Space & Time

Stability and the CFL condition

Welcome back! This is the second IPython Notebook of the series *Space and Time — Introduction to Finite-difference solutions of PDEs*, the second module of "Practical Numerical Methods with Python" (http://openedx.seas.gwu.edu/courses/GW/MAE6286/2014_fall/about).

In the first lesson of this series, we studied the numerical solution of the linear and non-linear convection equations, using the finite-difference method. Did you experiment there using different parameter choices? If you did, you probably ran into some unexpected behavior. Did your solution ever blow up (sometimes in a cool way!)?

In this IPython Notebook, we will explore why changing the discretization parameters can affect your solution in such a drastic way.

With the solution parameters we initially suggested, the spatial grid had 41 points and the timestep was 0.25. Now, we're going to experiment with the number of points in the grid. The code below corresponds to the linear convection case, but written into a function so that we can easily examine what happens as we adjust just one variable: **the grid size**.

In [1]:
```
import numpy
import matplotlib.pyplot as plt
%matplotlib inline
from matplotlib import rcParams
rcParams['font.family'] = 'serif'
```

```
rcParams['font.size'] = 16
```

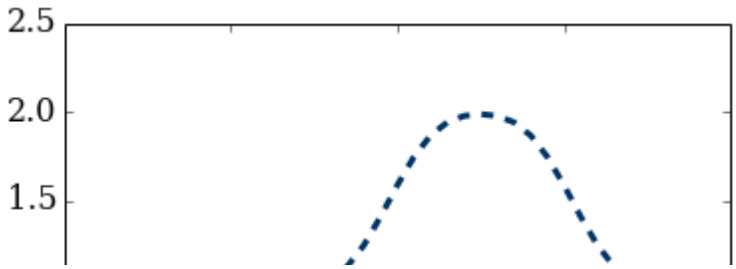
In [2]:

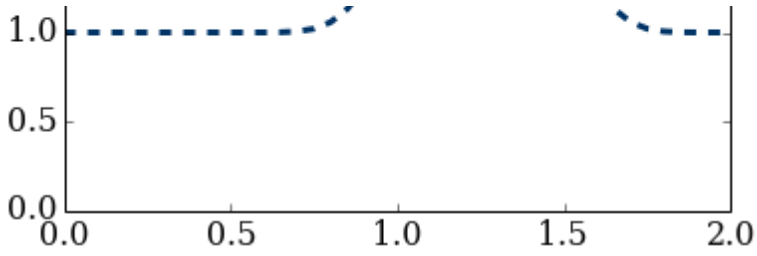
```
def linearconv(nx):
    """Solve the linear convection equation.

    Solves the equation d_t u + c d_x u = 0 where
    * the wavespeed c is set to 1
    * the domain is x \in [0, 2]
    * 20 timesteps are taken, with \Delta t = 0.025
    * the initial data is the hat function

    Produces a plot of the results

    Parameters
    ----------

    nx : integer
        number of internal grid points

    Returns
    -------

    None : none
    """
    dx = 2./(nx-1)
    nt = 20
    dt = .025
    c = 1

    u = numpy.ones(nx)
    u[.5/dx : 1/dx+1]=2

    un = numpy.ones(nx)

    for n in range(nt):
        un = u.copy()
        u[1:] = un[1:] -c*dt/dx*(un[1:] -un[0:-1])
        u[0] = 1.0


    plt.plot(numpy.linspace(0,2,nx), u, color='#003366', ls='--', lw=3)
    plt.ylim(0,2.5);
```

Now let's examine the results of the linear convection problem with an increasingly fine mesh. We'll try 41, 61 and 71 points ... then we'll shoot for 85. See what happens:

