
                pn = p.copy()
                p[1:-1,1:-1] = (dy**2*(pn[1:-1,2:]+pn[1:-1,0:-2])+dx**2*(pn[2:,1:-1]+pn[0:-2,1:-1]))/(2

                p[:,0] = 0        ##p = 0 @ x = 0
                p[:,-1] = y       ##p = y @ x = 2
                p[0,:] = p[1,:]   ##dp/dy = 0 @ y = 0
                p[-1,:] = p[-2,:]    ##dp/dy = 0 @ y = 1
                l1norm = (np.sum(np.abs(p[:])-np.abs(pn[:])))/np.sum(np.abs(pn[:]))

            return p
```

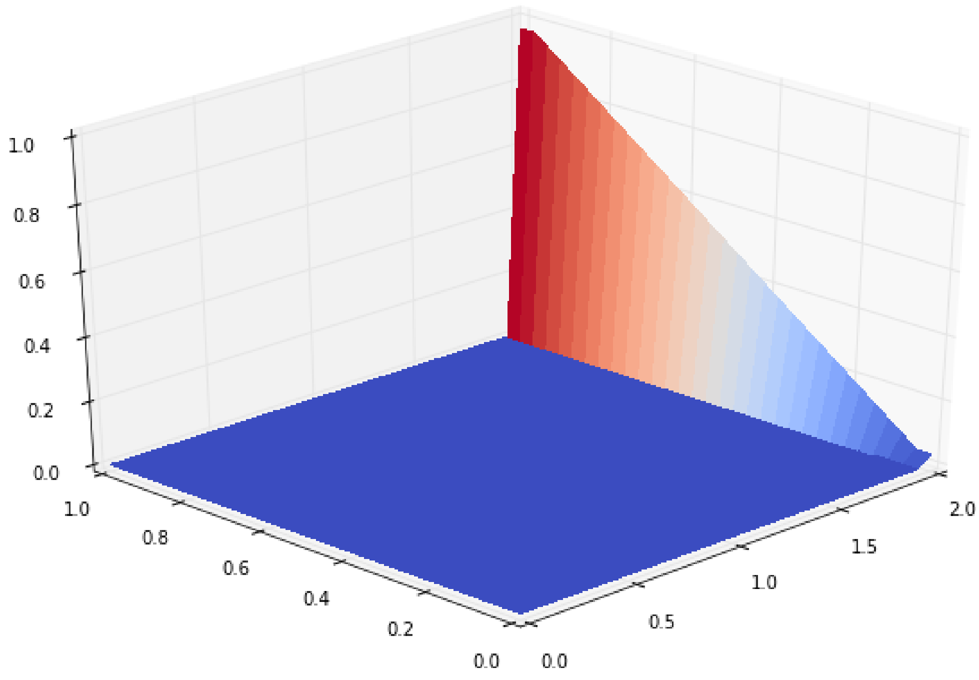
`laplace2d` takes five arguments, the `p` matrix, the `y`-vector, `dx`, `dy` and the value `l1norm_target`. This last value defines how close the `p` matrix should be in two consecutive iterations before the loop breaks and returns the calculated `p` value.

Note that when executing the cells above in your own notebook, there will be no output. You have defined the function but you have not yet called the function. It is now available for you to use, the same as `np.linspace` or any other function in our namespace.

```
In [4]: ##variable declarations
        nx = 31
        ny = 31
        c = 1
        dx = 2.0/(nx-1)
        dy = 2.0/(ny-1)


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


        ##plotting aids
        x = np.linspace(0,2,nx)
        y = np.linspace(0,1,ny)

        ##boundary conditions
        p[:,0] = 0      ##p = 0 @ x = 0
        p[:,-1] = y     ##p = y @ x = 2
        p[0,:] = p[1,:]     ##dp/dy = 0 @ y = 0
        p[-1,:] = p[-2,:]     ##dp/dy = 0 @ y = 1
```

Now let's try using our `plot2D` function to look at our initial conditions. If the function has been correctly defined, you should be able to begin typing `plot2D` and hit the **tab** key for auto-complete options.

