05_Step_4

January 6, 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

We continue our journey to solve the Navier-Stokes equation with Step 4. But don't continue unless you have completed the previous steps! In fact, this next step will be a combination of the two previous ones. The wonders of code reuse!

1.1 Step 4: Burgers' Equation

You can read about Burgers' Equation on its [wikipedia page](http://en.wikipedia.org/wiki/Burgers'_equation).
Burgers' equation in one spatial dimension looks like this:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \nu\frac{\partial^2 u}{\partial x^2}$$

As you can see, it is a combination of non-linear convection and diffusion. It is surprising how much you learn from this neat little equation!

We can discretize it using the methods we've already detailed in Steps 1 to 3. Using forward difference for time, backward difference for space and our 2nd-order method for the second derivatives yields:

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} + u_i^n\frac{u_i^n - u_{i-1}^n}{\Delta x} = \nu\frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2}$$

As before, once we have an initial condition, the only unknown is u_i^{n+1}. We will step in time as follows:

$$u_i^{n+1} = u_i^n - u_i^n\frac{\Delta t}{\Delta x}(u_i^n - u_{i-1}^n) + \nu\frac{\Delta t}{\Delta x^2}(u_{i+1}^n - 2u_i^n + u_{i-1}^n)$$

1.1.1 Initial and Boundary Conditions

To examine some interesting properties of Burgers' equation, it is helpful to use different initial and boundary conditions than we've been using for previous steps.

Our initial condition for this problem is going to be:

$$u = -\frac{2\nu}{\phi}\frac{\partial \phi}{\partial x} + 4 \tag{1}$$

$$\phi = \exp\left(\frac{-x^2}{4\nu}\right) + \exp\left(\frac{-(x-2\pi)^2}{4\nu}\right) \tag{2}$$

This has an analytical solution, given by:

$$u = -\frac{2\nu}{\phi}\frac{\partial \phi}{\partial x} + 4 \tag{3}$$

$$\phi = \exp\left(\frac{-(x-4t)^2}{4\nu(t+1)}\right) + \exp\left(\frac{-(x-4t-2\pi)^2}{4\nu(t+1)}\right) \tag{4}$$

Our boundary condition will be:

$$u(0) = u(2\pi)$$

This is called a periodic boundary condition. Pay attention! This will cause you a bit of headache if you don't tread carefully.

1.1.2 Saving Time with SymPy

The initial condition we're using for Burgers' Equation can be a bit of a pain to evaluate by hand. The derivative $\frac{\partial \phi}{\partial x}$ isn't too terribly difficult, but it would be easy to drop a sign or forget a factor of x somewhere, so we're going to use SymPy to help us out.

SymPy is the symbolic math library for Python. It has a lot of the same symbolic math functionality as Mathematica with the added benefit that we can easily translate its results back into our Python calculations (it is also free and open source).

Start by loading the SymPy library, together with our favorite library, NumPy.

In [1]:
```
import numpy as np
import sympy
```

We're also going to tell SymPy that we want all of its output to be rendered using . This will make our Notebook beautiful!

In [2]:
```
from sympy import init_printing
init_printing(use_latex=True)
```

Start by setting up symbolic variables for the three variables in our initial condition and then type out the full equation for ϕ. We should get a nicely rendered version of our ϕ equation.

In [3]:
```
x, nu, t = sympy.symbols('x nu t')
phi = sympy.exp(-(x-4*t)**2/(4*nu*(t+1))) + sympy.exp(-(x-4*t-2*np.pi)**2/(4*nu*(t+1)))
phi
```

Out[3]:

$$e^{-\frac{(-4t+x-6.28318530717959)^2}{4\nu(t+1)}} + e^{-\frac{(-4t+x)^2}{4\nu(t+1)}}$$

It's maybe a little small, but that looks right. Now to evaluate our partial derivative $\frac{\partial \phi}{\partial x}$ is a trivial task.

