Lab4_Notebook

September 26, 2020

1 Lab 4: Arrays and Libraries for Numerical Programming

In this session we will learn about arrays (commonly used in numercaal programming) and several very useful libraries for numerical programming and scientific computing which are very popular among engineers, physicists, biologists, economists, etc. These libraries (which often use arrays or lists) can be imported into our code using the familiar python command `import`.

We are going to be using the libraries: NumPy, SciPy and MatPlotLib - these are all part of the Anaconda installation, and should work if you are using Jupyter (or Spyder) within Anaconda.

The first one we will use, and is the most commonly used for scientific programming, is NumPy

www.numpy.org

NumPy is an excellent library for numerical programming which provides code for common numerical operations, such as

Vector and Matrix operations (creating and manipulating arrays of numbers)

Linear algebra

Fourier transforms

Random number generation

To find basic information about NumPy, check the website.

We will start with Numpy arrays.

1.1 Numpy Arrays

An array is very similar to a list, which we introduced in the last session. An array is a list of numbers, on which you can perform mathematical operations. Handling arrays is similar to handling lists, but here are the main differences:

Arrays need to be set up using numpy, and you need to use numpy commands to perform operations on them

Arrays can only contain one type of variable (integers or floating points)

You cannot store text in arrays

You will need to use arrays, not lists, if you want to perform maths operations on entire sequences of numbers

1.1.1 Setting up arrays

We start by demonstrating an example of how to create an array of numbers using NumPy's functions `linspace()` and `arange()`:

`arange` creates an array full of integers and has the basic structure:

`arange(start, stop, step)`

`linspace` creates an array full of floats and has the basic structure:

`linspace(start, stop, number-of-elements)`

First we have to import Numpy. Using the commond below imports Numpy and all the Numpy functions will have the form `np.XXX( )`

[1]:
```
import numpy as np
```

[2]:
```
a = np.arange(1,10)
print(a)
```

```
[1 2 3 4 5 6 7 8 9]
```

we can use the same sort of commands as we did for lists for working with array elements...

[3]:
```
print(a[0])
print(a[2])
print(a[-1])
print(len(a))
```

```
1
3
9
9
```

[4]:
```
a = np.arange(1,10,2)
print(a)
```

```
[1 3 5 7 9]
```

Notice that `arange` does not includes the end point of the defined range.

Now let's try `linspace`

[5]:
```
a = np.linspace(0, 10, 21)
print(a)
```

```
[ 0.   0.5  1.   1.5  2.   2.5  3.   3.5  4.   4.5  5.   5.5  6.   6.5
  7.   7.5  8.   8.5  9.   9.5 10. ]
```

Notice that `linspace` includes the end points of the defined range.

You'll see that we needed to have 21 array elements to have values spaced by 0.5 between 0 and 10 (21 elements gves 20 gaps of 0.5)

Sometimes we need to set up array with a fixed number of elements but where we don't yet know what values we want to store. We can set up an array and fill is with zeros…using `zeros`

[6]:
```
x = np.zeros(5)
print(x)
```

```
[0. 0. 0. 0. 0.]
```

1.1.2 Working with Arrays

NumPy arrays can be used in mathematical operations. For example, arrays can be multiplied by/divided by/added a constant:

[7]:
```
x=np.linspace(0,5,11)
print(x)

a=5
print(a*x)
```

```
[0.  0.5 1.  1.5 2.  2.5 3.  3.5 4.  4.5 5. ]
[ 0.   2.5  5.   7.5 10.  12.5 15.  17.5 20.  22.5 25. ]
```

…and we can assign this to a new array y. In the case below, Python knows that y needs to be array, since x is an array…

[8]:
```
y=a*x
print(y)

y=a+x
print(y)
```

```
[ 0.   2.5  5.   7.5 10.  12.5 15.  17.5 20.  22.5 25. ]
[ 5.   5.5  6.   6.5  7.   7.5  8.   8.5  9.   9.5 10. ]
```

We can also perform mathematical or trig functions on arrays:

[9]:
```
y = np.cos(x)
print(y)
```

```
[ 1.          0.87758256  0.54030231  0.0707372  -0.41614684 -0.80114362
 -0.9899925  -0.93645669 -0.65364362 -0.2107958   0.28366219]
```

