First steps in Computational Fluid Dynamics

September 28, 2017

Chapter 1

Introduction

1.1 Short note by BCAM CFD group

These notes are an adaptation done by the BCAM CFD group for the short course on Computational Fluid Dynamics held at the University of the Basque Country UPV/EHU Faculty of Engineering in the framework of the Formula Student Bizkaia group.

Inside the exercises, some important points have been marked by **[this symbol]** , meaning that at each of those point, some useful remarks are made during the exercise. At the end of each exercise, the remarks are wrapped up in a summary section.

1.2 12 steps to Navier-Stokes

Hello! Welcome to the **12 steps to Navier-Stokes**. This is a practical module that is used in the beginning of an interactive Computational Fluid Dynamics (CFD) course taught by Prof. Lorena Barba since Spring 2009 at Boston University. The course assumes only basic programming knowledge (in any language) and of course some foundation in partial differential equations and fluid mechanics. The practical module was inspired by the ideas of Dr. Rio Yokota, who was a post-doc in Barba's lab, and has been refined by Prof. Barba and her students over several semesters teaching the course. The course is taught entirely using Python and students who don't know Python just learn as we work through the module.

This IPython notebook will lead you through the first step of programming your own Navier-Stokes solver in Python from the ground up. We're going to dive right in. Don't worry if you don't understand everything that's happening at first, we'll cover it in detail as we move forward and you can support your learning with the videos of Prof. Barba's lectures on YouTube.

For best results, after you follow this notebook, prepare your own code for Step 1, either as a Python script or in a clean IPython notebook.

To execute this Notebook, we assume you have invoked the notebook server using: `ipython notebook`.

Chapter 2

1D problems

2.1 Step 1: 1-D Linear Convection

The 1-D Linear Convection equation is the simplest, most basic model that can be used to learn something about CFD. It is surprising that this little equation can teach us so much! Here it is:**[Differential equation]**

$$\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} = 0 \tag{2.1}$$

With given initial conditions (understood as a *wave*), the equation represents the propagation of that initial *wave* with speed c, without change of shape. Let the initial condition be

$$(u(x,0) = u_0(x)$$

Then the exact solution of the equation is

$$u(x,t) = u_0(x - ct)$$

We discretize this equation in both space and time, using the Forward Difference scheme for the time derivative and the Backward Difference scheme for the space derivative. Consider discretizing the spatial coordinate x into points that we index from $i = 0$ to N, and stepping in discrete time intervals of size Δt.

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

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

Our discrete equation, then, is: **[Discretization]**

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

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. If there are given initial conditions, then the only unknown in this discretization is u_i^{n+1}. We can solve for our unknown to get an equation that allows us to advance in time, as follows:

$$u_i^{n+1} = u_i^n - c\frac{\Delta t}{\Delta x}(u_i^n - u_{i-1}^n) \tag{2.2}$$

Now let's try implementing this in Python.

We'll start by importing a few libraries to help us out.

- `numpy` is a library that provides a bunch of useful matrix operations akin to MATLAB
- `matplotlib` is a 2D plotting library that we will use to plot our results
- `time` and `sys` provide basic timing functions that we'll use to slow down animations for viewing

```
In [1]: # Remember: comments in python are denoted by the pound sign
        import numpy                         #here we load numpy
        from matplotlib import pyplot        #here we load matplotlib
        import time, sys                     #and load some utilities
```

```
In [2]: #this makes matplotlib plots appear in the notebook (instead of a separate window)
        %matplotlib inline
```

Now let's define a few variables; we want to define an evenly spaced grid of points within a spatial domain that is 2 units of length wide, i.e., $x_i \in (0, 2)$.

[Discretization] We'll define a variable `nx`, which will be the number of grid points we want and `dx` will be the distance between any pair of adjacent grid points.

```
In [3]: nx = 41  # try changing this number from 41 to 81 and Run All ... what happens?
        dx = 2 / (nx-1)
        nt = 25    #nt is the number of timesteps we want to calculate
        dt = .025  #dt is the amount of time each timestep covers (delta t)
        c = 1      #assume wavespeed of c = 1
```

[Stability (CFL)]

```
CFL = c*dt/dx
        print(CFL)
0.5
```

We also need to set up our initial conditions. The initial velocity u_0 is given as $u = 2$ in the interval $0.5 \leq x \leq 1$ and $u = 1$ everywhere else in $(0, 2)$ (i.e., a hat function).

[Initial condition] Here, we use the function `ones()` defining a `numpy` array which is `nx` elements long with every value equal to 1.