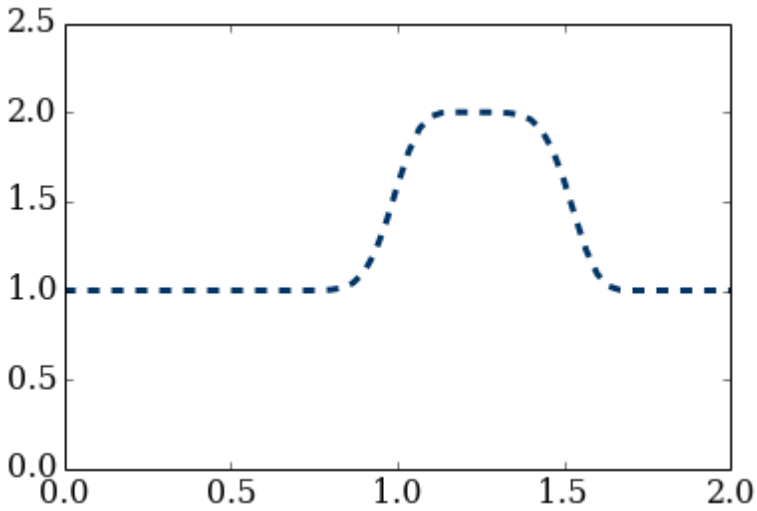
In [3]:

```
linearconv(41) #convection using 41 grid points
```

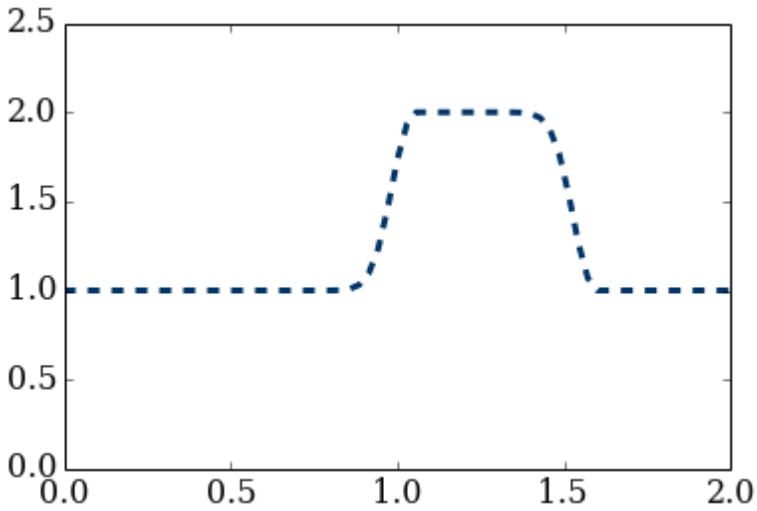




In [4]:
```
linearconv(61)
```

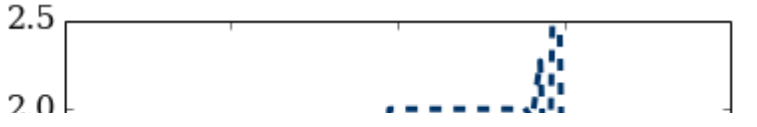


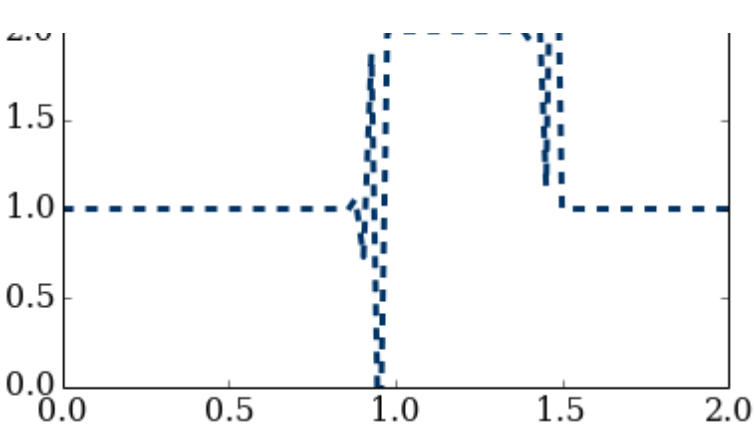
In [5]:
```
linearconv(71)
```



So far so good—as we refine the spatial grid, the wave is more square, indicating that the discretization error is getting smaller. But what happens when we refine the grid even further? Let's try 85 grid points.

