With modifications by A.Viceré, Urbino University

Space & Time

Introduction to numerical solution of PDEs

Welcome to *Space and Time: Introduction to finite-difference solutions of PDEs*, a course taken from the second module of "Practical Numerical Methods with Python".

In the previous module, we looked into numerical integration methods for the solution of ordinary differential equations (ODEs), using the phugoid model of glider flight as a motivation. In this module, we will study the numerical solution of *partial differential equations (PDEs)*, where the unknown is a multi-variate function. The problem could depend on time, t, and one spatial dimension x (or more), which means we need to build a discretization grid with each independent variable.

We will start our discussion of numerical PDEs with 1-D linear and non-linear convection equations, the 1-D diffusion equation, and 1-D Burgers' equation. We hope you will enjoy them!

1D linear convection

The *one-dimensional linear convection equation* is the simplest, most basic model that can be used to learn something about numerical solution of PDEs. It's surprising that this little equation can teach us so much! Here it is:

$$\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} = 0 \tag{1}$$

The equation represents a *wave* propagating with speed c in the x direction, without change of shape. For that reason, it's sometimes called the *one-way wave equation* (sometimes also the *advection equation*).

With an initial condition $u(x,0) = u_0(x)$, the equation has an exact solution given by:

$$u(x,t) = u_0(x-ct). \tag{2}$$

Go on: check it. Take the time and space derivative and stick them into the equation to see that it holds!

Look at the exact solution for a moment ... we know two things about it:

1. its shape does not change, being always the same as the initial wave, u_0, only shifted in the x-direction; and
2. it's constant along so-called **characteristic curves**, $x - ct =$constant. This means that for any point in space and time, you can move back along the characteristic curve to $t = 0$ to know the value of the solution.

 **Characteristic curves for positive wave speed.**

Why do we call the equations *linear*? PDEs can be either linear or non-linear. In a linear equation, the unknown function u and its derivatives appear only in linear terms, in other words, there are no products, powers, or transcendental functions applied on them.

What is the most important feature of linear equations? Do you remember? In case you forgot: solutions can be superposed to generate new solutions that still satisfy the original equation. This is super useful!

Finite-differences

In the previous lessons, we discretized time derivatives; now we have derivatives in both space *and* time, so we need to discretize with respect to *both* these variables.

Imagine a *space-time* plot, where the coordinates in the vertical direction represent advancing in time—for example, from t^n to t^{n+1}—and the coordinates in the horizontal direction move in space: consecutive points are x_{i-1}, x_i, and x_{i+1}. This creates a grid where a point has both a temporal and spatial index. Here is a graphical representation of the space-time grid:

$$\begin{array}{rcccc} t^{n+1} & \rightarrow & \bullet & \bullet & \bullet \\ t^n & \rightarrow & \bullet & \bullet & \bullet \\ & & x_{i-1} & x_i & x_{i+1} \end{array}$$

For the numerical solution of $u(x,t)$, we'll use subscripts to denote the spatial position, like u_i, and superscripts to denote the temporal instant, like u^n. We would then label the solution at the top-middle point in the grid above as follows: u_i^{n+1}.

Each grid point below has an index i, corresponding to the spatial position and increasing to the right, and an index n, corresponding to the time instant and increasing upwards. A small grid segment would have the following values of the numerical solution at each point:

$$\begin{array}{ccc} \bullet & \bullet & \bullet \\ u_{i-1}^{n+1} & u_i^{n+1} & u_{i+1}^{n+1} \\ \bullet & \bullet & \bullet \\ u_{i-1}^{n} & u_i^{n} & u_{i+1}^{n} \\ \bullet & \bullet & \bullet \\ u_{i-1}^{n-1} & u_i^{n-1} & u_{i+1}^{n-1} \end{array}$$

Another way to explain our discretization grid is to say that it is built with constant steps in time and space, Δt and Δx, as follows:

$$\begin{aligned} x_i &= i\,\Delta x \quad \text{and} \quad t^n = n\,\Delta t \\ u_i^n &= u(i\,\Delta x, n\,\Delta t) \end{aligned} \tag{3}$$

Discretizing our model equation

Let's see how to discretize the 1-D linear convection equation in both space and time. By definition, the partial derivative with respect to time changes only with time and not with space; its discretized form changes only the n indices. Similarly, the partial derivative with respect to x changes with space not time, and only the i indices are affected.