In [4]:
```
phiprime = phi.diff(x)
phiprime
```

Out[4]:

$$-\frac{e^{-\frac{(-4t+x)^2}{4\nu(t+1)}}}{4\nu\left(t+1\right)}\left(-8t+2x\right) - \frac{1}{4\nu\left(t+1\right)}\left(-8t+2x-12.5663706143592\right)e^{-\frac{(-4t+x-6.28318530717959)^2}{4\nu(t+1)}}$$

If you want to see the unrendered version, just use the Python print command.

In [5]:
```
print phiprime
```

```
-(-8*t + 2*x)*exp(-(-4*t + x)**2/(4*nu*(t + 1)))/(4*nu*(t + 1)) - (-8*t + 2*x - 12.5663706143592)*exp(-
```

1.1.3 Now what?

Now that we have the Pythonic version of our derivative, we can finish writing out the full initial condition equation and then translate it into a usable Python expression. For this, we'll use the lambdify function, which takes a SymPy symbolic equation and turns it into a callable function.

```
In [6]: from sympy.utilities.lambdify import lambdify

        u = -2*nu*(phiprime/phi)+4
        print u
```

```
-2*nu*(-(-8*t + 2*x)*exp(-(-4*t + x)**2/(4*nu*(t + 1)))/(4*nu*(t + 1)) - (-8*t + 2*x - 12.5663706143592
```

1.1.4 Lambdify

To lambdify this expression into a useable function, we tell lambdify which variables to request and the function we want to plug them in to.

```
In [7]: ufunc = lambdify((t, x, nu), u)
        print ufunc(1,4,3)
```

```
3.49170664206
```