In [6]:
```
linearconv(85)
```





Oops. This doesn't look anything like our original hat function. Something has gone awry. It's the same code that we ran each time, so it's not a bug!

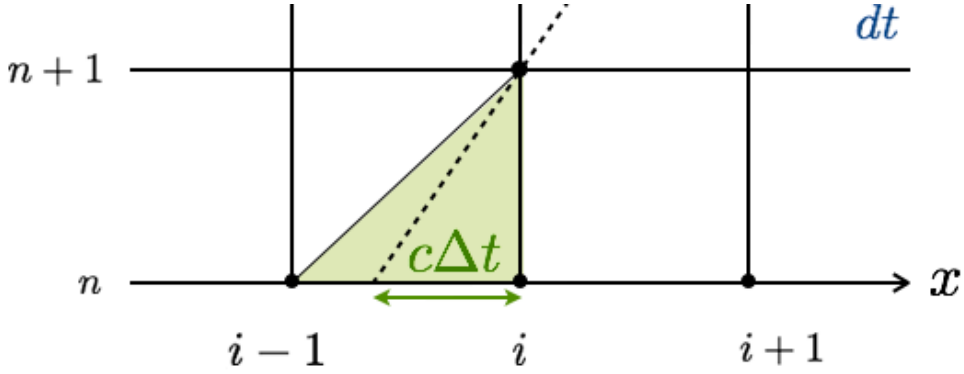
What happened?

To answer that question, we have to think a little bit about what we're actually implementing in code when we solve the linear convection equation with the forward-time/backward-space method.

In each iteration of the time loop, we use the existing data about the solution at time n to compute the solution in the subsequent time step, $n+1$. In the first few cases, the increase in the number of grid points returned more accurate results. There was less discretization error and the translating wave looked more like a square wave than it did in our first example.

Each iteration of the time loop advances the solution by a time-step of length Δt, which had the value 0.025 in the examples above. During this iteration, we evaluate the solution u at each of the x_i points on the grid. But in the last plot, something has clearly gone wrong.

What has happened is that over the time period Δt, the wave is travelling a distance which is greater than dx, and we say that the solution becomes *unstable* in this situation. The length dx of grid spacing is inversely proportional to the number of total points nx: we asked for more grid points, so dx got smaller. Once dx got smaller than the $c\Delta t$—the distance travelled by the numerical solution in one time step—it's no longer possible for the numerical scheme to solve the equation correctly!



Graphical interpretation of the CFL condition.

Consider the illustration above. When the distance $c\Delta t$, covered by the numerical solution in one time step, is smaller than Δx, then the information about the solution in the previous step is contained within the green triangle, representing the *domain of dependence* of the numerical scheme.

If Δx is smaller than $c\Delta t$, then the information about the solution needed for u_i^{n+1} is not available in the numerical scheme, because the characteristic line traced from the grid coordinate $i, n+1$ lands *behind* the point $i-1$ on the grid. But information on the solution there is not available from the numerical scheme!

The following condition ensures that the domain of dependence of the differential equation should be contained in the *numerical* domain of dependence:

$$\sigma = \frac{c\Delta t}{\Delta x} \leq 1 \tag{1}$$

Stability of the numerical solution can be enforced if the step size `dt` is calculated with respect to the size of `dx` to satisfy the condition above.

The value of $c\Delta t/\Delta x$ is called the **Courant-Friedrichs-Lewy number** (CFL number), often denoted by σ. The value $\sigma_{\max}$ that will ensure stability depends on the discretization used; for the forward-time/backward-space scheme, the condition for stability is $\sigma < 1$.