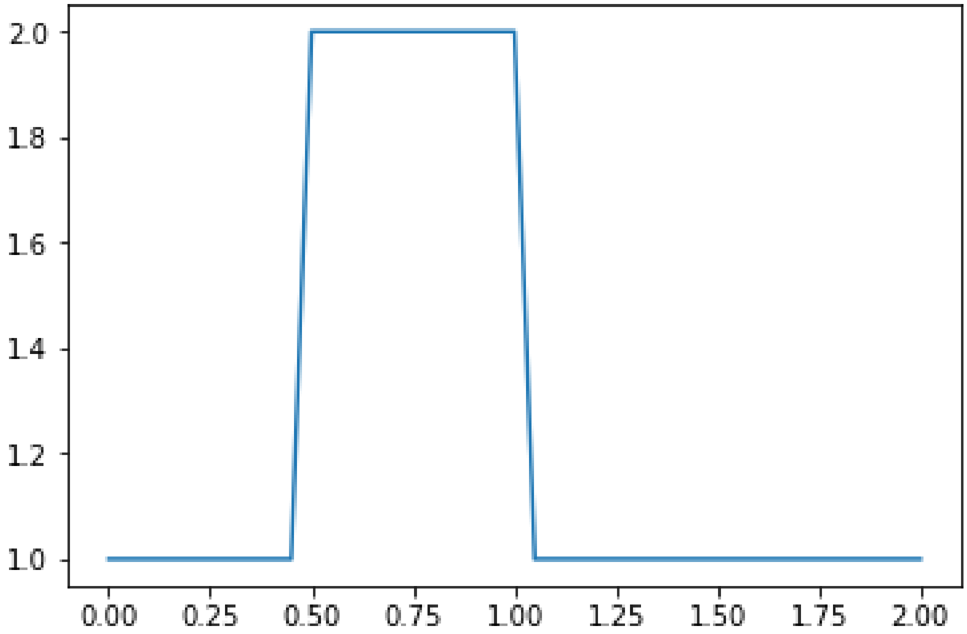
```
In [4]: u = numpy.ones(nx)      #numpy function ones()
        u[int(.5 / dx):int(1 / dx + 1)] = 2  #setting u = 2 between 0.5 and 1 as per our I.C.s
        print(u)
```

```
[ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  2.  2.  2.  2.  2.  2.  2.  2.
  2.  2.  2.  1.  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 using a Matplotlib plot. We've imported the `matplotlib` plotting library `pyplot` and the plotting function is called `plot`, so we'll call `pyplot.plot`. To learn about the myriad possibilities of Matplotlib, explore the Gallery of example plots.

Here, we use the syntax for a simple 2D plot: `plot(x,y)`, where the `x` values are evenly distributed grid points:

```
In [5]: pyplot.plot(numpy.linspace(0, 2, nx), u);
```



[Discretization] Why doesn't the hat function have perfectly straight sides? Think for a bit.

Now it's time to implement the discretization of the convection equation using a finite-difference scheme.

For every element of our array `u`, we need to perform the operation $u_i^{n+1} = u_i^n - c\frac{\Delta t}{\Delta x}(u_i^n - u_{i-1}^n)$

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 convected.

We first initialize our 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'll start by nesting one loop inside the other. Note the use of the nifty `range()` function. When we write: `for i in range(1,nx)` we will iterate through the `u` array, but we'll be skipping the first element (the zero-th element). *Why?*

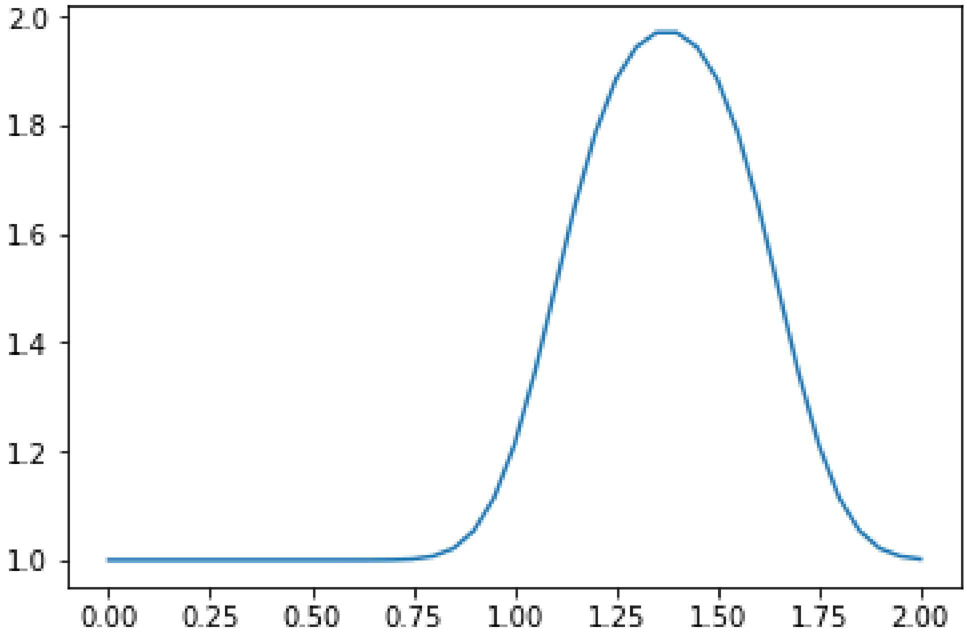
[Boundary condition]

```
In [6]: un = numpy.ones(nx) #initialize a temporary array

        for n in range(nt):  #loop for values of n from 0 to nt, so it will run nt times
```

```
un = u.copy() ##copy the existing values of u into un
for i in range(1, nx): ## you can try commenting this line and...
#for i in range(nx): ## ... uncommenting this line and see what happens!
    u[i] = un[i] - c * dt / dx * (un[i] - un[i-1])
```

Now let's try plotting our u array after advancing in time.