```
In [5]: plot2D(x, y, p)
```



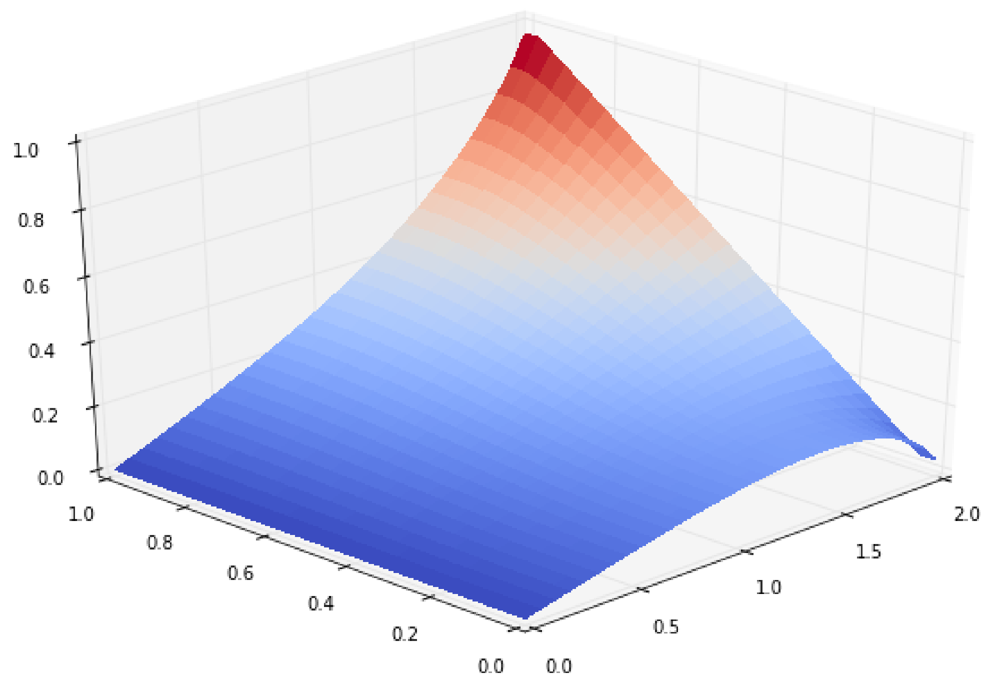
It worked! This is the initial state of our problem, where the value of `p` is zero everywhere except for along $x = 2$ where $p = y$. Now let's try to run our `laplace2d` function with a specified L1 target of .01

[Hint: if you are having trouble remembering the order in which variables are sent to a function, you can just type `laplace2d(` and the iPython Notebook will put up a little popup box to remind you]

```
In [6]: p = laplace2d(p, y, dx, dy, 1e-4)
```

Now try plotting this new value of `p` with our plot function.

```
In [7]: plot2D(x, y, p)
```



---

## 1.2 Learn More

The next step will be to solve Poisson's equation. Watch **Video Lesson 11** on You Tube to understand why we need Poisson's equation in CFD.

```
In [8]: from IPython.display import YouTubeVideo
        YouTubeVideo('ZjfxA3qq2Lg')

Out[8]: <IPython.lib.display.YouTubeVideo at 0x7f474a87cdd0>
```

And for a detailed walk-through of the discretization of Laplace and Poisson equations (steps 9 and 10), watch **Video Lesson 12** on You Tube:

```
In [9]: from IPython.display import YouTubeVideo
        YouTubeVideo('iwL8ashXhWU')

Out[9]: <IPython.lib.display.YouTubeVideo at 0x7f474a6515d0>

In [10]: from IPython.core.display import HTML
         def css_styling():
             styles = open("../styles/custom.css", "r").read()
             return HTML(styles)
         css_styling()

Out[10]: <IPython.core.display.HTML at 0x7f474a75b9d0>
```

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