PDE Diaries

A. Nieto

September 2020

1 The rules of the game

"We do not know what the rules of the game are; all we are allowed to do is to watch the playing. Of course, if we watch long enough we may eventually catch on to a few rules. The rules of the game are what we mean by fundamental physics" - Richard Feynman

The universe, as I want to believe it, moves according to some given rules, it follows a hidden pattern and it goes somewhere unknown. The partial differential equations (PDE) offer us a way to glimpse into those rules, when only as a coarse approximation. Any simulation we made reflects the universe in its conception, it moves on its own, following only the rules given by us at the beginning, but independently from our will and in some cases even beyond our prediction capabilities.

The name "PDE Diares" is quite misleading. I came up with this name on one of my holiday trips, the idea was to have a small notebook where I was suppose to solve some PDE exercises in a daily basis. As you can imagine after a short period of time the notebook became just a dead weight in my luggage, nevertheless the name stuck.

This text is actually a summary of some ideas I had while trying to learn how to code numerical simulations with Python. I'll start from the end, because as any other story, its development depends almost uniquely on the retrospective ideas of the author, on what it was already thought. My hope is that you will enjoy learning about *the rules of the game* as much as I did!

2 Disclaimer

The text, as it is today, is unfortunately not prepared as an introduction to numerical simulations, I'm assuming for the moment that the readers have some notion of how a numerical simulation goes, from the discretization of the PDEs to the post-processing visualization of the results. I'll eventually provide some external links I find adequate for beginners and some time in the future I'll try to include some basic stuff as well ... some day.

3 Nomenclature

I was thinking for a long time what would be the best way to shorten the already short nomenclature of the PDEs and discretization schemes. The risk I see by this approach is that eventually no one will understand what the extra-shortened nomenclature was supposed to mean, included me. We can give it a try anyway, for the sake of beauty!

3.1 PDE nomenclature

Let's start with some basic nomenclature, having the scalar field of an arbitrary quantity in function of time and 2D space $u(t, x, y)$, we express the change of u w.r.t. one of its variables as:

$$\frac{\partial u}{\partial t} \rightarrow u_t$$
$$\frac{\partial u}{\partial x} \rightarrow u_x$$
$$\frac{\partial u}{\partial y} \rightarrow u_y$$

The second derivative w.r.t. the same variable is then written as:

$$\frac{\partial^2 u}{\partial x^2} \rightarrow u_{xx}$$

And finally the mixture of several partial derivatives w.r.t. different variables can be expressed as:

$$\frac{\partial^2 u}{\partial x \partial y} \rightarrow u_{xy}$$

That way we can keep our PDEs nice and clean, but what about the discretization schemes?

3.2 Discretization schemes - "classic" nomenclature

This one is going to be trickier as the discretized schemes relate on many upper- and sub-indexes to be understandable. The first step is to define the discretization scheme that is normally used for partial derivatives. Take into account that these schemes are derived from the Taylor expansion by neglecting the H.O.T. (Higher Order Terms), which will lead irrevocably to a cumulative error and thus it counts only as an approximation of the derivative terms.

For a 1D scalar field of the quantity $u(x, t)$:

$$u_t \approx \frac{u_i^{n+1} - u_i^n}{\Delta t} \tag{1a}$$

$$u_x \approx \frac{u_i^n - u_{i-1}^n}{\Delta x} \tag{1b}$$

Where $n+1$ express a leap forward in time and $i-1$ a leap backward in x-space.

When dealing with 2D scalar fields of the type $u(t,x,y)$ it becomes necessary to express the leaps in y-space by using an extra sub-index, so that the previous expressions become:

$$u_t \approx \frac{u_{i,j}^{n+1} - u_{i,j}^n}{\Delta t} \tag{2a}$$

$$u_x \approx \frac{u_{i,j}^{n} - u_{i-1,j}^n}{\Delta x} \tag{2b}$$

$$u_y \approx \frac{u_{i,j}^{n} - u_{i,j-1}^n}{\Delta y} \tag{2c}$$

And the derivative of $u(t,x,y)$ w.r.t. y is expressed by using backward leaps in y-space as given in (2c). For the second order derivatives w.r.t. the same variable we have:

$$u_{xx} \approx \frac{u_{i+1,j}^{n} - 2u_{i,j}^n + u_{i-1,j}^n}{\Delta x^2} \tag{3a}$$

$$u_{yy} \approx \frac{u_{i,j+1}^{n} - 2u_{i,j}^n + u_{i,j-1}^n}{\Delta y^2} \tag{3b}$$

I think you can start to see what a mess of indexes this nomenclature will produce when discretizing more complex PDEs. But all of these indexes are necessary for a good comprehension of the discretization schemes!