```
In [7]: pyplot.plot(numpy.linspace(0, 2, nx), u);
```



OK! So our hat function has definitely moved to the right, but it's no longer a hat. **What's going on?**

2.1.1 Remarks for this exercise

In this simple example we can already comment about:

1. **[Differential equation]** So far, we just assume that we are *given* a (partial differential) equation that describes some physical phenomena (in this specific case of the governing equation (2.1), the transport of the quantity u). We are then interested in solving this equation, and since in principle it is difficult to find an *analytical* (i.e. exact) solution, we try to find an *approximate* solution using the *numerical approach*.

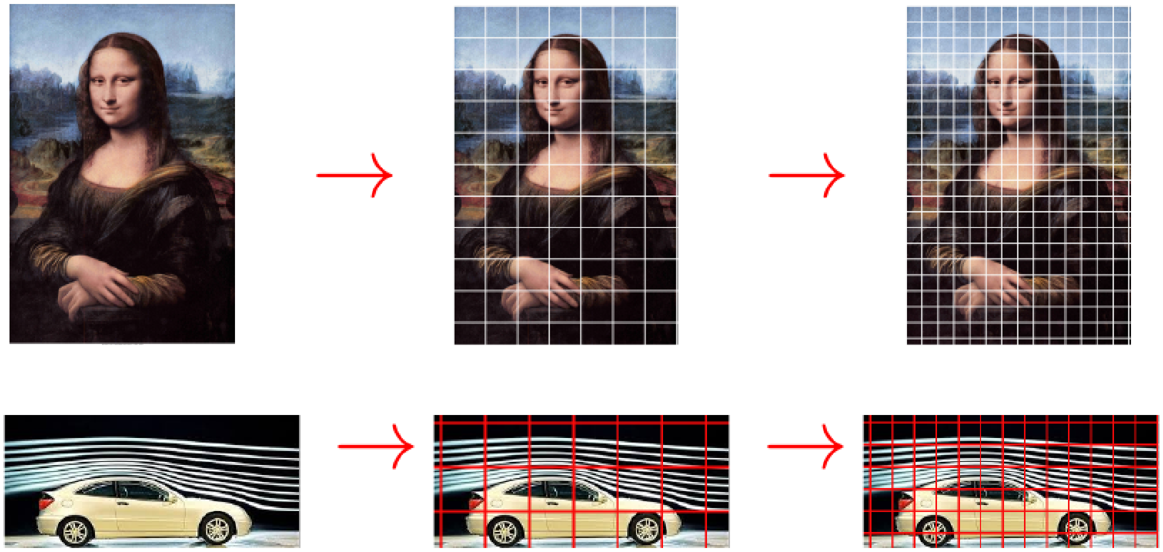
2. **[Discretization]** This concept is strictly related to the above search for an approximate solution. Here we have the transition from *continuum to discrete* domain (mathematically, from infinite to finite dimensional). That is, in order to use a computer, we need to transform our problem (2.1) into the following form:

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \tag{2.3}$$

where **A** is a *matrix*, **x** is the vector of the *unknowns* that we are interested in, and **b** is the vector of *known* values. Just for understanding this, could you recognize in the discretized equation above, how that equation (2.2) translates to form (2.3)?

Other examples in real life may be the analogical picture and the digital picture, or try to express the temperature in a room by having a finite number of measures of it.

Or think of the elementary school when you had to reproduce a painting: you might have draw a *grid* to help you divide the whole big picture into smaller pieces, that are somehow "easier" to handle. The same happens with fluids: we cannot know where each molecule of fluid goes, but we can try to divide the whole domain into smaller pieces (elements) and to see how the fluid inside them behaves (velocity, pressure, flux in/out etc.).



3. **[Initial condition]** Keep in mind that for solving differential equations, often it is necessary to specify an initial condition. In some cases, we might have to specify an initial condition in the whole domain even if we don't know it exactly. We will have to *guess* to facilitate the solution (for instance the Formula Racing car, we might initialize the whole domain with a uniform velocity).

4. **[Boundary condition]** This is one of the *key issues* in CFD (in general, in numerical solution of partial differential equations). The computer has *finite memory*, thus we need to specify what happens on the end of our domain. And we need to do it carefully to respect the physics, since the solution is often strongly affected by the boundaries. In this specific case, the information is transported by velocity c from left to right (positive sign), thus the left boundary (i.e. the first node) will not be affected by what is happening in the domain. In addition to that, it would be not coherent to take information from outside the domain

(we are using backward difference approximation for space, in the formula we would need $u[-1]$).

5. **[Mesh size and resolution]** Try to change the number of elements in the mesh from 41 to 81. What happens? As a digital camera, we can increase or decrease the *resolution* of our discrete solution, having a direct impact on the *cost* of the computation (required cpu time and memory).

6. **[Stability (CFL)]** Now try to increase the number of elements from 81 to 82 (and then above). What happens? Not only the cost of the computation is affected when we modify the resolution of our calculation. But also *stability* issues might occur. The Courant–Friedrichs–Lewy (CFL) condition prescribes a limitation in certain types of computations that has to be respected:

   $$CFL = \frac{c\,\Delta t}{\Delta x} \leq CFL_{\max}$$

   Intuitively, the speed c of our solution must not be greater than the space/time resolution of our scheme, given by dx and dt. Following the photographic camera example, it is like we must capture the movement of a very fast car: we need a high-fidelity camera that is able to perform slow motion reproduction (high number of frames per second).

   Note, by *stability* we mean basically that our simulation does not amplify numerical errors during the time evolution (i.e. the simulation does not "blow up").

2.2 Step 2: Non-linear Convection

Now we're going to implement non-linear convection using the same methods as in step 1. The 1D convection equation is:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = 0 \tag{2.4}$$

[Non-linearty] Instead of a constant factor c multiplying the second term, now we have the solution u multiplying it. Thus, the second term of the equation is now *non-linear*.
We're going to use the same discretization as in Step 1 — forward difference in time and backward difference in space. Here is the discretized equation.

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

Solving for the only unknown term, u_i^{n+1}, yields:

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

As before, the Python code starts by loading the necessary libraries. Then, we declare some variables that determine the discretization in space and time (you should experiment by changing these parameters to see what happens). Then, we create the initial condition u_0 by initializing the array for the solution using $u = 2$ @ $0.5 \leq x \leq 1$ and $u = 1$ everywhere else in $(0, 2)$ (i.e., a hat function).