We'll discretize the spatial coordinate x into points indexed from $i = 0$ to N, and then step in discrete time intervals of size Δt.

From the definition of a derivative (and simply removing the limit), we know that for Δx sufficiently small:

$$\frac{\partial u}{\partial x} \approx \frac{u(x+\Delta x) - u(x)}{\Delta x} \tag{4}$$

This formula could be applied at any point x_i. But note that it's not the only way that we can estimate the derivative. The geometrical interpretation of the first derivative $\partial u/\partial x$ at any point is that it represents the slope of the tangent to the curve $u(x)$. In the sketch below, we show a slope line at x_i and mark it as "exact." If the formula written above is applied at x_i, it approximates the derivative using the next spatial grid point: it is then called a *forward difference* formula.

$$\frac{\partial u}{\partial x} \approx \frac{u(x) - u(x-\Delta x)}{\Delta x} \tag{5}$$

But as shown in the sketch below, we could also estimate the spatial derivative using the point behind x_i, in which case it is called a *backward difference*.

$$\frac{\partial u}{\partial x} \approx \frac{u(x+\frac{\Delta x}{2}) - u(x-\frac{\Delta x}{2})}{\Delta x} \tag{6}$$

We could even use the two points on each side of x_i, and obtain what's called a *central*

difference (but in that case the denominator would be $2\Delta x$).

 **Three finite-difference approximations at x_i.**

We have three possible ways to represent a discrete form of $\partial u/\partial x$:

- Forward difference: uses x_i and $x_i + \Delta x$,
- Backward difference: uses x_i and $x_i - \Delta x$,
- Central difference: uses two points on either side of x_i.

The sketch above also suggests that some finite-difference formulas might be better than others: it looks like the *central difference* approximation is closer to the slope of the "exact" derivative. Curious if this is just an effect of our exaggerated picture? We have in fact already seen that the central difference is "better", and we'll discuss later this in more rigorous terms.

The three formulas are:

$$\frac{\partial u}{\partial x} \approx \frac{u(x_{i+1}) - u(x_i)}{\Delta x} \quad \text{Forward} \tag{7}$$

$$\frac{\partial u}{\partial x} \approx \frac{u(x_i) - u(x_{i-1})}{\Delta x} \quad \text{Backward} \tag{8}$$

$$\frac{\partial u}{\partial x} \approx \frac{u(x_{i+1}) - u(x_{i-1})}{2\Delta x} \quad \text{Central} \tag{9}$$

Euler's method is equivalent to using a forward-difference scheme for the time derivative. Let's stick with that, and choose the backward-difference scheme for the space derivative. Our discrete equation is then:

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} + c\frac{u_i^n - u_{i-1}^n}{\Delta x} = 0, \tag{10}$$

where n and $n+1$ are two consecutive steps in time, while $i-1$ and i are two neighboring points of the discretized x coordinate. With given initial conditions, the only unknown in this discretization is u_i^{n+1}. We solve for this unknown to get an equation that lets us step in time, as follows:

$$u_i^{n+1} = u_i^n - c\frac{\Delta t}{\Delta x}(u_i^n - u_{i-1}^n) \tag{11}$$

We like to make drawings of a grid segment, showing the grid points that influence our numerical solution. This is called a **stencil**. Below is the stencil for solving our model equation with the finite-difference formula we wrote above.

 **Stencil for the "forward-time/backward-space" scheme.**

And compute!

Alright. Let's get a little Python on the road. First: we need to load our array and plotting libraries, as usual. And if you noticed in the Lesson 02.04, we taught you a neat trick to set some global plotting parameters with the `rcParams` module. We like to do that.

In [ ]:
```
import numpy
from matplotlib import pyplot
%matplotlib inline
from matplotlib import rcParams
rcParams['font.family'] = 'serif'
rcParams['font.size'] = 16
```

As a first exercise, we'll solve the 1D linear convection equation with a *square wave* initial condition, defined as follows:

$$u(x,0) = \begin{cases} 2 & \text{where } 0.5 \leq x \leq 1, \\ 1 & \text{everywhere else in } (0,2) \end{cases} \tag{12}$$

We also need a boundary condition on x: let $u = 1$ at $x = 0$. Our spatial domain for the numerical solution will only cover the range $x \in (0,2)$.