3.3 Discretization schemes - "short" nomenclature

What I propose next is a compromise between given information and readability, with the aim of making simpler and faster schemes, less prone to typos and more easy to correct. We start by writing (2) as:

$$u_t \approx \frac{u_{oo}^{+} - u_{oo}}{\Delta t} \tag{4a}$$

$$u_x \approx \frac{u_{oo} - u_{-o}}{\Delta x} \tag{4b}$$

$$u_y \approx \frac{u_{oo} - u_{o-}}{\Delta y} \tag{4c}$$

And then we write (3) as:

$$u_{xx} \approx \frac{u_{+o} - 2u_{oo} + u_{-o}}{\Delta x^2} \tag{5a}$$

$$u_{yy} \approx \frac{u_{o+} - 2u_{oo} + u_{o-}}{\Delta y^2} \tag{5b}$$

So what had happened here? In summary:

- All the letters n were dropped, as we normally only used one instance of the quantity in the next time step, that instance is represented now by the upper-index $+$, as shown in (4a).

- All the letters i, j and commas were dropped, as we already know we are dealing with a 2D space in function of x and y, and the former comes always first in the sub-indexes order.
- The backward and forward leaps in space are represented with the sub-indexes $-$ and $+$ respectively. The order here is important, as $u_{+o} \neq u_{o+}$, the former expressing a leap forward in x-space $(i+1)$ and the latter a leap forward in y-space $(j+1)$.
- When there are no leaps in space, the sub-index o is used as analog to $i+0$ or $j+0$.

That way we have cleared most of the writing while still holding a compressible nomenclature. We'll see in future examples how this "short" nomenclature will help us to quickly write and program our schemes.

4 Arrays 1D

As we are going to be working intensively with array operators, it is important to understand what is actually happening. Having a good understanding of what an arrays is and its graphical representation is vital for this goal. In this section we are going to deal only with 1D arrays, the same ideas will be expanded in the next section for the 2D domain.

4.1 Graphical representation 1D

When imagining the graphical representation of a 1D array, a common way to do it would be setting a line of values increasing from left to right. The position of such values play a very important role as it determines not only which discretization scheme will be apply to them but also its final values.

Let's start by defining a 1x6 zeros array. The reason for the size of this array is merely arbitrary, I think is a good compromise of a large enough array, that will allow later to show some basic operation, and is small enough to avoid being cumbersome. We create the array (out of thin air!) with the numpy command:

```
>>> u = np.zeros(6)
```

Such command will yield the following representation:

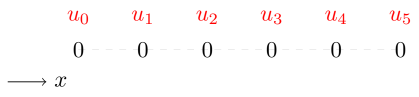


Figure 1: 1x6 zeros array.

Where the subindex of u indicates the position of the given value in the array using the argument nomenclature used in Python (where zero is the first element of the array). For example, if we want to get the value of the first element of our brand array, we use the command:

```
>>> u[0]
>>> 0.0
```

It is interestingg to note that if we start counting on zero, because that's the way python does, our last element is always the size of our array minus one.

4.2 Inner and Boundary values 1D

For all the discretization schemes I have known so far, we always talk about the "inner" and the "boundary" values in a separate way. The former represent what is happening inside our 1D domain and the latter takes care of the interactions with the external world. A nice way of visualizing both values in our 1x6 array would be as presented below:

4.3 Python loop operators 1D

One common way of approaching the discretization schemes of the PDEs is to use a series of `for` loops. Let's say we have the most basic PDE given for the linear 1D convection:

$$u_t + cu_x = 0 \tag{6}$$

Where c is the constant velocity at which the quantity $u(t, x)$ is moving. We'll apply the 1D discretization schemes presented in (1) to obtain the discretized version of equation (6):

$$\frac{u_o^+ - u_o}{\Delta t} + c\frac{u_o - u_-}{\Delta x} = 0 \tag{7}$$

as you may have notice, we have already implemented the "short" nomenclature for the discretization schemes discussed in subsection 3.3, only for a 1D case. Solving for u_o^+ results in:

$$u_o^+ = u_o - c\frac{\Delta t}{\Delta x}(u_o - u_-) \tag{8}$$

and use the following python code to solve (8) for all points in a 1x6 array:

```python
for i in range(len(x)):
    u[i] = un[i] - c*dt/dx*(un[i]-un[i-1])
```