```
In [ ]: import numpy                 #we're importing numpy and calling it np locally
        from matplotlib import pyplot    #and our 2D plotting library, calling it plt
        %matplotlib inline


        nx = 41
        dx = 2 / (nx - 1)
        nt = 20    #nt is the number of timesteps we want to calculate
        dt = .025  #dt is the amount of time each timestep covers (delta t)

        u = numpy.ones(nx)      #as before, we initialize u with every value equal to 1.
        u[int(.5 / dx) : int(1 / dx + 1)] = 2  #then set u = 2 between 0.5 and 1 as per our I.C.

        pyplot.plot(numpy.linspace(0, 2, nx), u) ##Plot the results

        un = numpy.ones(nx) #initialize our placeholder array un, to hold the time-stepped solut
```

The code snippet below is *unfinished*. We have copied over the line from Step 1 that executes the time-stepping update. Can you edit this code to execute the non-linear convection instead?

```
In [ ]: for n in range(nt):  #iterate through time
            un = u.copy() ##copy the existing values of u into un
            for i in range(1, nx):  ##now we'll iterate through the u array

             ###This is the line from Step 1, copied exactly.  Edit it for our new equation.
             ###then uncomment it and run the cell to evaluate Step 2

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

        pyplot.plot(numpy.linspace(0, 2, nx), u) ##Plot the results
```

[Stability and CFL]

2.2.1 Remarks for this exercise

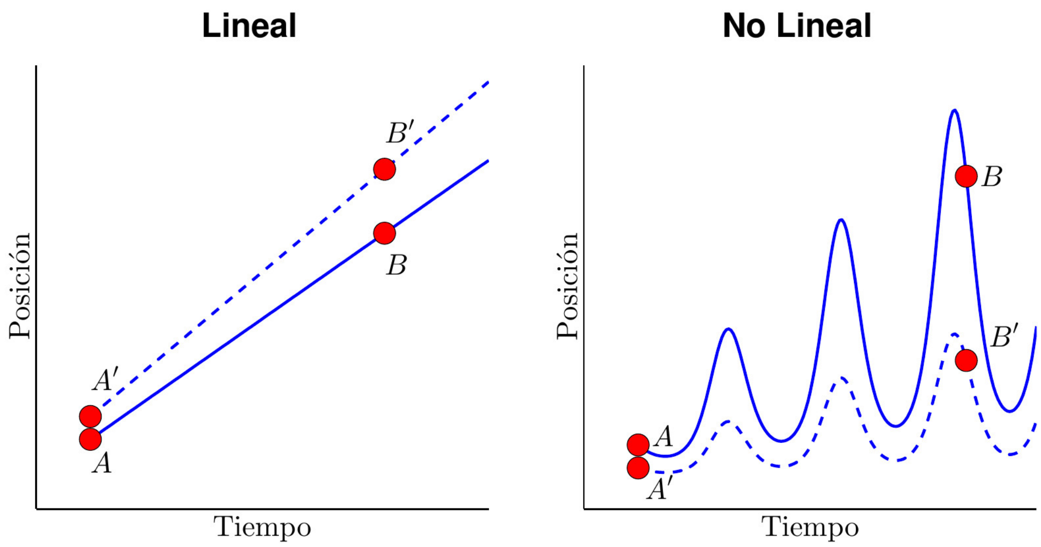
In this simple example we can already comment about:

1. **[Non-linearity]** We just repeat the definition of *linear* and *non-linear* function: Mathematically, a linear function $f(x)$ is one which satisfies both of the following properties:

    - Additivity or superposition principle: $f(x+y) = f(x) + f(y)$;
    - Homogeneity:$f(\alpha x) = \alpha f(x)$.

    As an exercise, just check whether the governing equation (2.4) satisfies these properties.

    Intuitively, for a *linear* function the output is "directly" (linearly) proportional to the input (think of the equation of a line, the exponent is one x^1). While for a *non-linear* function the output is not that directly related to the input (think of a parabola, we have a dependence to the square x^2).



The fact is that the *non-linearity* is a key issue in the solution of the equations of fluid motion, and is the source of the main difficulties in their solution.

2. **[Stability and CFL]** What do you observe about the evolution of the hat function under the non-linear convection equation? What happens when you change the numerical parameters and run again? Try to change the parameters as before (number of elements, total time, size of time step). What happens? Would you be able to calculate the CFL limit in this case?