 **Square wave initial condition.**

Now let's define a few variables; we want to make an evenly spaced grid of points within our spatial domain. In the code below, we define a variable called `nx` that will be the number of spatial grid points, and a variable `dx` that will be the distance between any pair of adjacent grid points. We also can define a step in time, `dt`, a number of steps, `nt`, and a value for the wave speed: we like to keep things simple and make $c = 1$.

In [ ]:
```
nx = 41  # try changing this number from 41 to 81 and Run All ... what happens?
dx = 2./(nx-1)
nt = 25
dt = .02
c = 1      # assume wavespeed of c = 1
x = numpy.linspace(0,2,nx)
```

In [ ]:
```
print(dx)
```

```
0.05
```

We also need to set up our initial conditions. Here, we use the NumPy function `ones()` defining an array which is `nx` elements long with every value equal to 1. How useful! We then *change a slice* of that array to the value $u = 2$, to get the square wave, and we print out the initial array just to admire it. But which values should we change? The problem states that we need to change the indices of `u` such that the square wave begins at $x = 0.5$ and ends at $x = 1$.

We can use the `numpy.where` function to return a list of indices where the vector x meets (or doesn't meet) some condition.

In [ ]:
```
u = numpy.ones(nx)      #numpy function ones()

lbound = numpy.where( x >= 0.5 )
ubound = numpy.where( x <= 1.0 )
print(lbound)
print(ubound)
```

```
(array([10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26,
       27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40]),)
(array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19, 20]),)
```

That leaves us with two vectors. `lbound` , which has the indices for $x \geq .5$ and 'ubound', which has the indices for $x \leq 1$. To combine these two, we can use an intersection, with `numpy.intersect1d` .

In [ ]:
```
bounds = numpy.intersect1d(lbound, ubound)
u[bounds]=2  #setting u = 2 between 0.5 and 1 as per our I.C.s
print(u)
```

```
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
```

Remember that Python can also combine commands, we could have instead written

```
u[numpy.intersect1d(numpy.where(x >= 0.5), numpy.where(x <= 1))] = 2
```

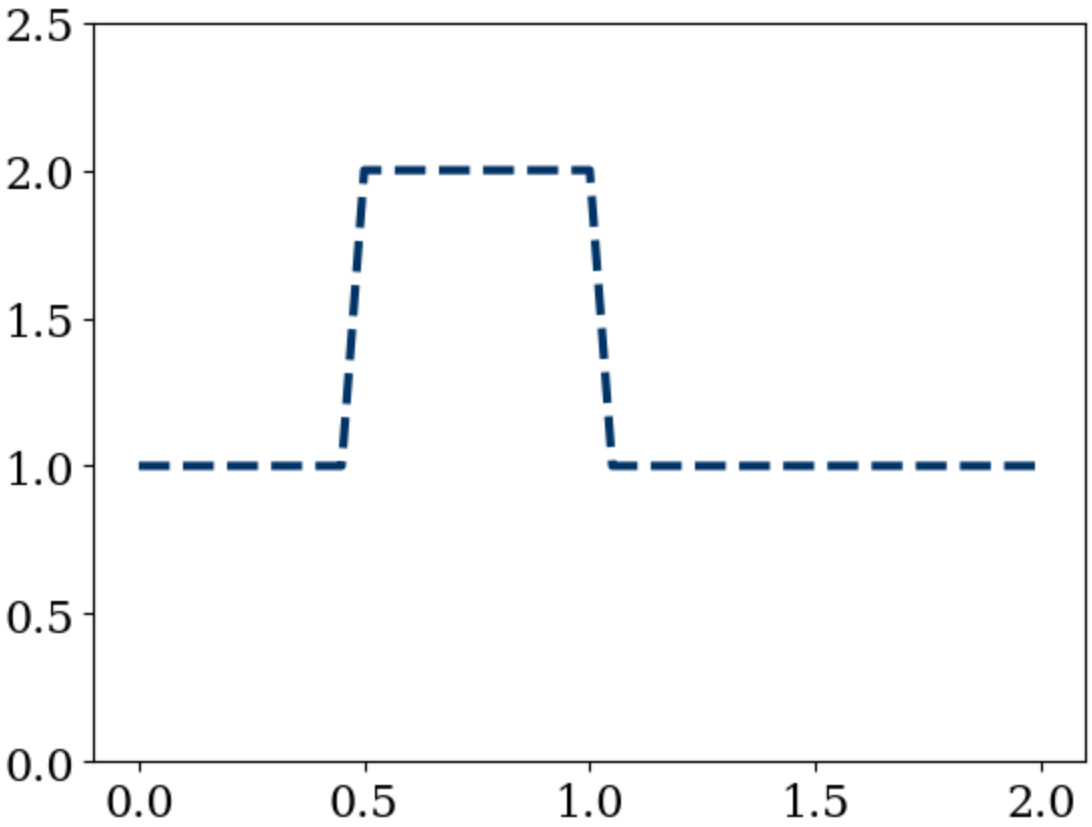
Even more compactly, one could have combined the boolean conditions inside the `numpy.where` call

In [ ]:
```
u = numpy.ones(nx)      #numpy function ones()
u[numpy.where((x >= 0.5) & (x <= 1))] = 2
print(u)
```

```
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
```

Now let's take a look at those initial conditions we've built with a handy plot.

In [ ]:
```
pyplot.plot(x, u, color='#003366', ls='--', lw=3)
pyplot.ylim(0,2.5);
```



It does look pretty close to what we expected. But it looks like the sides of the square wave are not perfectly vertical. Is that right? Think for a bit.

Now it's time to write some code for the discrete form of the convection equation using our chosen finite-difference scheme.

For every element of our array `u`, we need to perform the operation:

$$u_i^{n+1} = u_i^n - c\frac{\Delta t}{\Delta x}(u_i^n - u_{i-1}^n) \tag{13}$$

We'll store the result in a new (temporary) array `un`, which will be the solution u for the next time-step. We will repeat this operation for as many time-steps as we specify and then we can see how far the wave has traveled.

We first initialize the placeholder array `un` to hold the values we calculate for the $n+1$ timestep, using once again the NumPy function `ones()`.

Then, we may think we have two iterative operations: one in space and one in time (we'll learn differently later), so we may start by nesting a spatial loop inside the time loop, as shown below. You see that the code for the finite-difference scheme is a direct expression of the discrete equation:

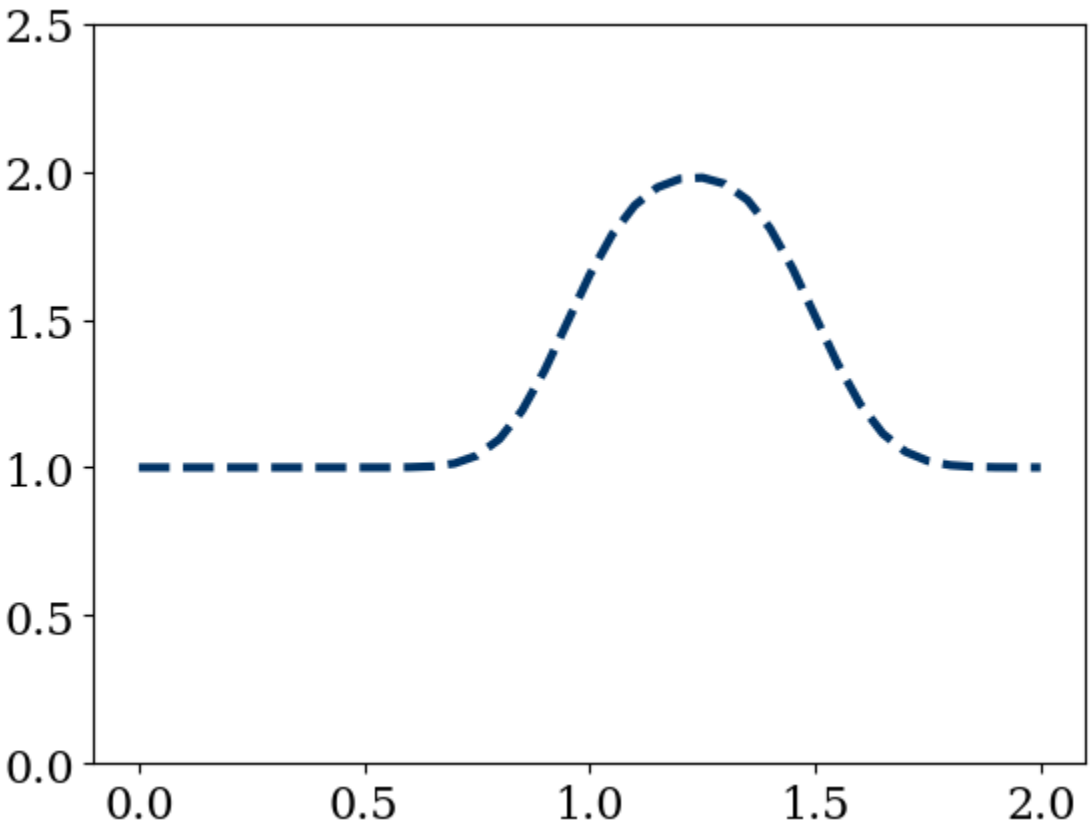
In [ ]:
```
for n in range(1,nt):
    un = u.copy()
    for i in range(1,nx):
```

```
        u[i] = un[i]-c*dt/dx*(un[i]-un[i-1])
```

