Guide to Python3

Jiajing Guan

December 23, 2020

1 Requirement

The code attached is run on Python 3.8. There are some differences between Python 2.7 and Python 3.x. Please verify the version of Python installed by running the following command before starting:

```
python --version
```

Most matrix operations rely on Numpy and Scipy package of Python. The most common way of installing packages of Python is by running `pip`. One can check if `pip` has been installed by running:

```
python -m pip --version
```

If `pip` is not installed, here is a useful link for installing `pip`: Link.

Now that `pip` has been installed, run the following command to install Numpy and Scipy:

```
pip install numpy
pip install scipy
```

We will need to use Python to plot a figure later. So while we are at it, run the following command to install matplotlib:

```
pip install matplotlib
```

Once Numpy is installed, we can import the package in the preamble section by:

```
import numpy as np
import matplotlib.pyplot as plt
```

We have several functions that need to be imported. Also add these lines in the preamble:

```
from scipy import sparse
from scipy.sparse.linalg import spsolve
```

Later when we call functions defined in Numpy, we will just use `np.(function_name)`.

2 Sample code that performs FDM

To demostrate the basic functionality of Python3, we will use Finite Difference Method to solve the following ODE problem:

$$\begin{aligned} -\xi u^{''} + u^{'} &= 0 \quad \text{for } x \in (0,1) \\ u(0) &= 1 - e^{-1/\xi} \\ u(1) &= 0 \end{aligned} \tag{1}$$

Here, the parameter is $\xi = 10^a$, where a is chosen from a uniform distribution of $[-4, 0]$.

My usually habit in using Python3 is to create different problems as separate classes. Inside the class, I will define different functions that fulfill different purposes. When I wish to call the functions, I will create a class instance and call functions on the instance. We will go into details about how to call functions later.

First, let's consider what input FDM would need. We need to define the stepsize h and parameter ξ. To make the class as flexible as possible, we want to treat ξ as an input to functions, not class, so that we could run functions to compute approximations with different ξ without instantiating the class every time. Now let's set up the class:

```
class CD_1D:
    def __init__(self, h):
        self.h = h
        self.N = int(1/self.h)+1
        self.nx = self.N-2
        self.X = np.linspace(0,1,num=self.N)
```

In Python, if one wants to declare a class, one usually needs a function named `__init__`. This is where the inputs to the class will go and this function is automatically called when the class is instantiated. In the code above, we can see that the input h is saved to `self.h`. Any variable prefixed by `self.` can be accessed within the class. Now any functions inside this class can access h by calling `self.h`. We also create some essential variables that are useful, such as `self.N` (the total number of discretized points in $[0,1]$), `self.nx` (the number of interior points) and `self.X` (the discretized points in $[0,1]$).

We know that the FDM solve can be obtained by the following equation:

$$AU = F$$

$$\begin{bmatrix} \frac{2\xi}{h^2} & \frac{1}{2h}-\frac{\xi}{h^2} & & & \\ -\frac{1}{2h}-\frac{\xi}{h^2} & \frac{2\xi}{h^2} & \frac{1}{2h}-\frac{\xi}{h^2} & & \\ & \ddots & \ddots & \ddots & \\ & & -\frac{1}{2h}-\frac{\xi}{h^2} & \frac{2\xi}{h^2} & \frac{1}{2h}-\frac{\xi}{h^2} \\ & & & -\frac{1}{2h}-\frac{\xi}{h^2} & \frac{2\xi}{h^2} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ \vdots \\ u_{N-1} \end{bmatrix} = \begin{bmatrix} (\frac{1}{2h}+\frac{\xi}{h^2})(1-e^{-1/\xi}) \\ 0 \\ \vdots \\ \vdots \\ 0 \end{bmatrix}$$

$$\frac{\xi}{h^2}\begin{bmatrix} 2 & -(1-\frac{h}{2\xi}) & & & \\ -(1+\frac{h}{2\xi}) & 2 & -(1-\frac{h}{2\xi}) & & \\ & \ddots & \ddots & \ddots & \\ & & -(1+\frac{h}{2\xi}) & 2 & -(1-\frac{h}{2\xi}) \\ & & & -(1+\frac{h}{2\xi}) & 2 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ \vdots \\ u_{N-1} \end{bmatrix} = \frac{\xi}{h^2}\begin{bmatrix} (1+\frac{h}{2\xi})(1-e^{-1/\xi}) \\ 0 \\ \vdots \\ \vdots \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} 2 & -(1-\frac{h}{2\xi}) & & & \\ -(1+\frac{h}{2\xi}) & 2 & -(1-\frac{h}{2\xi}) & & \\ & \ddots & \ddots & \ddots & \\ & & -(1+\frac{h}{2\xi}) & 2 & -(1-\frac{h}{2\xi}) \\ & & & -(1+\frac{h}{2\xi}) & 2 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ \vdots \\ u_{N-1} \end{bmatrix} = \begin{bmatrix} (1+\frac{h}{2\xi})(1-e^{-1/\xi}) \\ 0 \\ \vdots \\ \vdots \\ 0 \end{bmatrix}$$