2.3 Step 3: Diffusion Equation in 1-D

The one-dimensional diffusion equation is: **[Partial differential equation]**

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

The first thing you should notice is that —unlike the previous two simple equations we have studied— this equation has a second-order derivative. We first need to learn what to do with it!

Discretizing $\frac{\partial^2 u}{\partial x^2}$

[Different discretizations for different terms]] The second-order derivative can be represented geometrically as the line tangent to the curve given by the first derivative. We will discretize the second-order derivative with a Central Difference scheme: a combination of Forward Difference and Backward Difference of the first derivative. Consider the Taylor expansion of u_{i+1} and u_{i-1} around u_i:

$$u_{i+1} = u_i + \Delta x \frac{\partial u}{\partial x}\bigg|_i + \frac{\Delta x^2}{2}\frac{\partial^2 u}{\partial x^2}\bigg|_i + \frac{\Delta x^3}{3!}\frac{\partial^3 u}{\partial x^3}\bigg|_i + O(\Delta x^4)$$

$$u_{i-1} = u_i - \Delta x \frac{\partial u}{\partial x}\bigg|_i + \frac{\Delta x^2}{2}\frac{\partial^2 u}{\partial x^2}\bigg|_i - \frac{\Delta x^3}{3!}\frac{\partial^3 u}{\partial x^3}\bigg|_i + O(\Delta x^4)$$

If we add these two expansions, you can see that the odd-numbered derivative terms will cancel each other out. If we neglect any terms of $O(\Delta x^4)$ or higher (and really, those are very small), then we can rearrange the sum of these two expansions to solve for our second-derivative.

$u_{i+1} + u_{i-1} = 2u_i + \Delta x^2 \frac{\partial^2 u}{\partial x^2}\bigg|_i + O(\Delta x^4)$

Then rearrange to solve for $\frac{\partial^2 u}{\partial x^2}\bigg|_i$ and the result is:

$$\frac{\partial^2 u}{\partial x^2} = \frac{u_{i+1} - 2u_i + u_{i-1}}{\Delta x^2} + O(\Delta x^2)$$

Back to Step 3

We can now write the discretized version of the diffusion equation in 1D:

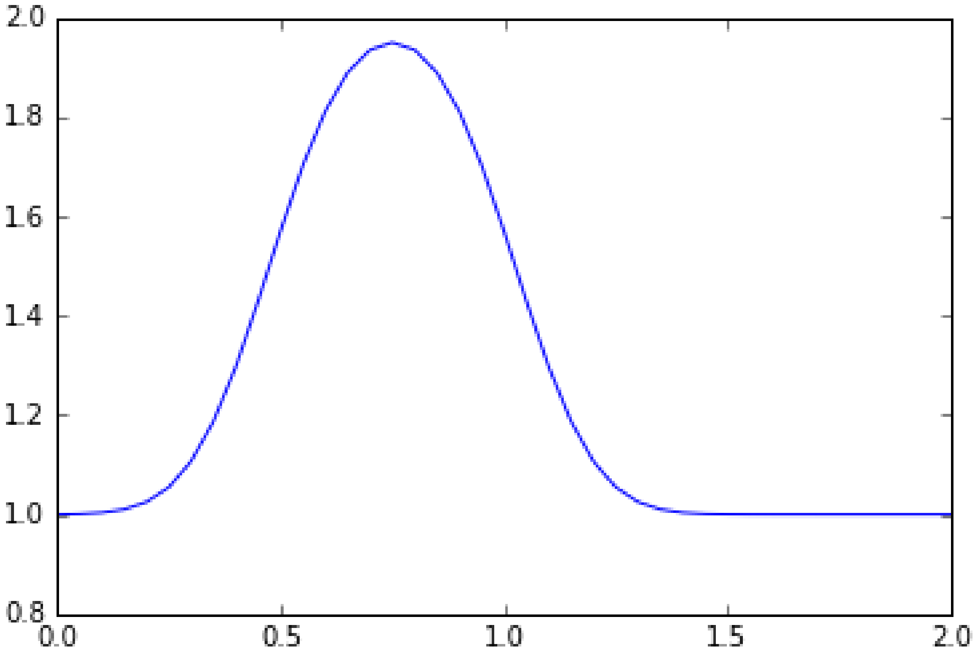
$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = \nu \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2}$$

As before, we notice that once we have an initial condition, the only unknown is u_i^{n+1}, so we re-arrange the equation solving for our unknown:

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

The above discrete equation allows us to write a program to advance a solution in time. But we need an initial condition. Let's continue using our favorite: the hat function. So, at $t = 0$, $u = 2$ in the interval $0.5 \leq x \leq 1$ and $u = 1$ everywhere else. We are ready to number-crunch!

```
In [1]: import numpy                     #loading our favorite library
        from matplotlib import pyplot    #and the useful plotting library
        %matplotlib inline

        nx = 41
        dx = 2 / (nx - 1)
        nt = 20    #the number of timesteps we want to calculate
        nu = 0.3   #the value of viscosity
        sigma = .2 #sigma is a parameter, we'll learn more about it later
        dt = sigma * dx**2 / nu #dt is defined using sigma ... more later!


        u = numpy.ones(nx)      #a numpy array with nx elements all equal to 1.
        u[int(.5 / dx):int(1 / dx + 1)] = 2  #setting u = 2 between 0.5 and 1 as per our I.C.s

        un = numpy.ones(nx) #our placeholder array, un, to advance the solution in time

        for n in range(nt):  #iterate through time
            un = u.copy() ##copy the existing values of u into un
            for i in range(1, nx - 1):
                u[i] = un[i] + nu * dt / dx**2 * (un[i+1] - 2 * un[i] + un[i-1])

        pyplot.plot(numpy.linspace(0, 2, nx), u);
```



[Physical meaning of viscosity]

2.3.1 Remarks for this exercise

In this simple example we can already comment about:

1. **[Partial differential equation]** Both the *advection*, *diffusion*, and *unsteady* terms are present in the Navier-Stokes equations. Each term can be discretized in a different way, and the different options are present in the "Solver" window of your CFD solver (e.g. Star CCM+). So it is important to understand that you can *combine them*, choose for some that are more *stable* but less accurate, or accurate but more unstable (for the advection term for instance). Or for the time discretization, you can go for more costly and slow but stable (implicit method, using bigger time steps), or less costly and fast but unstable (explicit methods, respecting the CFL condition, using smaller time steps).

2. **[Physical meaning of viscosity]** Try to change the value of viscosity ν. What happens to the solution? Note that the higher viscosity ν, the more "smeared" the solution is. In the case of the racing car, the fluid is *air*, and the viscosity is "low" compared to the (high) velocity. This is true away from the surface of the car. However, closer we get to the surface (at surface the velocity is zero), we have that the velocity gets lower, thus the viscosity starts to be important. In this sense, we need to "capture the boundary layer", and we need to produce a finer mesh on the surface.

   If you want, you can notice that the viscous CFL is defined as

$$\sigma_{visc} = \frac{\nu \Delta t}{(\Delta x)^2} < \frac{1}{2}$$

Chapter 3

2D problems

3.1 Step 5: 2-D Linear Convection

Up to now, all of our work has been in one spatial dimension (Steps 1 to 4). We can learn a lot in just 1D, but let's grow up to flatland: two dimensions.

In the following exercises, you will extend the first four steps to 2D. To extend the 1D finite-difference formulas to partial derivatives in 2D or 3D, just apply the definition: a partial derivative with respect to x is the variation in the x direction *at constant* y.

In 2D space, a rectangular (uniform) grid is defined by the points with coordinates:

$$x_i = x_0 + i\Delta x$$

$$y_i = y_0 + i\Delta y$$

Now, define $u_{i,j} = u(x_i, y_j)$ and apply the finite-difference formulas on either variable x, y *acting separately* on the i and j indices. All derivatives are based on the 2D Taylor expansion of a mesh point value around $u_{i,j}$.

Hence, for a first-order partial derivative in the x-direction, a finite-difference formula is:

$$\left.\frac{\partial u}{\partial x}\right|_{i,j} = \frac{u_{i+1,j} - u_{i,j}}{\Delta x} + \mathcal{O}(\Delta x)$$

and similarly in the y direction. Thus, we can write backward-difference, forward-difference or central-difference formulas for Steps 5 to 12. Let's get started!

The PDE governing 2-D Linear Convection is written as **[Two-dimensional: vector equation]**

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

This is the exact same form as with 1-D Linear Convection, except that we now have two spatial dimensions to account for as we step forward in time.

Again, the timestep will be discretized as a forward difference and both spatial steps will be discretized as backward differences.

With 1-D implementations, we used i subscripts to denote movement in space (e.g. $u_i^n - u_{i-1}^n$). Now that we have two dimensions to account for, we need to add a second subscript, j, to account for all the information in the regime.

Here, we'll again use i as the index for our x values, and we'll add the j subscript to track our y values.

With that in mind, our discretization of the PDE should be relatively straightforward.

$$\frac{u_{i,j}^{n+1} - u_{i,j}^{n}}{\Delta t} + c\frac{u_{i,j}^{n} - u_{i-1,j}^{n}}{\Delta x} + c\frac{u_{i,j}^{n} - u_{i,j-1}^{n}}{\Delta y} = 0$$

As before, solve for the only unknown:

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

We will solve this equation with the following initial conditions:

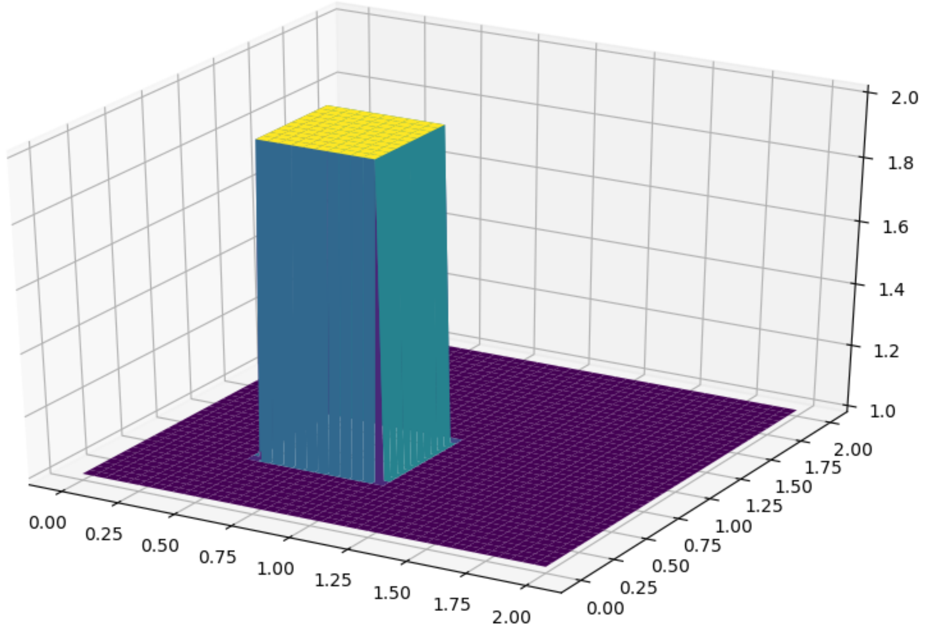
$$u(x,y) = \begin{cases} 2 \text{ for} & 0.5 \le x,y \le 1 \\ 1 \text{ for} & \text{everywhere else} \end{cases}$$

and boundary conditions:

$$u = 1 \text{ for } \begin{cases} x = 0,\ 2 \\ y = 0,\ 2 \end{cases}$$

```
In [12]: from mpl_toolkits.mplot3d import Axes3D    ##New Library required for projected 3d plot

         import numpy
         from matplotlib import pyplot, cm
         %matplotlib inline

         ###variable declarations
         nx = 81
         ny = 81
         nt = 100
         c = 1
         dx = 2 / (nx - 1)
         dy = 2 / (ny - 1)
         sigma = .2
         dt = sigma * dx

         x = numpy.linspace(0, 2, nx)
         y = numpy.linspace(0, 2, ny)

         u = numpy.ones((ny, nx)) ##create a 1xn vector of 1's
         un = numpy.ones((ny, nx)) ##

         ###Assign initial conditions

         ##set hat function I.C. : u(.5<=x<=1 && .5<=y<=1 ) is 2
         u[int(.5 / dy):int(1 / dy + 1),int(.5 / dx):int(1 / dx + 1)] = 2

         ###Plot Initial Condition
```

```
##the figsize parameter can be used to produce different sized images
fig = pyplot.figure(figsize=(11, 7), dpi=100)
ax = fig.gca(projection='3d')
X, Y = numpy.meshgrid(x, y)
surf = ax.plot_surface(X, Y, u[:], cmap=cm.viridis)
```



3D Plotting Notes

To plot a projected 3D result, make sure that you have added the Axes3D library.

```
from mpl_toolkits.mplot3d import Axes3D
```

The actual plotting commands are a little more involved than with simple 2d plots.

```
fig = pyplot.figure(figsize=(11, 7), dpi=100)
ax = fig.gca(projection='3d')
surf2 = ax.plot_surface(X, Y, u[:])
```

The first line here is initializing a figure window. The **figsize** and **dpi** commands are optional and simply specify the size and resolution of the figure being produced. You may omit them, but you will still require the

```
fig = pyplot.figure()
```

The next line assigns the plot window the axes label 'ax' and also specifies that it will be a 3d projection plot. The final line uses the command

```
plot_surface()
```

which is equivalent to the regular plot command, but it takes a grid of X and Y values for the data point positions.

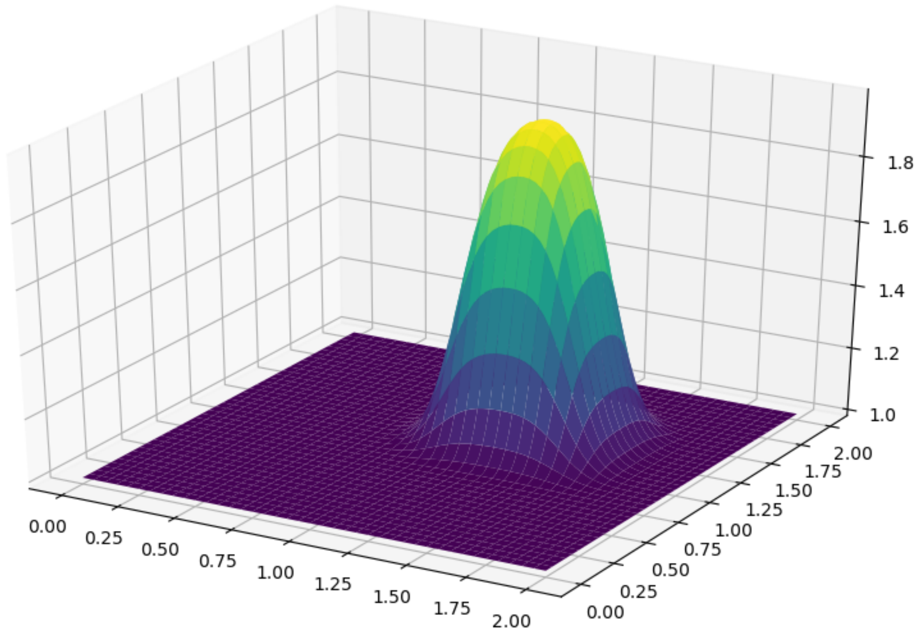
Note The `X` and `Y` values that you pass to `plot_surface` are not the 1-D vectors `x` and `y`. In order to use matplotlibs 3D plotting functions, you need to generate a grid of `x, y` values which correspond to each coordinate in the plotting frame. This coordinate grid is generated using the numpy function `meshgrid`.