Note—We will learn later that the code as written above is *quite inefficient*, and there are better ways to write this, Python-style. But let's carry on.

Now let's inspect our solution array after advancing in time with a line plot.

In [ ]:
```
pyplot.plot(x, u, color='#003366', ls='--', lw=3)
pyplot.ylim(0,2.5);
```



That's funny. Our square wave has definitely moved to the right, but it's no longer in the shape of a top-hat. **What's going on?**

Dig deeper

The solution differs from the expected square wave because the discretized equation is an approximation of the continuous differential equation that we want to solve. There are errors: we knew that. But the modified shape of the initial wave is something curious. Maybe it can be improved by making the grid spacing finer. Why don't you try it? Does it help?

Spatial truncation error

Recall the finite-difference approximation we are using for the spatial derivative:

$$\frac{\partial u}{\partial x} \approx \frac{u(x+\Delta x) - u(x)}{\Delta x} \tag{14}$$

We obtain it by using the definition of the derivative at a point, and simply removing the limit, in the assumption that Δx is very small. But we already learned with Euler's method that this introduces an error, called the *truncation error*.

Using a Taylor series expansion for the spatial terms now, we see that the backward-difference scheme produces a first-order method, in space.

$$\frac{\partial u}{\partial x}(x_i) = \frac{u(x_i) - u(x_{i-1})}{\Delta x} + \frac{\Delta x}{2}\frac{\partial^2 u}{\partial x^2}(x_i) - \frac{\Delta x^2}{6}\frac{\partial^3 u}{\partial x^3}(x_i) + \cdots \tag{15}$$

The dominant term that is neglected in the finite-difference approximation is of $\mathcal{O}(\Delta x)$. We also see that the approximation *converges* to the exact derivative as $\Delta x \to 0$. That's good news!

In summary, the chosen "forward-time/backward space" difference scheme is first-order in both space and time: the truncation errors are $\mathcal{O}(\Delta t, \Delta x)$. We'll come back to this!

Non-linear convection

Let's move on to the non-linear convection equation, using the same methods as before. The 1-D convection equation is:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = 0 \tag{16}$$

The only difference with the linear case is that we've replaced the constant wave speed c by the variable speed u. The equation is non-linear because now we have a product of the solution and one of its derivatives: the product $u\,\partial u/\partial x$. This changes everything!

We're going to use the same discretization as for linear convection: forward difference in time and backward difference in space. Here is the discretized equation:

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} + u_i^n\frac{u_i^n - u_{i-1}^n}{\Delta x} = 0 \tag{17}$$

Solving for the only unknown term, u_i^{n+1}, gives an equation that can be used to advance in time:

$$u_i^{n+1} = u_i^n - u_i^n\frac{\Delta t}{\Delta x}(u_i^n - u_{i-1}^n) \tag{18}$$