How do we create the sparse matrix A and vector F? In Numpy and Scipy, there are a lot of functions very similar to the ones in Matlab. Like `spdiags` in Matlab, there is a function in Scipy that perform similarly, which can be called by `scipy.sparse.diags`. We can create A by the following command:

```
r = self.h/(2*xi)
A = sparse.diags([-(1+r),2,-(1-r)],[-1,0,1],\
        shape=(self.nx,self.nx), format = 'csr')
```

Vector F can be created by the following command:

```
F = np.zeros((self.nx,1))
F[0] = (1+r)*(1-np.exp(-1/xi))
```

We can solve a sparse system by the function `spsolve` imported earlier. Putting everything together, we can write the function that solves this system with an input ξ as:

```
def generate_one_sol(self, xi):
        r = self.h/(2*xi)
        A = sparse.diags([-(1+r),2,-(1-r)],[-1,0,1],\
                shape=(self.nx,self.nx), format = 'csr')
        F = np.zeros((self.nx,1))
        F[0] = (1+r)*(1-np.exp(-1/xi))
        u = spsolve(A,F)
        U = np.zeros((self.N,1))
        U[1:-1,0] = u
        U[0] = 1-np.exp(-1/xi)
        return U
```

The last three lines before the return statement fills in the boundary conditions. Besides this FDM solver, we will add a function that produce the exact solution:

```
def u_exact(self,xi):
        u = 1-np.exp((self.X-1)/xi)
        return u
```

Now that we have a class with a function that would use FDM to solve Equation 1, how do we call it? What I usually do is to create a DEMO file that would call this class. But this time, to keep all code in one file, I will put what I usually write in another file under the line `if __name__ == "__main__":`. What this line does is that when Python is processing the script, it will treat the commands inside this if block as the main function and run it.

In this main block, I want to run the `generate_one_sol` function on $h = 1/2048$ and $\xi = 0.1$. Then I will obtain the exact solution by running `u_exact`. Lastly, I want to plot these two variables on one plot with the approximation plotted in red circles. To run these functions in class `CD_1D`, we need to first instantiate the class by running:

```
env = CD_1D(h)
```

Then we will call functions by:

```
U = env.generate_one_sol(xi)
U_ex = env.u_exact(xi)
```

Now we will plot these quantities. A useful link is provided here: Link. For the most part, this is very similar to Matlab plotting. The one thing we need to be careful about is that the figure can only be shown we call `plt.show()` at the end of the plotting block. The following commands will produce the desired figure:

```
plt.plot(env.X,U,'ro')
plt.plot(env.X,U_ex,'b')
plt.ylabel('u')
plt.xlabel('x')
plt.show()
```

Putting everything together, the main function block is shown below:

```
if __name__ == "__main__":
        h = 1/2048
        xi = 0.1
        env = CD_1D(h)
        U = env.generate_one_sol(xi)
```

```
U_ex = env.u_exact(xi)
plt.plot(env.X,U,'ro')
plt.plot(env.X,U_ex,'b')
plt.ylabel('u')
plt.xlabel('x')
plt.show()
```

Let's save this file as `CD_1d.py`. To run this script, run the following command in the command line that is rooted in the same folder as the script:

```
python3 CD_1d.py
```

3 Main differences between Matlab and Python

The first difference, and probably the biggest difference, is that Python is sensitive to indentation. The indentation level determines the block structure.

The second difference is on how array can be sliced. Besides starting index from 0, Python slicing cuts off before the ending index. For example, the line `U[1:-1,0]` will access the second to the second to last indexed entries in the first column of `U`, not the second to the last entry in `U`. I found myself always confused by this difference so it is a good idea to run commands on simple examples to verify that the slicing does exactly what you want it to do before putting them in your code.

The last point, not difference, I want to make is that I found the Numpy package to be very similar to Matlab commands. Most times, if I want to find the Python substitute for a Matlab function, I just google "numpy Matlab function name". Nine times out of ten, I could find a function for it.