Using a `for` loop is an easy way of dealing with the discretization scheme presented by (8), but not the most efficient one. As we increase the number of elements of our arrays, the time needed to to the loop will increase exponentially. We'll see in the next section a way of bypass, to certain extend, this problematic by using the so-called "array operators".

4.4 Python array operators 1D

A way of improving our simulation performance is to use the array operators instead of the `for` loop discused in the previous section.

5 Arrays 2D

5.1 2D Graphical representation of the arrays

Let's start by defining a 6x6 zeros array with the numpy command:

```
u = np.zeros((6, 6))
```

This 6x6 array (or matrix) is going to be our workhorse for the rest of the section, here we are going to visualize the different discretization schemes. By plotting our brand new u array in an imaginary grid we obtain:

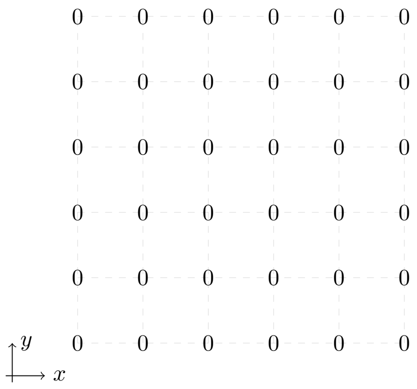


Figure 2: 6x6 zeros array.

Only as a test, let's say we want to represent the identity matrix for this 6x6 array, that is, the central diagonal elements will have a value of 1 and the rest will stay in 0.

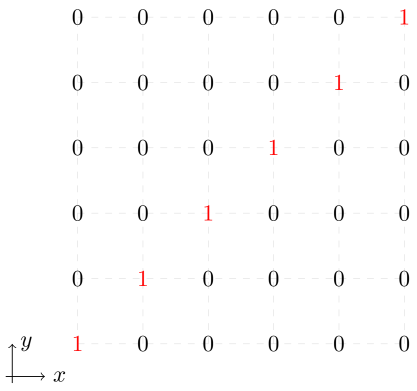
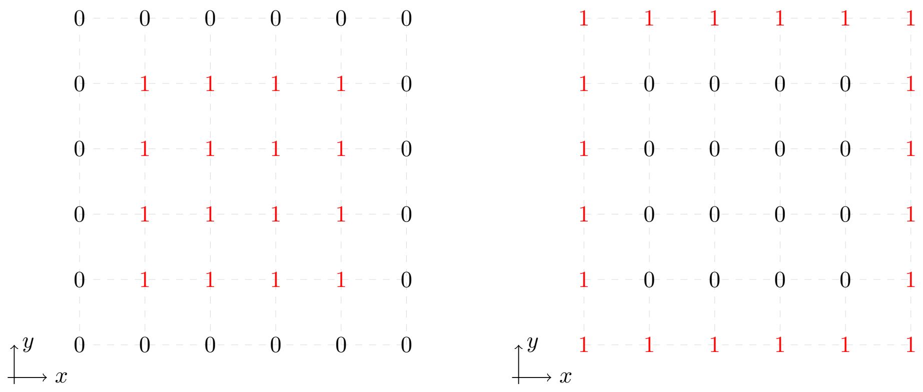


Figure 3: Identity matrix.

Contrary to the numerical representation of the identity matrix, where the diagonal of ones would go downwards from left to right, the values on the graphical representation have its origin (0,0) on the south-left corner and increase in the x-axis to the right and in the y-axis upward.

5.2 Inner and Boundary values

So far good right? Then let's go to the interesting stuff, when discretizing PDEs we always need to treat the main (or inner) values separate from the boundary values.



(a) Inner values. (b) Boundary values.

Figure 4: Different types of values depending on its location in the array.

Both inner and boundary values will have different discretization schemes depending on the requirement given by the problem at hand, but we will come to that point later.