1.1.5 Back to Burgers' Equation

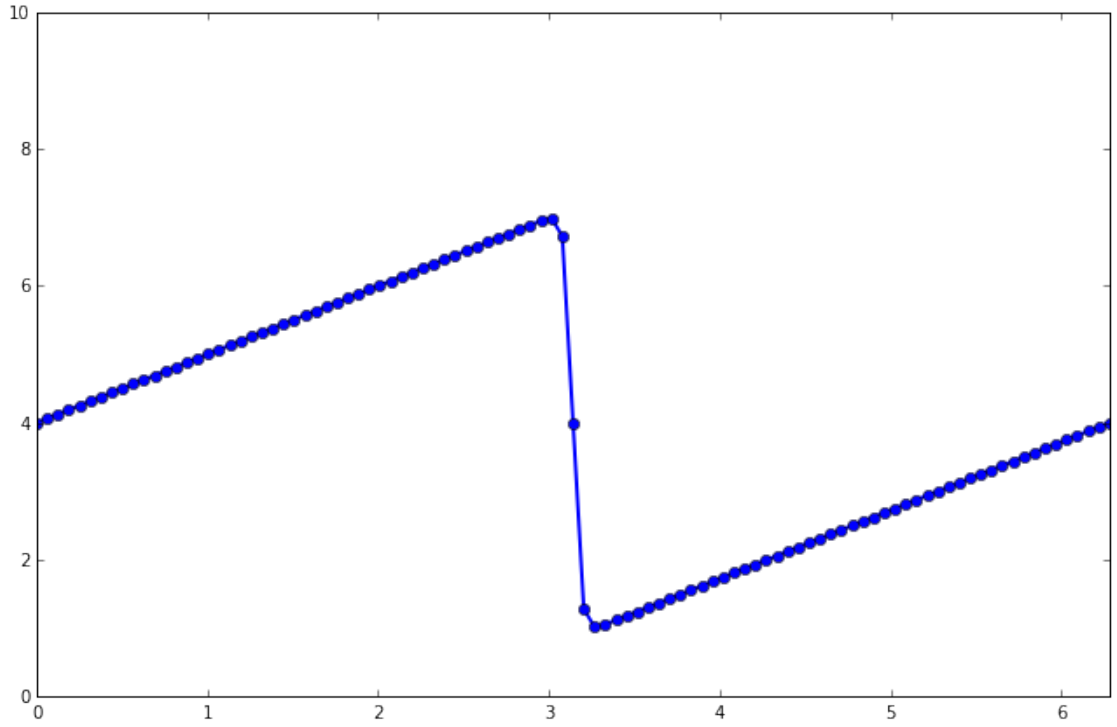
Now that we have the initial conditions set up, we can proceed and finish setting up the problem. We can generate the plot of the initial condition using our lambdify-ed function.

```
In [8]: import matplotlib.pyplot as plt
        %matplotlib inline

        ###variable declarations
        nx = 101
        nt = 100
        dx = 2*np.pi/(nx-1)
        nu = .07
        dt = dx*nu

        x = np.linspace(0, 2*np.pi, nx)
        #u = np.empty(nx)
        un = np.empty(nx)
        t = 0

        u = np.asarray([ufunc(t, x0, nu) for x0 in x])
        u
```

```
Out[8]: array([ 4.        ,  4.06283185,  4.12566371,  4.18849556,  4.25132741,
                4.31415927,  4.37699112,  4.43982297,  4.50265482,  4.56548668,
                4.62831853,  4.69115038,  4.75398224,  4.81681409,  4.87964594,
                4.9424778 ,  5.00530965,  5.0681415 ,  5.13097336,  5.19380521,
                5.25663706,  5.31946891,  5.38230077,  5.44513262,  5.50796447,
                5.57079633,  5.63362818,  5.69646003,  5.75929189,  5.82212374,
                5.88495559,  5.94778745,  6.0106193 ,  6.07345115,  6.136283  ,
                6.19911486,  6.26194671,  6.32477856,  6.38761042,  6.45044227,
                6.51327412,  6.57610598,  6.63893783,  6.70176967,  6.76460125,
                6.82742866,  6.89018589,  6.95176632,  6.99367964,  6.72527549,
```

```
4.        ,  1.27472451,  1.00632036,  1.04823368,  1.10981411,
1.17257134,  1.23539875,  1.29823033,  1.36106217,  1.42389402,
1.48672588,  1.54955773,  1.61238958,  1.67522144,  1.73805329,
1.80088514,  1.863717  ,  1.92654885,  1.9893807 ,  2.05221255,
2.11504441,  2.17787626,  2.24070811,  2.30353997,  2.36637182,
2.42920367,  2.49203553,  2.55486738,  2.61769923,  2.68053109,
2.74336294,  2.80619479,  2.86902664,  2.9318585 ,  2.99469035,
3.0575222 ,  3.12035406,  3.18318591,  3.24601776,  3.30884962,
3.37168147,  3.43451332,  3.49734518,  3.56017703,  3.62300888,
3.68584073,  3.74867259,  3.81150444,  3.87433629,  3.93716815,  4.        ])
```

In [9]:
```
plt.figure(figsize=(11,7), dpi=100)
plt.plot(x,u, marker='o', lw=2)
plt.xlim([0,2*np.pi])
plt.ylim([0,10]);
```



This is definitely not the hat function we've been dealing with until now. We call it a "saw-tooth function". Let's proceed forward and see what happens.

1.1.6 Periodic Boundary Conditions

One of the big differences between Step 4 and the previous lessons is the use of periodic boundary conditions. If you experiment with Steps 1 and 2 and make the simulation run longer (by increasing `nt`) you will notice that the wave will keep moving to the right until it no longer even shows up in the plot.

With periodic boundary conditions, when a point gets to the right-hand side of the frame, it wraps around back to the front of the frame.

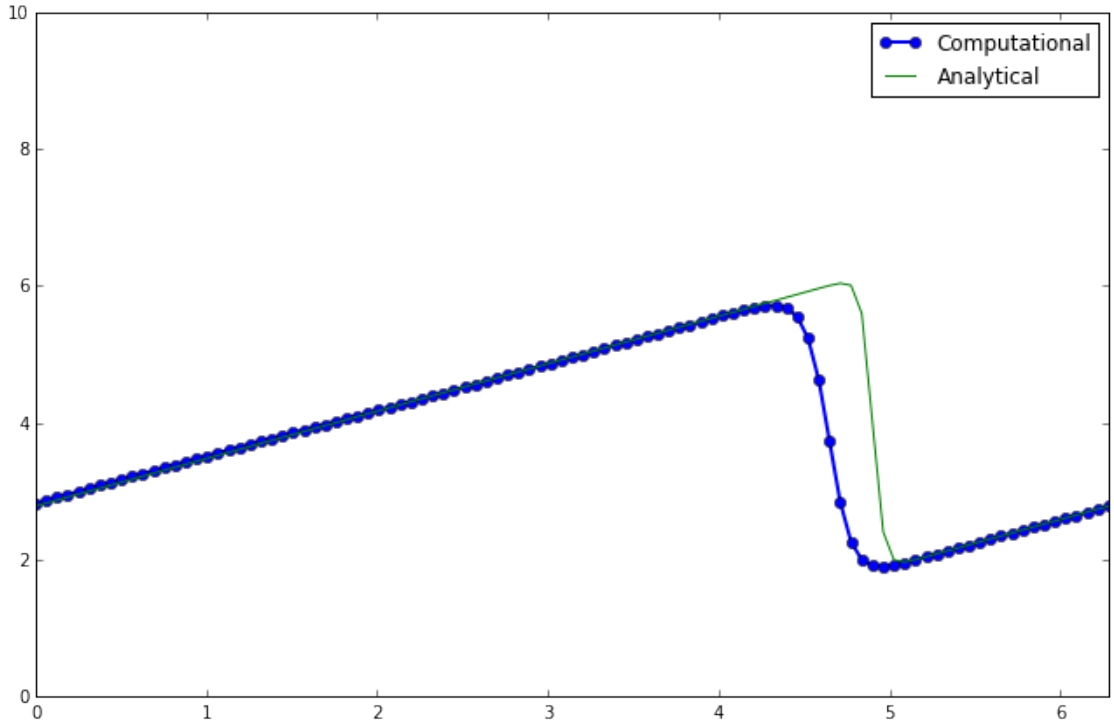
Recall the discretization that we worked out at the beginning of this notebook:

$$u_i^{n+1} = u_i^n - u_i^n \frac{\Delta t}{\Delta x}(u_i^n - u_{i-1}^n) + \nu \frac{\Delta t}{\Delta x^2}(u_{i+1}^n - 2u_i^n + u_{i-1}^n)$$

What does u_{i+1}^n mean when i is already at the end of the frame?
Think about this for a minute before proceeding.

```
In [10]: for n in range(nt):
             un = u.copy()
             for i in range(nx-1):
                 u[i] = un[i] - un[i] * dt/dx *(un[i] - un[i-1]) + nu*dt/dx**2*\
                         (un[i+1]-2*un[i]+un[i-1])
             u[-1] = un[-1] - un[-1] * dt/dx * (un[-1] - un[-2]) + nu*dt/dx**2*\
                         (un[0]-2*un[-1]+un[-2])

         u_analytical = np.asarray([ufunc(nt*dt, xi, nu) for xi in x])
```

```
In [11]: plt.figure(figsize=(11,7), dpi=100)
         plt.plot(x,u, marker='o', lw=2, label='Computational')
         plt.plot(x, u_analytical, label='Analytical')
         plt.xlim([0,2*np.pi])
         plt.ylim([0,10])
         plt.legend();
```



1.2 What next?

The subsequent steps, from 5 to 12, will be in two dimensions. But it is easy to extend the 1D finite-difference formulas to the partial derivatives in 2D or 3D. Just apply the definition — a partial derivative with respect to x is the variation in the x direction while keeping y constant.

Before moving on to Step 5, make sure you have completed your own code for steps 1 through 4 and you have experimented with the parameters and thought about what is happening. Also, we recommend that you take a slight break to learn about array operations with NumPy.

```
In [12]: from IPython.core.display import HTML
         def css_styling():
             styles = open("../styles/custom.css", "r").read()
             return HTML(styles)
         css_styling()
```

```
Out[12]: <IPython.core.display.HTML at 0x7ff51650d950>
```