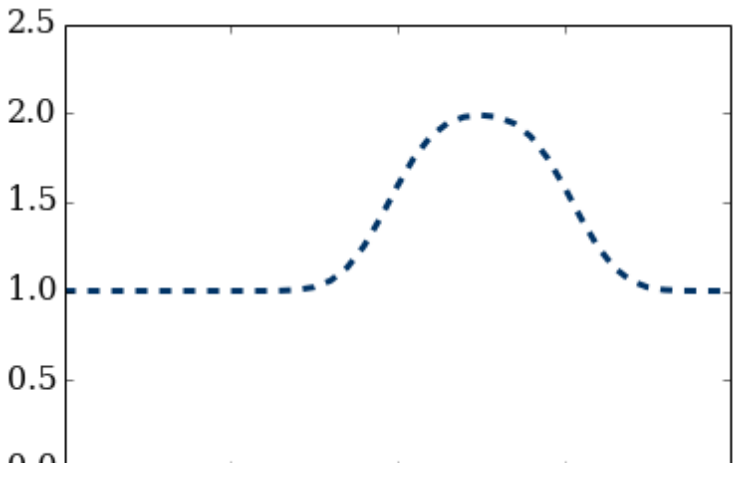
In a new version of our code—written *defensively*—, we'll use the CFL number to calculate the appropriate time-step `dt` depending on the size of `dx`.

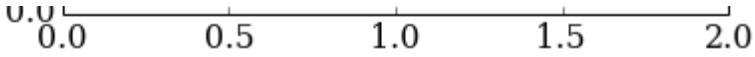
In [7]:
```
def linearconv(nx):
    """Solve the linear convection equation.

    Solves the equation d_t u + c d_x u = 0 where
```

```
    * the wavespeed c is set to 1
    * the domain is x \in [0, 2]
    * 20 timesteps are taken, with \Delta t computed using the CFL 0.5
    * the initial data is the hat function

    Produces a plot of the results

    Parameters
    ----------

    nx : integer
        number of internal grid points

    Returns
    -------

    None : none
    """
    dx = 2./(nx-1)
    nt = 20
    c = 1
    sigma = .5

    dt = sigma*dx

    u = numpy.ones(nx)
    u[.5/dx : 1/dx+1]=2

    un = numpy.ones(nx)

    for n in range(nt):
        un = u.copy()
        u[1:] = un[1:] -c*dt/dx*(un[1:] -un[0:-1])
        u[0] = 1.0


    plt.plot(numpy.linspace(0,2,nx), u, color='#003366', ls='--', lw=3)
    plt.ylim(0,2.5);
```

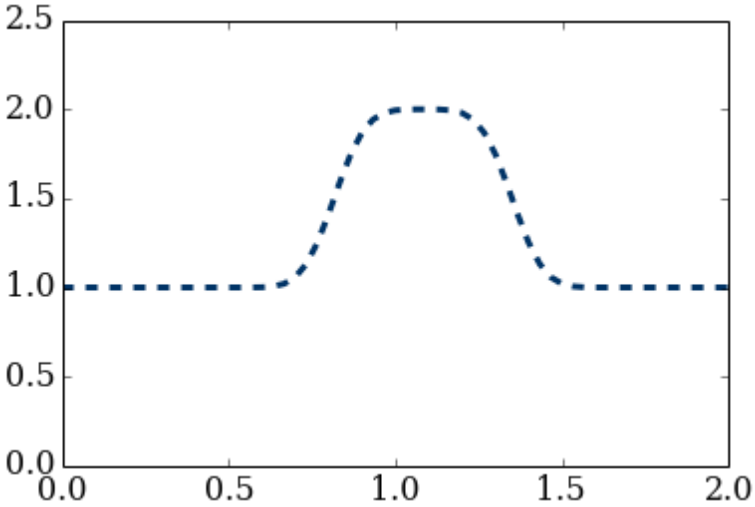
Now, it doesn't matter how many points we use for the spatial grid: the solution will always be stable!

In [8]:
```
linearconv(41)
```

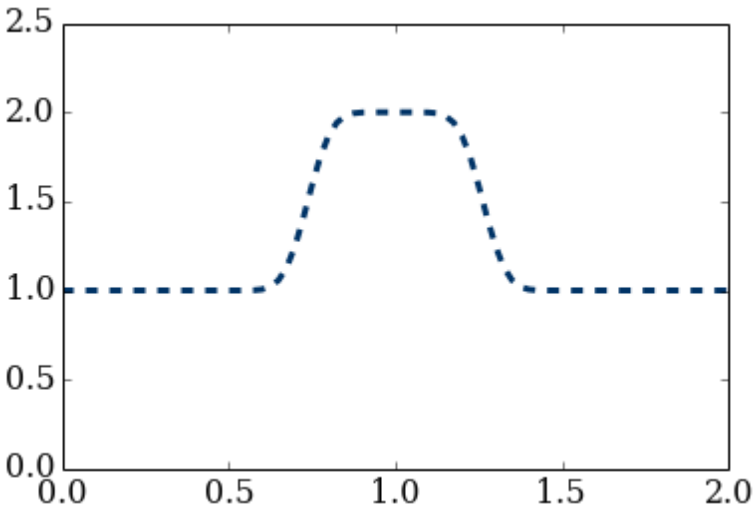




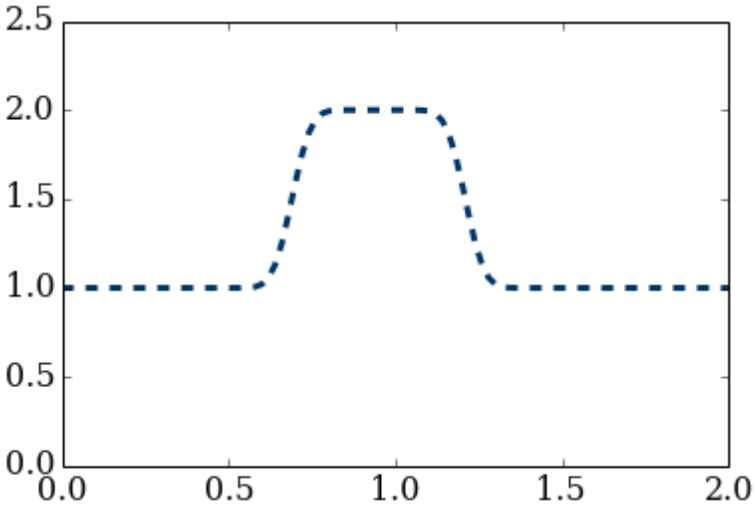
In [9]:
```
linearconv(61)
```



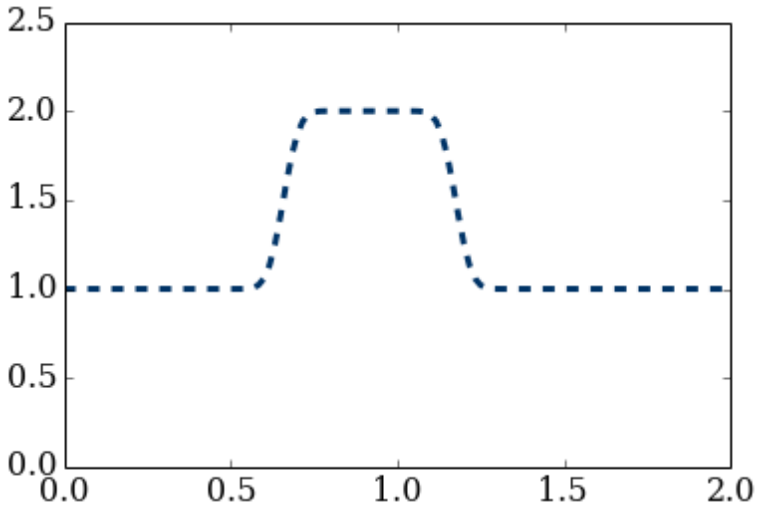
In [10]:
```
linearconv(81)
```



In [11]:
```
linearconv(101)
```



In [12]:
```
linearconv(121)
```



Notice that as the number of points `nx` increases, the wave convects a shorter and shorter distance. The number of time iterations we have advanced the solution to is held constant at `nt = 20`, but depending on the value of `nx` and the corresponding values of `dx` and `dt`, a shorter time window is being examined overall.

The cell below loads the style of the notebook.

In [13]:
```
from IPython.core.display import HTML
css_file = '../../styles/numericalmoocstyle.css'
HTML(open(css_file, "r").read())
```

Out[13]: