Introduction to numpy, scipy and matplotlib

Modeling and Simulation: CC-05 unit II

Document in a Glance:

1.1 Install numpy, scipy and matplotlib

Before working with numpy, scipy and matplotlib, we need to install them as follows. Open a cmd window and use the next set of commands to install NumPy, SciPy and Matplotlib: Assuming that you have already installed Python.

```
python -m pip install numpy
python -m pip install scipy
python -m pip install matplotlib
```

After running each of the above commands you should see the last line saying *Successfully installed*. Then Launch Python from a cmd window and check the version of Scipy, you should see something like this:

```
C:\>python
Python 3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019, 22:39:24) [MSC v.1916 32 bit (Intel)]
on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import scipy as sp
>>> sp.version.version
'1.4.1'
```

1.1.1: NumPy

NumPy (Numeric Python) is probably the most fundamental package in Python designed to support a powerful multi-dimensional array object as well as high-level mathematical and numerical functions that can be utilized for efficient scientific computing. It is also clear use for scientific computing; it can also be utilized as an efficient multi-dimensional container of generic data.

1.1.2: SciPy

SciPy (Scientific Python) is a set of open source scientific and numerical tools built on the Numpy extension of Python. It adds significant power to the interactive Python session by providing the user with high-level commands and classes for manipulating and visualizing data. Scipy builds on Numpy, and for all basic array handling needs you can use Numpy functions when using SciPy functions.

1.1.3: Matplotlib

Matplotlib is probably the single most used Python package for 2D-graphics. It provides both a very quick way to visualize data from Python and publication-quality figures in many formats.

1.2: Importing the packages

There are several ways to import NumPy and Matplotlib, but the community has created the modules strongly recommends following these import conventions. They have adopted, as shown below.

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

It is also recommended to import SciPy sub-packages individually; similar to what is shown below.

```
from scipy import linalg, optimizepy
```

These conventions are used throughout official NumPy and SciPy source code and documentation, as well as other examples and documentations. Although it is not required to follow these conventions, again, it is still strongly recommended. We will also be using these conventions as for the remainder of this tutorial.

The SciPy and Matplotlib utilize NumPy arrays; therefore it is appropriate to discuss them first. The array object class is the central feature of NumPy. Arrays are similar to lists in Python, except that every element of an array must be of the same type, typically a numeric type like float or int. Arrays make operations with large amounts of numeric data very fast and are generally much more efficient than lists.

1.3: NumPy and methods

NumPy's array class is called ndarray, also known by the alias array. There are many methods of creating arrays. An array can be created directly from a list of values:

```
>>> np.array([[2, 3, 4], [1, 2, 3]])
 array([2, 3, 4], [1, 2, 3])
>>> cvalues = [22.2, 131.7, 6.4, 7.]
>>> np.array(cvalues)
array([22.2, 131.7, 6.4, 7.])
 >>> np.array([2, 3, 4], dtype=float)
array([2., 3., 4.])
```

You can also generate an array of values from a given half-open interval using numpy.arange:

```
>>> np.arange(3)
 array([0, 1, 2])
>>> np.arange(3.0)
 array([ 0., 1., 2.])
>>> np.arange(3,7)
array([3, 4, 5, 6])
>>> np.arange(3,7,2)
```

```
array([3, 5])
```

An array of evenly spaced values over a closed interval can be generated using *numpy.linspaces*

```
 >>> np.linspace(2.0, 3.0, num=5)
 array([ 2. , 2.25, 2.5 , 2.75, 3. ])
>>> np.linspace(2.0, 3.0, num=5, endpoint=False)
array([ 2. , 2.2, 2.4, 2.6, 2.8])
>>> np.linspace(2.0, 3.0, num=5, retstep=True)
 (array([ 2. , 2.25, 2.5 , 2.75, 3. ]), 0.25)
```

Special arrays can also be generated using NumPy, like an array of zeroes, one, and even one with a diagonal filled with one while the rest are zeroes.

```
 >>> np.zeros(2,2)
array([[ 0., 0.], [ 0., 0.]])
>>> np.ones(5)
array([ 1., 1., 1., 1., 1.])
>>> np.full((2, 2), 10)
array([[10, 10], [10, 10]])
>>> np.eye(3, dtype=int)
array([[1, 0, 0], [0, 1, 0], [0, 0, 1]])
```

1.3.1: Array Indexing

There are many options to indexing using NumPy, which gives numpy indexing great power, but with power comes some complexity and the potential for confusion. Single numpy arrays can be indexed similar to indexing Python arrays. For multidimensional arrays, you also have the ability to get a single row or the element of multiple rows that are of the same column. For example:

```
 >>> Z = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
>>> Z[1, :]
array([5, 6, 7, 8])
 >>> Z [:, 1:2]
array([[ 2], [ 6], [10]])
```

You can also get multiple values from an array using Integer Array Indexing. Using the previous array Z:

```
>>> Z[[0, 1, 2], [0, 1, 0]] # (0,0), (1,1) and (2,0) element
array([1, 6, 9])
```

To further showcase the indexing power of NumPy, here is another example using the same array:

```
>>> Z[Z > 2]
array([ 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
```

1.3.2: Array Attribute

Array attributes reflect information that is intrinsic to the array itself. Generally, accessing an array through its attributes allows you to get and sometimes set intrinsic properties of the array without creating a new array. The exposed attributes are the core parts of an array and only some of them can be reset meaningfully without creating a new array:

ndarray.flags	Information about the memory layout of the array
ndarray.shape	Tuple of array dimensions
ndarray.ndim	Number of array dimensions
ndarray.size	Number of elements in the array
ndarray.itemsize	Length of one array element in bytes
ndarray.nbytes	Total bytes consumed by the elements of the array
ndarray.base	Base object if memory is from some other object
ndarray.dtype	Data-type of the array's elements
ndarray.T	Same as self.transpose(), except that self is returned if self.ndim < 2
ndarray.flat	A 1-D iterator over the array

1.3.3: Basic Array Methods

NumPy has several methods for and handling and manipulating. Given a multidimensional array a, you can generate a copy of that array as a Python list:

```
>>> a = np.array([[1, 2], [3, 4]])
>>> a.tolist()
[[1, 2], [3, 4]]
```

You can also get an element of after it is converted to a standard Python scalar:

```
 >>> a
array([[1, 2], [3, 4]])
>>> a.item(3)
4
>>> a.item((1,0))
 3
```

You can also insert an element into the array using numpy.itemset:

```
 >>> a.itemset(3, 9)
>>> a
array([[1, 2], [3, 9]])
>>> a.itemset((1,0), 21)
>>> a
array([[ 1, 2], [21, 9]])
```

Replacing multiple elements is also possible using numpy.put:

```
 >>> a = np.arange(5)
 >>> a
array([0, 1, 2, 3, 4])
>>> np.put(a, [0,2],[-22, 57])
>>> a
 array([-22, 1, 57, 3, 4])
```

Or you can also replace every element in the array with a single element:

```
 >>> a.fill(22)
>>> a
array([22, 22, 22, 22, 22])
```

You can also join a sequence of arrays along an existing axis:

```
 >>> a = np.array([1,2], float)
>>> b = np.array([3,4,5,6], float)
>>> c = np.array([7,8,9], float)
>>> np.concatenate((a, b, c))
array([1., 2., 3., 4., 5., 6., 7., 8., 9.])
```

1.3.4: Array Shape Manipulation

The shape of an array can also be manipulated and changed with various commands. The first one would be numpy.reshape, which gives the array a new shape without modifying its data:

```
 >>> a = np.arange(6)
 array([0, 1, 2, 3, 4, 5. 6])
>>> a.reshape((3, 2))
array([[0, 1], [2, 3], [4, 5]])
```

The previous method has a restriction since the new shape is limited to the total number of elements in the array. Another method that does not have the same restriction as the previous method is numpy.resize. If the new array resulting from the specified shape is larger than the original array, then the new array is filled with repeated copies of the original array:

```
 >>> a=np.array([[0,1],[2,3]])
 >>> np.resize(a,(2,3))
array([[0, 1, 2], [3, 0, 1]])
```

Another method lets you interchange the two axes of an array:

```
 >>> a = np.array([[1, 2], [3, 4]])
>>> a
array([[1, 2], [3, 4]])
>>> a.swapaxes(1,0)
 array([[1, 3], [2, 4]])
```

1.3.5: Mathematical Function

NumPy also provides a vast library for mathematical routines, ranging from basic Algebraic and Arithmetic functions, to Trigonometric and Hyperbolic functions, and even handling of complex numbers.

```
 >> a = np.sin(np.pi/3)
>>> a
 0.8660254037844386
>>> b = np.cos(np.sqrt(9))
>>> b
 -0.98999249660044542
 >>> c = np.multiply(a, b)
>>> c
 -0.85735865161196523
 >>> np.reciprocal(c)
-1.166373020345508
```

1.3.6: Polynomials

NumPy supplies methods for working with polynomials. Given a set of roots, it is possible to show the polynomial coefficients:

```
 >>> np.poly([-1, 1, 1, 10])
array([ 1, -11, 9, 11, -10])
```

In the example, the array output corresponds to coefficients of the equation $x^4 - 11x^3 + 9x^2 + 11x - 10$. The opposite can also be done to get the roots. The roots function can receive an array of coefficients as an input and returns an array of roots:

```
 >>> np.roots([1, 4, -2, 3])
array([-4.57974010+0.j , 0.28987005+0.75566815j, 0.28987005-0.75566815j])
```

NumPy also has the ability to return the derivative and antiderivative (indefinite integral) of a polynomial. Given an array of coefficients of a polynomial, when can get the derivative using numpy.polyder:

```
 >>> p = np.poly1d([1,1,1,1])
>>> p2 = np.polyder(p)
>>> p2
poly1d([3, 2, 1])
```

and the anti-derivative using numpy.polyint:

```
 >>> p = np.poly1d([3,2,1])
 >>> p2 = np.polyint(p)
>>> p2
poly1d([1., 1., 1., 0.])
```

Lastly, you can also evaluate a polynomial at specific values:

```
 >>> np.polyval([3,0,1], 5)
76
```

In the previous example, the polynomial is 3x2 + 1 evaluated at x = 5, which looks like $3(5)^2 + 0(5) + 1$, which evaluates to 76.

1.4: SciPy Basics

SciPy extends the functionality of the NumPy Routines. SciPy is organized into sub-packages according to different scientific computing domains, but we are not going to cover each and every one of them in detail, but rather discuss and provide examples of some of its capabilities.

1.4.1: Integration

SciPy provides several integration techniques under scipy.integrate, including an ordinary differential equation integrator.

```
 >>> import scipy.integrate as integrate
 >>> import scipy.special as special
 >>> result = integrate.quad(lambda x: special.jv(2.5,x), 0, 4.5)
 >>> result
(1.1178179380783253, 7.866317182537226e-09)
```

1.4.2: Interpolation

There are several general interpolation facilities available in SciPy, for data in one, two, and higher dimensions. For example, when evaluating a one dimensional vector of data, you can usescipy.interpolate.interp1d:

```
 >>> x = np.linspace(0, 10, num=11, endpoint=True)
>>>x array([ 0., 1., 2., 3., 4., 5., 6., 7., 8., 9., 10.])
>>> y = np.cos(-x**2/9.0)
>>> y
array([  1.  ,  0.99383351,  0.90284967,  0.54030231,  -0.20550672,  -0.93454613,  -
0.65364362, 0.6683999 , 0.67640492, -0.91113026, 0.11527995])
>>> f1 = interp1d(x, y)
>>> f2 = interp1d(x, y, kind='cubic')
```

The result f1 and f2 are class instances, and each one can be treated like a function, which interpolates between known data values to obtain unknown values. Given an interval plugged in to the instance, the result can be seen by displaying it in a graph.

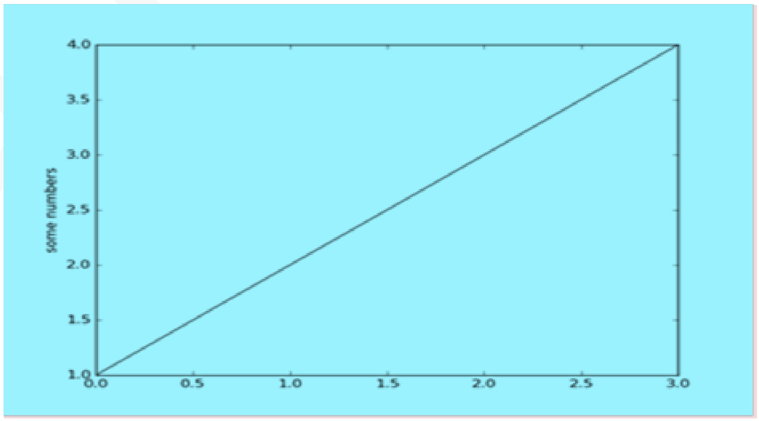
SciPy doesn't have any functions that handle plotting. Instead, we will use Matplotlib, which will be discussed in the next section.

1.5: Matplotlib basics

We have already covered NumPy and SciPy, and in terms of providing plotting functions, neither provides any kind of support. There are several plotting packages available for Python, the most commonly used one being Matplotlib. And pyplot is a collection of command style 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. Before working with matplotlib, it is highly recommended to install IPython first. IPython is an *enhanced interactive Python* shell that has lots of interesting features including named inputs and outputs, access to shell commands, improved debugging and many more. It is central to the scientific-computing workflow in Python for its use in combination with Matplotlib.

A basic example of plotting using Matplotlib is shown below, wherein matplotlib.pyplot.ylable is utilize to create a label for the y-axis:

```
 >>> import matplotlib.pyplot as plt
 >>> plt.plot([1,2,3,4])
>>> plt.ylabel('some numbers')
>>> plt.show()
```

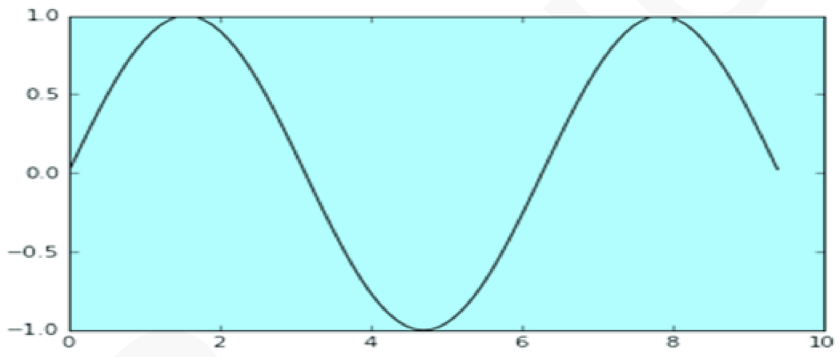


1.5.1: Plotting

The most important function in matplotlib is plot, which allows you to plot 2D data. Here is a simple example:

```
import numpy as np
import matplotlib.pyplot as plt
# Compute the x and y coordinates for points on a sine curve
x = np.arange(0, 3 * np.pi, 0.1)
y = np.sin(x)
# Plot the points using matplotlib
plt.plot(x, y)
plt.show()  # You must call plt.show() to make graphics appear.
```

Running this code produces the following plot:

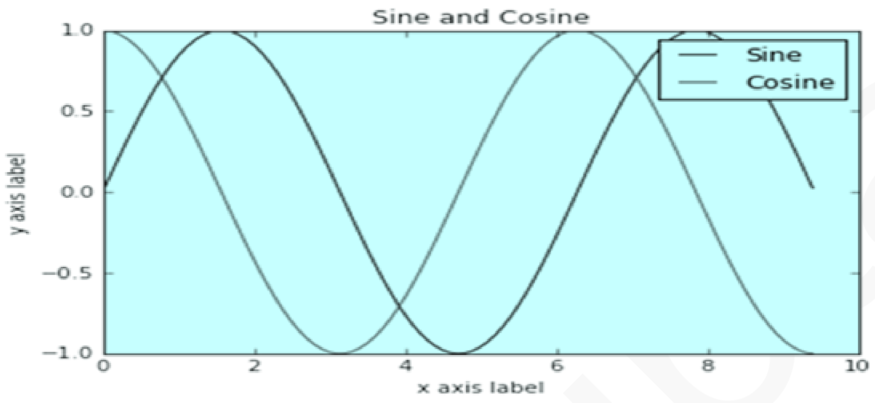


With just a little bit of extra work we can easily plot multiple lines at once, and add a title, legend, and axis labels:

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

# Compute the x and y coordinates for points on sine and cosine curves
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(x)
y_cos = np.cos(x)

# Plot the points using matplotlib
plt.plot(x, y_sin)
plt.plot(x, y_cos)
plt.xlabel('x axis label')
plt.ylabel('y axis label')
plt.title('Sine and Cosine')
plt.legend(['Sine', 'Cosine'])
plt.show()
```



You can read much more about the plot function in the documentation related to plot.

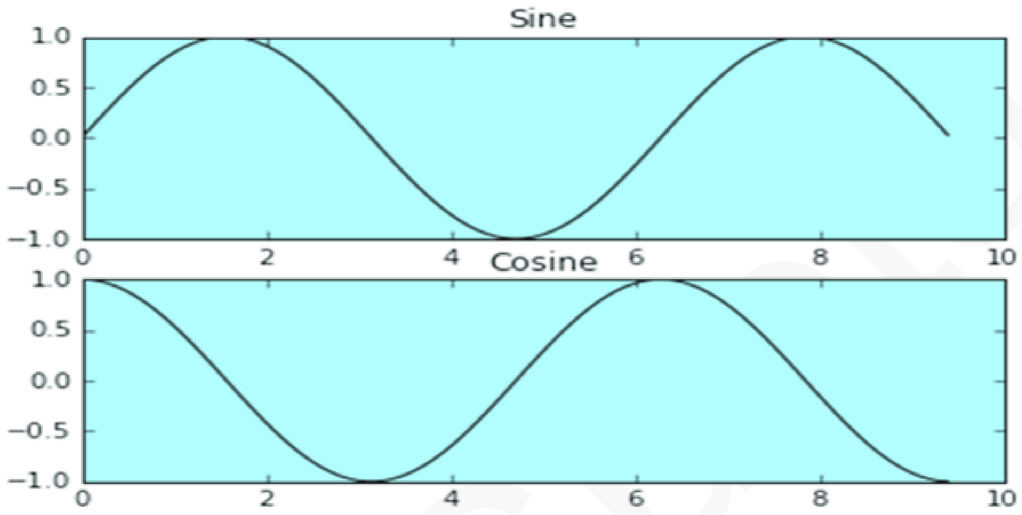
1.5.2: Subplots

You can plot different things in the same figure using the subplot function. Here is an example:

```
import numpy as np
import matplotlib.pyplot as plt
# Compute the x and y coordinates for points on sine and cosine curves
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(x)
y_cos = np.cos(x)

# Set up a subplot grid that has height 2 and width 1,
# and set the first such subplot as active.
plt.subplot(2, 1, 1)
# Make the first plot
plt.plot(x, y_sin)
plt.title('Sine')
# Set the second subplot as active, and make the second plot.
plt.subplot(2, 1, 2)
plt.plot(x, y_cos)
plt.title('Cosine')

# Show the figure.
plt.show()
```



You can read much more about the subplot function in the documentation related to subplot.