```
X, Y = numpy.meshgrid(x, y)
```

Iterating in two dimensions

To evaluate the wave in two dimensions requires the use of several nested for-loops to cover all of the `i`'s and `j`'s. Since Python is not a compiled language there can be noticeable slowdowns in the execution of code with multiple for-loops. First try evaluating the 2D convection code and see what results it produces. **[Boundary conditions in 2D]**

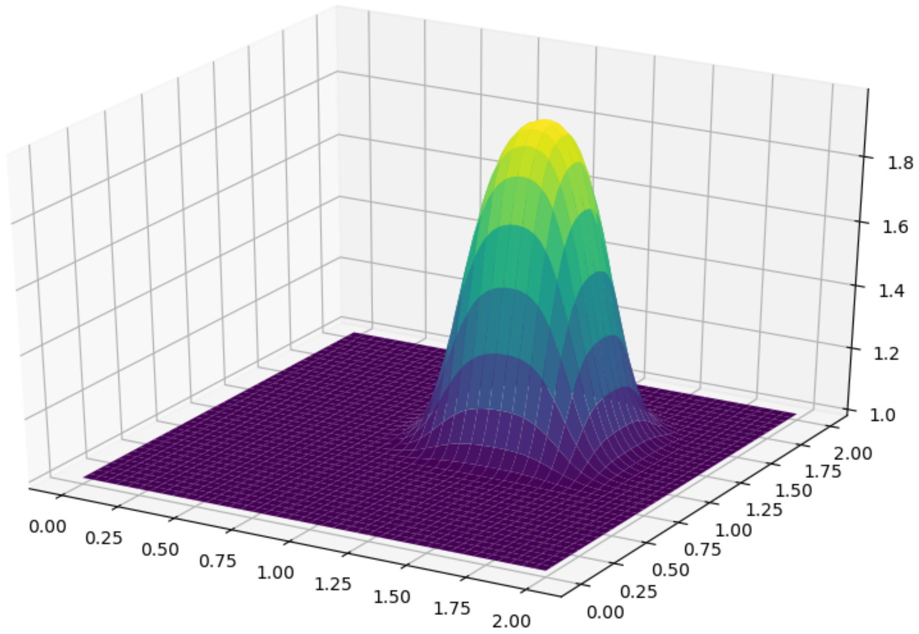
```
In [13]: u = numpy.ones((ny, nx))
         u[int(.5 / dy):int(1 / dy + 1), int(.5 / dx):int(1 / dx + 1)] = 2

         for n in range(nt + 1): ##loop across number of time steps
             un = u.copy()
             row, col = u.shape   # Gets the dimensions of the array "u"
             for j in range(1, row):
                 for i in range(1, col):
                     u[j, i] = (un[j, i] - (c * dt / dx * (un[j, i] - un[j, i - 1])) -
                                           (c * dt / dy * (un[j, i] - un[j - 1, i])))

                     # Boundary conditions are implemented at Each Time step:
                     u[0, :] = 1
                     u[-1, :] = 1
                     u[:, 0] = 1
                     u[:, -1] = 1

         fig = pyplot.figure(figsize=(11, 7), dpi=100)
         ax = fig.gca(projection='3d')
         surf2 = ax.plot_surface(X, Y, u[:], cmap=cm.viridis)
```



3.1.1 Array Operations

Here the same 2D convection code is implemented, but instead of using nested for-loops, the same calculations are evaluated using array operations.

```
In [14]: u = numpy.ones((ny, nx))
         u[int(.5 / dy):int(1 / dy + 1), int(.5 / dx):int(1 / dx + 1)] = 2

         for n in range(nt + 1): ##loop across number of time steps
             un = u.copy()
             u[1:, 1:] = (un[1:, 1:] - (c * dt / dx * (un[1:, 1:] - un[1:, :-1])) -
                                       (c * dt / dy * (un[1:, 1:] - un[:-1, 1:])))

             # Boundary conditions are implemented at Each Time step:
             u[0, :] = 1
             u[-1, :] = 1
             u[:, 0] = 1
             u[:, -1] = 1

         fig = pyplot.figure(figsize=(11, 7), dpi=100)
         ax = fig.gca(projection='3d')
         surf2 = ax.plot_surface(X, Y, u[:], cmap=cm.viridis)
```



3.1.2 Remarks for this exercise

In this simple example we can already comment about:

1. **[Two-dimensional: vector equation]** This is a *vector* equation. Can you say how the velocity c is oriented in this case? The NV equations are *vector* equations. This is important for many aspects of the solution strategy, but also for the *post-process* of the results: streamlines, pathlines, turbulence etc.

2. **[Boundary conditions in 2D]** Try to change the values of the boundary condition from 1 to 2 for instance. What do you notice? The boundary condition is transported into the domain! This gives an idea of how the *boundary conditions affects heavily* the solution inside the domain, and it is very important to set them in a suitable way (think about inflow or outflow: we need some knowledge of the practical case we are solving).