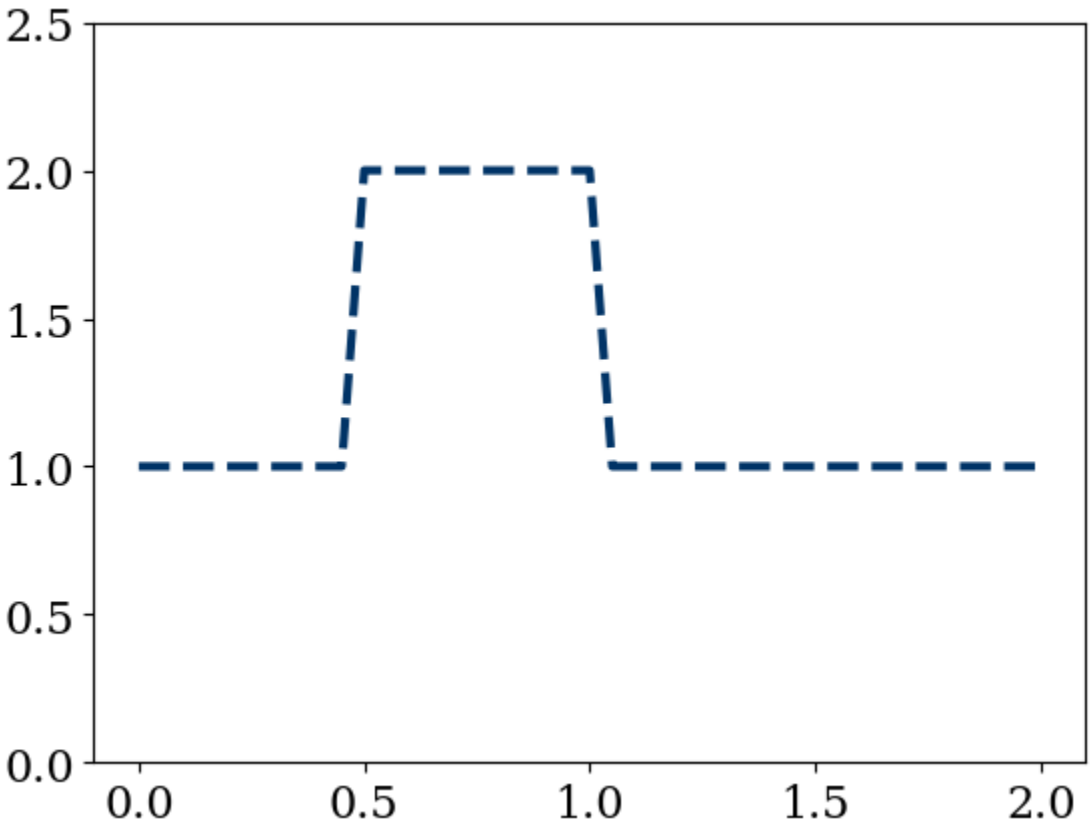
There is very little that needs to change from the code written so far. In fact, we'll even use the same square-wave initial condition. But let's re-initialize the variable `u` with the initial values, and re-enter the numerical parameters here, for convenience (we no longer need c, though).

In [ ]:
```
##problem parameters
nx = 41
dx = 2./(nx-1)
nt = 10
dt = .02

##initial conditions
u = numpy.ones(nx)
u[numpy.where((x >= 0.5) & (x <= 1))] = 2
```

How does it look?

In [ ]:
```
pyplot.plot(x, u, color='#003366', ls='--', lw=3)
pyplot.ylim(0,2.5);
```



Changing just one line of code in the solution of linear convection, we are able to now get the non-linear solution: the line that corresponds to the discrete equation now has `un[i]` in the place where before we just had `c`. So you could write something like:

```
for n in range(1,nt):
  un = u.copy()
  for i in range(1,nx):
    u[i] = un[i]-un[i]*dt/dx*(un[i]-un[i-1])
```

We're going to be more clever than that and use NumPy to update all values of the spatial grid in one fell swoop. We don't really need to write a line of code that gets executed *for each* value of u on the spatial grid. Python can update them all at once! Study the code below, and compare it with the one above. Here is a helpful sketch, to illustrate the array

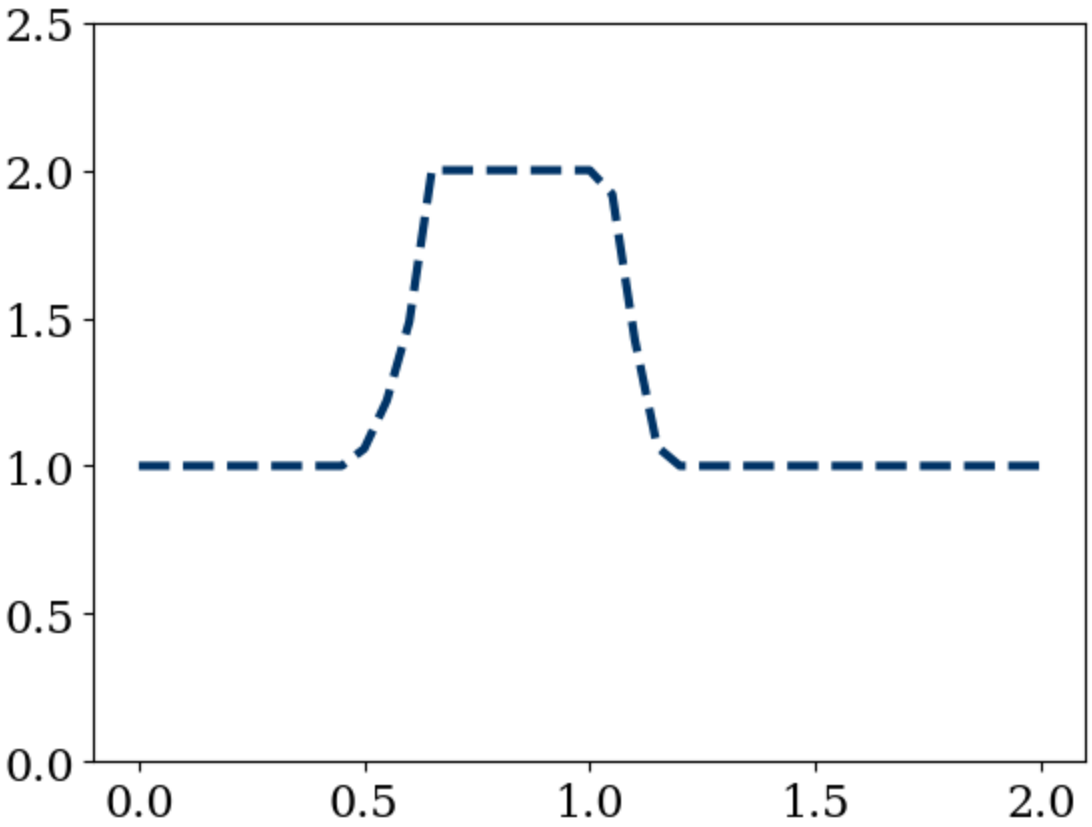
operation—also called a "vectorized" operation—for $u_i - u_{i-1}$.

 **Sketch to explain vectorized stencil operation. Adapted from "Indices point between elements" by Nelson Elhage.**

In [ ]:
```
for n in range(1, 4):
    un = u.copy()
    print("A ", un[5:])
    u[1:] = un[1:]-un[1:]*dt/dx*(un[1:]-un[0:-1])
    u[0] = 1.0
    print("B ", u[5:])
```

```
A  [1. 1. 1. 1. 1. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
B  [1.  1.  1.  1.  1.  1.2 2.  2.  2.  2.  2.  2.  2.  2.  2.  2.  1.4 1.
 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1. ]
A  [1.  1.  1.  1.  1.  1.2 2.  2.  2.  2.  2.  2.  2.  2.  2.  2.  1.4 1.
 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1. ]
B  [1.    1.    1.    1.    1.    1.104 1.36  2.    2.    2.    2.    2.
 2.    2.    2.    2.    1.736 1.16  1.    1.    1.    1.    1.    1.
 1.    1.    1.    1.    1.    1.    1.    1.    1.    1.    1.    1.   ]
A  [1.    1.    1.    1.    1.    1.104 1.36  2.    2.    2.    2.    2.
 2.    2.    2.    2.    1.736 1.16  1.    1.    1.    1.    1.    1.
 1.    1.    1.    1.    1.    1.    1.    1.    1.    1.    1.    1.   ]
B  [1.        1.        1.        1.        1.        1.0580736 1.220736
 1.488     2.        2.        2.        2.        2.        2.
 2.        2.        1.9193216 1.427264  1.064     1.        1.
 1.        1.        1.        1.        1.        1.        1.
 1.        1.        1.        1.        1.        1.        1.
 1.       ]
```

In [ ]:
```
pyplot.plot(x, u, color='#003366', ls='--', lw=3)
pyplot.ylim(0,2.5);
```



Hmm. That's quite interesting: like in the linear case, we see that we have lost the sharp sides of our initial square wave, but there's more. Now, the wave has also lost symmetry! It seems to be lagging on the rear side, while the front of the wave is steepening. Is this another form of numerical error, do you ask? No! It's physics!

Dig deeper

Think about the effect of having replaced the constant wave speed c by the variable speed given by the solution u. It means that different parts of the wave move at different speeds. Make a sketch of an initial wave and think about where the speed is higher and where it is lower ...

References

- Elhage, Nelson (2015), "Indices point between elements"