Notice we have used `np.cos( )`. This is a special numpy function that allows you to operate on arrays. If you tried `y=cos(x)` it would not work if x is an array. Every maths and trig function you are likely to use has an `np.xxx( )` version (e.g. `np.cos( ), np.sin( ), np.sqrt( ), np.log( )`

Two arrays of the same length can be multiplied/divided/added:

[10]:
```
print(x)
print(y)
```

```
print(x*y)
```

```
[0.  0.5 1.  1.5 2.  2.5 3.  3.5 4.  4.5 5. ]
[ 1.          0.87758256  0.54030231  0.0707372  -0.41614684 -0.80114362
 -0.9899925  -0.93645669 -0.65364362 -0.2107958   0.28366219]
[ 0.          0.43879128  0.54030231  0.1061058  -0.83229367 -2.00285904
 -2.96997749 -3.27759841 -2.61457448 -0.9485811   1.41831093]
```

Looping through arrays is a common technique. We will find ourselves using the `len( )` function often when doing this kind of thing…

We can sum up the elements of an array using the "brute-force" method as:

[11]:
```
sum_it = 0
for i in range(0, len(x), 1):
    sum_it = sum_it + x[i]

print (sum_it)
```

```
27.5
```

A quicker way to sum up an array in NumPy is using function `np.sum( )`

(and there are lots of other functions for acting on arrays - look them up!)

[12]:
```
print(np.sum(x))
```

```
27.5
```

1.1.3 Arrays as vectors

You can use arrays to store vectors, and perform vector manipulation. For example, lets set up two arrays (x and y) each with three elements and treat them as vectors with three components…

[13]:
```
x = np.array([1, 5, 2])
y = np.array([7, 4 ,1])
print(x)
print(y)
```

```
[1 5 2]
[7 4 1]
```

[14]:
```
print(x+y)
print(np.dot(x,y))
print(np.cross(x,y))
```

```
[8 9 3]
29
[ -3  13 -31]
```

Notice how the dot product gives us a scaler answer (of course) and the cross product gives a vector, which we could assign as a new array if we wished.

This kind of manipulation is very helpful when doing approximate numerical methods on vector quantities...

1.2 The SciPy library

SciPy contains many routines that can perform the type of numerical method that we carried out in lab 3. Many of them operate on arrays.

For example, let us look at the `scipy integrate` package which is a package of functions that performed numerical integration.

[15]:
```
from scipy import integrate
```

[ ]:
```
help(integrate)
```

The help function lists the content of the library integrate within SciPy, including all available functions related to numerical integration. Say we are interested in integrating our set of data using the method `trapz` from this library. We really need to take a numerical methods class to understand what this function does, so don't worry about how this works! We only need to remember that this function calculates an integral given a set of data.

We could also show the information for `trapz` by performing the `help()` function again.

[ ]:
```
help(integrate.trapz)
```

You can see from the help tool that to perform the `trapz` function both `y` and `x` must be arrays.

Say we want to integrate the function `cos(x)` over an interval from 0 to 2pi using the `trapz` method :

[16]:
```
import numpy as np
import math
from scipy import integrate
x = np.linspace(0, 2*math.pi, 100)      # create 100 divisions of the interval
y = np.cos(x)                           # Note: You have toy use np.cos( ) NOT␣
 ↪cos( ) since it is an array
integral_value = integrate.trapz(y,x)   # Calls scipy's integrate function␣
 ↪called trapz, using the x and y arrays
print (integral_value)
```

```
-2.914335439641036e-16
```

This is really just one very simple example of SciPy. Many more examples can be found in SciPy's cookbook.

1.3 Matplotlib

Matplotlib is a Python 2D plotting library useful for creating quality figures, plots, charts and simulations within Python scripts and shells. The matplotlib.org tutorials provide excellent guidance on how to produce different types of graphs and visualisations.

In this lab we will start by learning how to plot simple functions and then extend this to produce a simulation.

1.3.1 Plotting in matplotlib

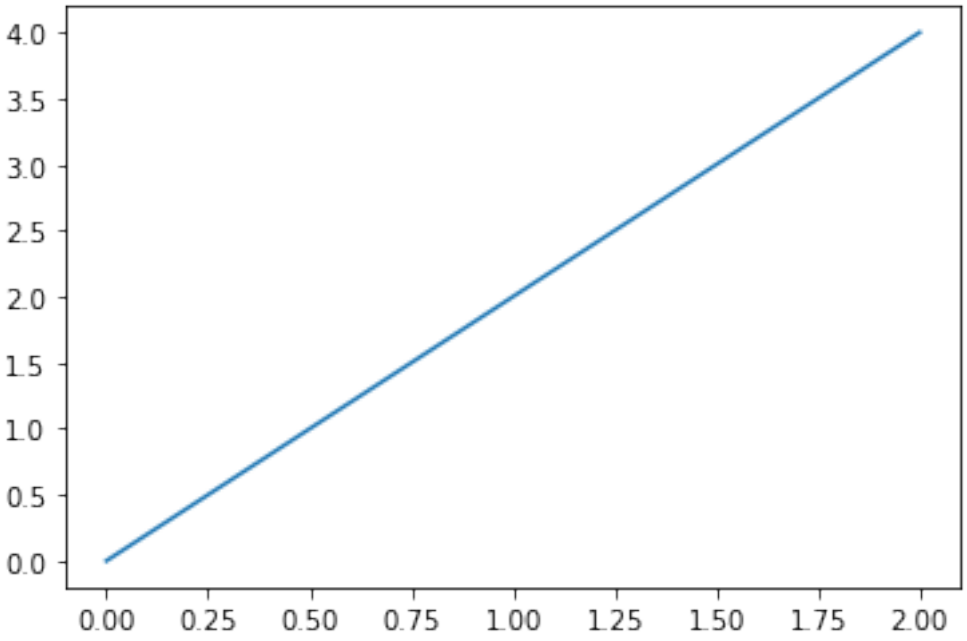
`matplotlib.pyplot` is a collection of functions that make matplotlib work like MATLAB. Each pyplot function makes some change to a figure: e.g., creates a figure, creates a plotting area in a figure, plots some lines in a plotting area, decorates the plot with labels, etc.

After importing this library, a first call to `plt.plot` will automatically create the necessary figure and axes to achieve the desired plot. Subsequent calls to `plt.plot` re-use the current axes and each add another line. Setting the title, legend and axis labels also automatically use the current axes.

The main input parameters for pyplot are the x and y coordinates. Usually these are lists or arrays. Other input parameters such as the line type, line colour, marker type and marker size can all be specified within `plt.plot()`.

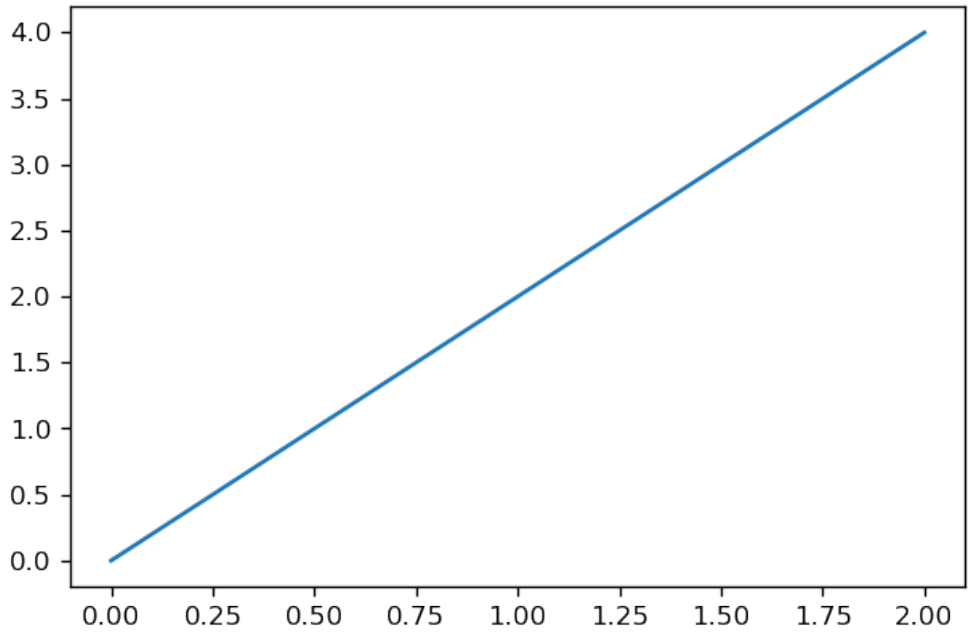
A basic plot for the equation $y = 2x$ is shown below. The function `show()` is necessary for the image to show up on the screen, and must be included at the end of the script.

[17]:
```
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(0, 2, 10)
plt.plot(x, 2*x,)
plt.show()
```



We can make the graph a bit bigger in the Jupyter window, by adding a command that sets the resolution of the figure…

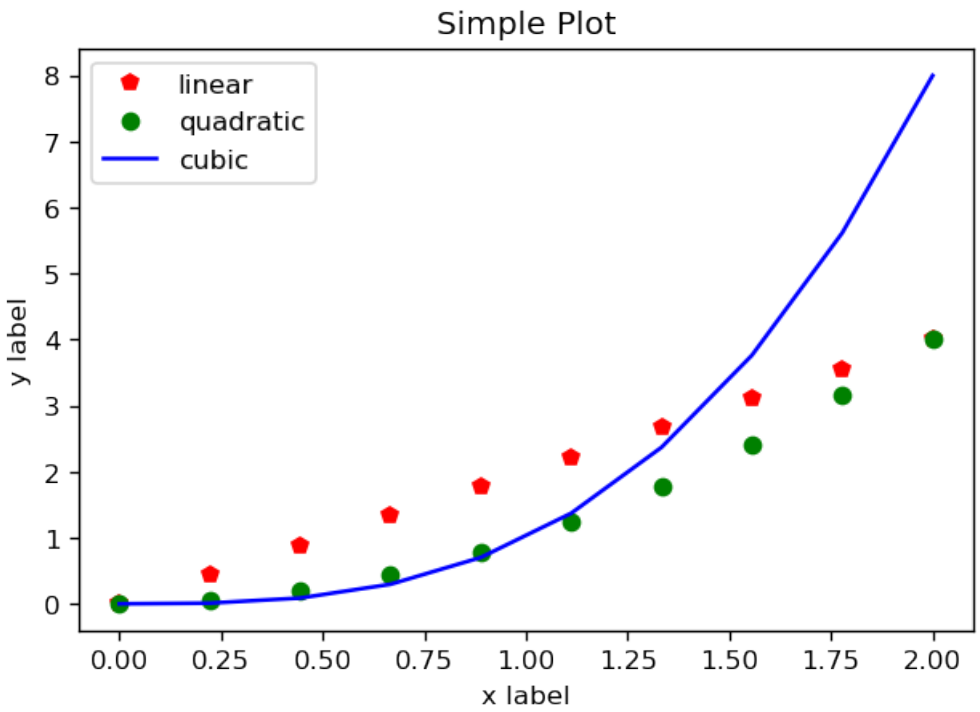
[18]:
```
import matplotlib.pyplot as plt
plt.rcParams['figure.dpi'] = 120       # Sets the resolution (in dpi) of the␣
 ↪figure
import numpy as np

x = np.linspace(0, 2, 10)
plt.plot(x, 2*x,)
plt.show()
```



You can see from the output of this code that no axis labels are given, and the axis scale has been automatically set. The example below shows how to add multiple lines to the same plot and add axis labels, line labels and a plot title.

The colour of each line has been changed and inputs given to specify the marker type and whether or not a line should be drawn between points. 'p' means draw hexagons, 'o' means draw circles. If nothing is specified then a line is drawn.

[19]:
```
x = np.linspace(0, 2, 10)
plt.plot(x, 2*x, 'p', label='linear', color = 'red')
plt.plot(x, x**2, 'o', label='quadratic', color = 'green')
plt.plot(x, x**3, label='cubic', color ='blue')
plt.xlabel('x label')
plt.ylabel('y label')
plt.title('Simple Plot')
plt.legend()
plt.show()
```



You can visit the plyplot.plot help page here on the MatPlotLib web site to see all the available options - there are loads !!:

https://matplotlib.org/3.3.2/api/_as_gen/matplotlib.pyplot.plot.html

Though to be honest, the best way to learn is seeing how other people do it (look online, or in other examples you come across)

[ ]: