CHAPTER

4 NumPy and Matplotlib

OUTLINE

Harry, I have no idea where this will lead us, but I have a definite feeling it will be a place both wonderful and strange.

***–"Dale Cooper" in the television series* Twin Peaks**

CHAPTER POINTS

- NumPy is a library that provides a flexible means to define and manipulate vectors, matrices, and higher-dimensional arrays.
- Matplotlib enables the visualization of numerical results with only a few lines of code.

In this chapter we will cover two important libraries that are available for Python: NumPy and Matplotlib. In this lecture we will make particular reference to nuclear and radiological engineering applications. It is not essential to understand these applications, but these will help motivate our discussion.

Computational Nuclear Engineering and Radiological Science Using Python
DOI: 10.1016/B978-0-12-812253-2.00005-4

4.1 NUMPY ARRAYS

NumPy is a collection of modules, called a library, that gives the programmer powerful array objects and linear algebra tools, among other things. In this section we will explore that arrays that NumPy supplies.

The basic unit in NumPy is a multi-dimensional array, sometimes called an N-dimensional or $N - D$ array. You can think of an array as a collection of pieces of data, most typically in our work the data we store in an array is a float. You have already seen arrays in other areas of mathematics: a one-dimensional (1-D) array you can think of as a vector, and a 2-D array is a matrix. We can generalize from there by thinking of a 3-D array is a vector of matrices, and so on. It is not too common in our work to go beyond a 3-D array, but one can define these more exotic data structures.

In the following code, we make a vector and a matrix. The first line tells Python that we want to use NumPy, but we do not want to type `numpy` every time we need to use a function from the library; we abbreviate to `np`.

```
In [1]: import numpy as np
        a_vector = np.array([1,2,3,4])
        a_matrix = np.array([(1,2,3),(4,5,6),(7,8,9)])
        print("The vector",a_vector)
        print("The matrix\n",a_matrix)

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

Now that we have defined arrays, we want to work with them. Arrays have several “attributes” that you can use to find out information regarding a particular array. The following code blocks explore these attributes, as noted in the comments:

```
In [2]: #shape tells you the shape
        print("The shape of a_vector is ", a_vector.shape)
        print("The shape of a_matrix is ", a_matrix.shape)

The shape of a_vector is  (4,)
The shape of a_matrix is  (3, 3)

In [3]: #ndim tells you the dimensionality of an array
        print("The dimension of a_vector is ", a_vector.ndim)
        print("The dimension of a_matrix is ", a_matrix.ndim)

The dimension of a_vector is  1
The dimension of a_matrix is  2

In [4]: #size is the total number of elements = the product of
        # the number of elements in each dimension
        print("The size of a_vector is ", a_vector.size,"= ",
              a_vector.shape[0])
        print("The size of a_matrix is ", a_matrix.size,"=",
              a_matrix.shape[0],"*",a_matrix.shape[1])
```

```
The size of a_vector is  4 =  4
The size of a_matrix is  9 = 3 * 3
```

For an existing array, you can change the shape after creating it. You can "reshape" an array to have different dimensions as long as the size of the array does not change. Here is an example:

```
In [5]: A = np.array([2,4,6,8])
        print("A is now a vector",A)
        A = A.reshape(2,2)
        print("A is now a matrix\n",A,"\nSorcery!")

A is now a vector [2 4 6 8]
A is now a matrix
 [[2 4]
 [6 8]]
Sorcery!
```

Notice how I needed to assign `A` with the reshaped array in the third line of code.

4.1.1 Creating Arrays in Neat Ways

In the examples above, we created arrays by specifying the value of each element in the array explicitly. We would like to have a way to define an array without having to type in each element. This is especially useful when we want to fill an array with thousands or millions of elements.

The function `arange` is a NumPy variant of `range`, which we saw earlier. The difference is that the function will create a NumPy array based on the parameters passed to `arange`.

```
In [6]:  #let's make a vector from 0 to 2*pi in intervals of 0.1
         dx = 0.1
         X = np.arange(0,2*np.pi,dx)
         print(X)

[ 0.   0.1  0.2  0.3  0.4  0.5  0.6  0.7  0.8  0.9  1.   1.1  1.2  1.3  1.4
  1.5  1.6  1.7  1.8  1.9  2.   2.1  2.2  2.3  2.4  2.5  2.6  2.7  2.8  2.9
  3.   3.1  3.2  3.3  3.4  3.5  3.6  3.7  3.8  3.9  4.   4.1  4.2  4.3  4.4
  4.5  4.6  4.7  4.8  4.9  5.   5.1  5.2  5.3  5.4  5.5  5.6  5.7  5.8  5.9
  6.   6.1  6.2]
```

We can also generate a fixed number of equally spaced points between fixed endpoints (a linearly increasing set of points) with the `linspace` function.

```
In [7]:  X = np.linspace(start = 0, stop = 2*np.pi, num = 62)
         print(X)

[ 0.          0.10300304  0.20600608  0.30900911  0.41201215  0.51501519
  0.61801823  0.72102126  0.8240243   0.92702734  1.03003038  1.13303342
  1.23603645  1.33903949  1.44204253  1.54504557  1.64804861  1.75105164
  1.85405468  1.95705772  2.06006076  2.16306379  2.26606683  2.36906987
  2.47207291  2.57507595  2.67807898  2.78108202  2.88408506  2.9870881
  3.09009113  3.19309417  3.29609721  3.39910025  3.50210329  3.60510632
```

```
 3.70810936  3.8111124   3.91411544  4.01711848  4.12012151  4.22312455
 4.32612759  4.42913063  4.53213366  4.6351367   4.73813974  4.84114278
 4.94414582  5.04714885  5.15015189  5.25315493  5.35615797  5.459161
 5.56216404  5.66516708  5.76817012  5.87117316  5.97417619  6.07717923
 6.18018227  6.28318531]
```

Notice how it starts and ends exactly where I told it to. The `linspace` function is very useful, and we will use it extensively.

BOX 4.1 NUMPY PRINCIPLE

The function

```
np.linspace(start, stop, num)
```

creates a NumPy array of length `num` starting at `start` and ending at `stop`.

There are other special arrays that you might want to define. Defining arrays to be all zeros or ones can be very useful for initializing arrays to a fixed value:

```
In [8]:  zero_vector = np.zeros(10) #vector of length 10
         zero_matrix = np.zeros((4,4)) #4 by 4 matrix
         print("The zero vector:",zero_vector)
         print("The zero matrix\n",zero_matrix)

The zero vector: [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
The zero matrix
 [[ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]]
```

```
In [9]: ones_vector = np.ones(10) #vector of length 10
        ones_matrix = np.ones((4,4)) #4 by 4 matrix
        print("The ones vector:",ones_vector)
        print("The ones matrix\n",ones_matrix)

The ones vector: [ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.]
The ones matrix
 [[ 1.  1.  1.  1.]
 [ 1.  1.  1.  1.]
 [ 1.  1.  1.  1.]
 [ 1.  1.  1.  1.]]
```

There are times when we want to define an array filled with random values. For these purposes NumPy has an additional module called `random` that generalizes the library we used earlier. The NumPy `random` module can create matrices with random entries between 0 and 1 with the function `np.random.rand`, random entries between endpoints using `np.random.uniform`, and random integers with `np.random.randint`.

```
In [10]: random_matrix = np.random.rand(2,3) #random 2 x 3 matrix
         print("Here’s a random 2 x 3 matrix\n",random_matrix)
```

```
        print("Another example")

        #make a random array between two numbers
        print(np.random.uniform(low=-5,high=6,size=(3,3)))

        #make random integers
        print(np.random.randint(low=1,high=6,size=10))
```

```
Here's a random 2 x 3 matrix
 [[ 0.13097005  0.52015702  0.94753032]
 [ 0.99428635  0.10108597  0.62091224]]
Another example
[[ 1.86706869 -1.76316942 -1.88067072]
 [-2.13386359  2.84442703 -2.04880365]
 [-2.06079464 -2.17357025 -3.70912541]]
[5 3 3 2 2 3 3 4 4 4]
```

It is also possible to automatically generate an identity matrix:

```
In [11]: #3 x 3 identity matrix
         identity33 = np.identity(3)
         print(identity33)
```

```
[[ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]]
```

There are many other ways to define matrices using NumPy, but the ones we have covered will be the most useful for us.

BOX 4.2 NUMPY PRINCIPLE

To create NumPy vectors in simple ways you can use the following functions

```
#create a vector of length l
with entries 1.0
np.ones(l)
#create a vector of length l
with entries 0.0
np.zeros(l)
#create a vector of length l
w/random values in [0,1]
np.random.rand(l)
```

BOX 4.3 NUMPY PRINCIPLE

To create NumPy matrices in simple ways you can use the following functions

```
#create an l by m matrix
with entries 1.0
np.ones((l,m))
#create an l by m matrix
with entries 0.0
np.zeros((l,m))
#create an l by m matrix
w/random values in [0,1]
np.random.rand(l,m)
#create an l by l identity matrix
np.identity(l)
```

4.1.2 Operations on Arrays

Previously, we talked about operator overloading where we can apply common operators, such as arithmetic operations, to objects other than numbers. We would like to be able to take several arrays and do things like add them, multiply by a scalar, and perform other linear algebra operations on them. NumPy has defined most of these operations for us by overloading the common operators. We will explore these operations here.

NumPy defines arithmetic operations mostly in the way that you would expect. For example, addition is easily accomplished provided that the arrays are of the same size and shape. We will begin by demonstrating operations on vectors.

```
In [12]: #vector addition
         x = np.ones(3) #3-vector of ones
         y = 3*np.ones(3)-1 #3-vector of 2's
         print(x,"+",y,"=",x+y)
         print(x,"-",y,"=",x-y)

[ 1.  1.  1.] + [ 2.  2.  2.] = [ 3.  3.  3.]
[ 1.  1.  1.] - [ 2.  2.  2.] = [-1. -1. -1.]
```

Multiplication and division are "element-wise"; this means that the elements in the same position are multiplied together:

```
In [13]: y = np.array([1.0,2.0,3.0])
         print(x,"*",y,"=",x*y)
         print(x,"/",y,"=",x/y)

[ 1.  1.  1.] * [ 1.  2.  3.] = [ 1.  2.  3.]
[ 1.  1.  1.] / [ 1.  2.  3.] = [ 1.          0.5         0.33333333]
```

If you want the dot product, you have to use the `dot` function:

```
In [14]: print(x,".",y,"=",np.dot(x,y))

[ 1.  1.  1.] . [ 1.  2.  3.] = 6.0
```

Matrices work about the same way as vectors, when it comes to arithmetic operations.

```
In [15]: silly_matrix = np.array([(1,2,3),(1,2,3),(1,2,3)])
         print("The sum of\n",identity33,"\nand\n",
               silly_matrix,"\nis\n",identity33+silly_matrix)

The sum of
 [[ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]]
and
 [[1 2 3]
 [1 2 3]
 [1 2 3]]
is
 [[ 2.  2.  3.]
 [ 1.  3.  3.]
 [ 1.  2.  4.]]
```

Multiplication and division are also element-wise:

```
In [16]: identity33 * silly_matrix
```

```
Out[16]: array([[ 1.,  0.,  0.],
                [ 0.,  2.,  0.],
                [ 0.,  0.,  3.]])
```

```
In [17]: identity33 / silly_matrix
```

```
Out[17]: array([[ 1.        ,  0.        ,  0.        ],
                [ 0.        ,  0.5       ,  0.        ],
                [ 0.        ,  0.        ,  0.33333333]])
```

The `dot` function will give you the matrix product when the it is passed two matrices:

```
In [18]: print("The matrix product of\n",identity33,"\nand\n",
               silly_matrix,"\nis\n",
               np.dot(identity33,silly_matrix))
```

```
The matrix product of
 [[ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]]
and
 [[1 2 3]
 [1 2 3]
 [1 2 3]]
is
 [[ 1.  2.  3.]
 [ 1.  2.  3.]
 [ 1.  2.  3.]]
```

To compute the product of a matrix and vector, we use the `dot` function and pass it the matrix followed by the vector.

```
In [19]: #matrix times a vector
         print(silly_matrix,"times", y, "is")
         print(np.dot(silly_matrix,y))
```

```
[[1 2 3]
 [1 2 3]
 [1 2 3]] times [ 1.  2.  3.] is
[ 14.  14.  14.]
```

When we use the `dot` function, the matrices and vectors must have the appropriate sizes. If the sizes are incompatible, Python will give an error.

BOX 4.4 NUMPY PRINCIPLE

Standard arithmetic operations are overloaded so that they work on an element by element basis on NumPy arrays. This means that the arrays must have the same size. To use linear algebra operations such as matrix multiplication or the dot product of two vectors you will want to use the `np.dot(a,b)` function.

4.1.3 Universal Functions

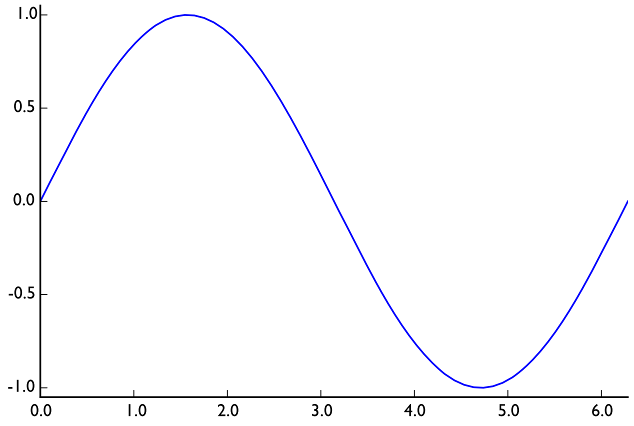
It is common that we might want to interpret a vector as a series of points to feed to a function. For example, the vector could be a list of angles we want to compute the sine of. For these, and more general situations, NumPy provides universal functions that operate on each element of an array. Common mathematical functions are defined in this way, and are used in a similar way to the functions in the `math` module we used previously:

```
In [20]: #recall we defined X as a linspace from 0 to 2pi
         print(X)
         #taking the sin(X) should be one whole sine wave
         print(np.sin(X))
```

```
[ 0.          0.10300304  0.20600608  0.30900911  0.41201215  0.51501519
  0.61801823  0.72102126  0.8240243   0.92702734  1.03003038  1.13303342
  1.23603645  1.33903949  1.44204253  1.54504557  1.64804861  1.75105164
  1.85405468  1.95705772  2.06006076  2.16306379  2.26606683  2.36906987
  2.47207291  2.57507595  2.67807898  2.78108202  2.88408506  2.9870881
  3.09009113  3.19309417  3.29609721  3.39910025  3.50210329  3.60510632
  3.70810936  3.8111124   3.91411544  4.01711848  4.12012151  4.22312455
  4.32612759  4.42913063  4.53213366  4.6351367   4.73813974  4.84114278
  4.94414582  5.04714885  5.15015189  5.25315493  5.35615797  5.459161
  5.56216404  5.66516708  5.76817012  5.87117316  5.97417619  6.07717923
  6.18018227  6.28318531]
[  0.00000000e+00   1.02820997e-01   2.04552066e-01   3.04114832e-01
   4.00453906e-01   4.92548068e-01   5.79421098e-01   6.60152121e-01
   7.33885366e-01   7.99839245e-01   8.57314628e-01   9.05702263e-01
   9.44489229e-01   9.73264374e-01   9.91722674e-01   9.99668468e-01
   9.97017526e-01   9.83797952e-01   9.60149874e-01   9.26323968e-01
   8.82678798e-01   8.29677014e-01   7.67880446e-01   6.97944155e-01
   6.20609482e-01   5.36696194e-01   4.47093793e-01   3.52752087e-01
   2.54671120e-01   1.53890577e-01   5.14787548e-02  -5.14787548e-02
  -1.53890577e-01  -2.54671120e-01  -3.52752087e-01  -4.47093793e-01
  -5.36696194e-01  -6.20609482e-01  -6.97944155e-01  -7.67880446e-01
  -8.29677014e-01  -8.82678798e-01  -9.26323968e-01  -9.60149874e-01
  -9.83797952e-01  -9.97017526e-01  -9.99668468e-01  -9.91722674e-01
  -9.73264374e-01  -9.44489229e-01  -9.05702263e-01  -8.57314628e-01
  -7.99839245e-01  -7.33885366e-01  -6.60152121e-01  -5.79421098e-01
  -4.92548068e-01  -4.00453906e-01  -3.04114832e-01  -2.04552066e-01
  -1.02820997e-01  -2.44929360e-16]
```

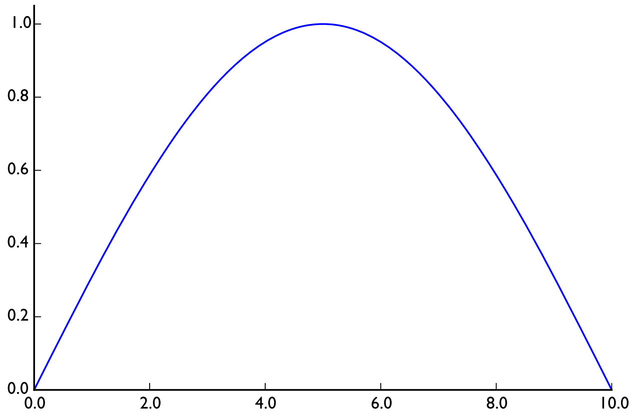
Universal functions are useful for plotting when we define a vector of points for the x axis and apply the function to get the y axis. Using `matplotlib`, which we cover extensively in a later section in this chapter, we can use a universal function to plot the sine function:

```
In [21]: import matplotlib.pyplot as plt
         plt.plot(X,np.sin(X));
```



Therefore, if we wanted to plot the fundamental mode of a slab reactor of width 10, we could combine the arithmetic operators and universal functions defined by NumPy in the following way:

```
In [22]: X = np.linspace(0,10,100)
         Y = np.sin(np.pi*X/10)
         plt.plot(X,Y);
```



4.1.4 Copying Arrays and Scope

The assignment operator = behaves differently for NumPy arrays than for other data types we have used. When you assign a new variable name to an existing array, it is the same as giving two names for the object. It does not copy the array into a new array.

```
In [23]: a = np.array([1.0,2,3,4,5,6])
         print(a)
         #this will make a and b different names for the same array
         b = a
         #changing b at position 2, also changes a
         b[2] = 2.56
         print("The value of array a is",a)
         print("The value of array b is",b)

[ 1.  2.  3.  4.  5.  6.]
The value of array a is [ 1.    2.    2.56  4.    5.    6.  ]
The value of array b is [ 1.    2.    2.56  4.    5.    6.  ]
```

The reason for this behavior is that the array could have thousands or millions of elements, and creating new arrays in a pell mell fashion could quickly fill up the computer memory and slow down the program. To make a real copy of the entire elements of an array you need to explicitly tell Python that you want to make a copy by using the `copy` function on the array. This assures that you only create copies of an array when you truly want that to happen. We can modify the previous code snippet to copy the array:

```
In [24]: a = np.array([1.0,2,3,4,5,6])
         print(a)
         #this will make a and b different copies for the same array
         b = a.copy()
         #changing b at position 2, will not change a
         b[2] = 2.56
         print("The value of array a is",a)
         print("The value of array b is",b)

[ 1.  2.  3.  4.  5.  6.]
The value of array a is [ 1.  2.  3.  4.  5.  6.]
The value of array b is [ 1.    2.    2.56  4.    5.    6.  ]
```

BOX 4.5 NUMPY PRINCIPLE

To copy a NumPy array named `origArray` to the NumPy array `copyArray` use the syntax

```
copyArray = origArray.copy()
```

The syntax

```
sameArray = origArray
```

will give an additional name, `sameArray`, for the array `origArray`.

Typically, when you pass a variable to a function it copies that variable into the function's memory scope. This does not happen with NumPy arrays. When you pass an array to a function, the function does not copy the array, it just assigns that array another name as in the previous example. This means that if the NumPy array is changed inside the function, it is also changed outside the function.

```
In [25]: def devious_function(func_array):
             #changes the value of array passed in
             func_array[0] = -1.0e6

         a = np.array([1.0,2,3,4,5,6])
         print("Before the function a =",a)
         devious_function(a)
         print("After the function a =",a)

Before the function a = [ 1.  2.  3.  4.  5.  6.]
After the function a =
[ -1.00000000e+06   2.00000000e+00   3.00000000e+00
  4.00000000e+00   5.00000000e+00   6.00000000e+00]
```

This is different than what we saw previously for passing floats, and integers to functions. The difference is that a NumPy array is a mutable object that could possibly be large. The technicalities are not of much interest here, but, because a NumPy array could be millions of elements, it is best for efficient memory usage if functions do not make multiple copies of millions of elements. This is same rationale that leads to the behavior of the assignment operator not automatically copying the entire array into a new array. For example, if I had an array with one billion elements, passing that array into a function could take a long time to make all of those copies into a new array, before the function even begins to do its work.

4.1.5 Indexing, Slicing, and Iterating

Oftentimes we will want to access or modify more than one element of an array at a time. We can do this by slicing. Slicing, in many ways, mirrors what we did with strings before. One feature we have not discussed is the plain : operator without a number on either side. This will give all the values in a particular dimension, as seen below.

```
In [26]: #bring these guys back
         a_vector = np.array([1,2,3,4])
         a_matrix = np.array([(1,2,3),(4,5,6),(7,8,9)])
         print("The vector",a_vector)
         print("The matrix\n",a_matrix)
         #single colon gives everything
         print("a_vector[:] =",a_vector[:])
         #print out position 1 to position 2 (same as for lists)
         print("a_vector[1:3] =",a_vector[1:3])
         print("For a matrix, we can slice in each dimension")
         #every column in row 0
         print("a_matrix[0,:] =",a_matrix[0,:])
         #columns 1 and 2 in row 0
         print("a_matrix[0,1:3] =",a_matrix[0,1:3])
         #every row in column 2
         print("a_matrix[:,2] =",a_matrix[:,2])

The vector [1 2 3 4]
The matrix
 [[1 2 3]
 [4 5 6]
 [7 8 9]]
a_vector[:] = [1 2 3 4]
a_vector[1:3] = [2 3]
For a matrix, we can slice in each dimension
a_matrix[0,:] = [1 2 3]
a_matrix[0,1:3] = [2 3]
a_matrix[:,2] = [3 6 9]
```

We can also use a `for` loop to iterate over an array. If the array is a vector, the iteration will be over each element. Iteration over a matrix will give you everything in a row. If you want to iterate over the columns of the matrix, take the transpose of the matrix when iterating.

```
In [27]: a_matrix = np.array([(1,2,3),(4,5,6),(7,8,9)])
         count = 0
         for row in a_matrix:
             print("Row",count,"of a_matrix is",row)
             count += 1

         count = 0
         for column in a_matrix.transpose():
             print("Column",count,"of a_matrix is",column)
             count += 1
```

```
Row 0 of a_matrix is [1 2 3]
Row 1 of a_matrix is [4 5 6]
Row 2 of a_matrix is [7 8 9]
Column 0 of a_matrix is [1 4 7]
Column 1 of a_matrix is [2 5 8]
Column 2 of a_matrix is [3 6 9]
```

To iterate over every element in the matrix you'll need two for loops: one to get the rows, and another to iterate over each element in the row. This is an example of nested `for` loops: a `for` loop with another `for` loop inside.

```
In [28]: a_matrix = np.array([(1,2,3),(4,5,6),(7,8,9)])
         row_count = 0
         col_count = 0
         for row in a_matrix:
             col_count = 0
             for col in row:
                 print("Row",row_count,"Column",col_count,
                       "of a_matrix is",col)
                 col_count += 1
             row_count += 1
```

```
Row 0 Column 0 of a_matrix is 1
Row 0 Column 1 of a_matrix is 2
Row 0 Column 2 of a_matrix is 3
Row 1 Column 0 of a_matrix is 4
Row 1 Column 1 of a_matrix is 5
Row 1 Column 2 of a_matrix is 6
Row 2 Column 0 of a_matrix is 7
Row 2 Column 1 of a_matrix is 8
Row 2 Column 2 of a_matrix is 9
```

4.1.6 NumPy and Complex Numbers

NumPy can handle complex numbers without much difficulty. A NumPy array can be specified to contain complex numbers by adding the parameter `dtype = “complex”` to the creation. For example, an array full of `0+0j` can be created by

```
In [29]: cArray = np.zeros(5, dtype = "complex")
         print(cArray)

[ 0.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j]
```

The `dtype` argument tells NumPy what datatype will be in the array. This will work with other methods we discussed for creating arrays `ones`, `identity`, and `array`.

You do not have to specify `dtype` if it is obvious that you are using complex numbers. For example, this will create an array of complex numbers without the `dtype` command

```
In [30]: cArray2 = np.array([1+1j,-1])
         print(cArray2)

[ 1.+1.j -1.+0.j]
```

The time that you have to be careful with complex numbers and NumPy is when you have a non-complex array that evaluates to a complex number inside a function. For instance, the square root of a negative number is imaginary. Calling `np.sqrt` on a negative float will give an error, unless the `dtype = “complex”` argument is provided. If this argument is not provided, the result may be “not a number” or `nan`, and Python may throw an error. Here is an example of the wrong way and the right way to take a square root of an array that contains negative numbers:

```
In [31]: fArray = np.array([-1,1])
         print("Wrong way gives", np.sqrt(fArray))
         print("Right way gives", np.sqrt(fArray, dtype = "complex"))
         print(cArray2)

Wrong way gives [ nan   1.]
Right way gives [ 0.+1.j  1.+0.j]
```

This is an important consideration when using built-in mathematical functions with NumPy.

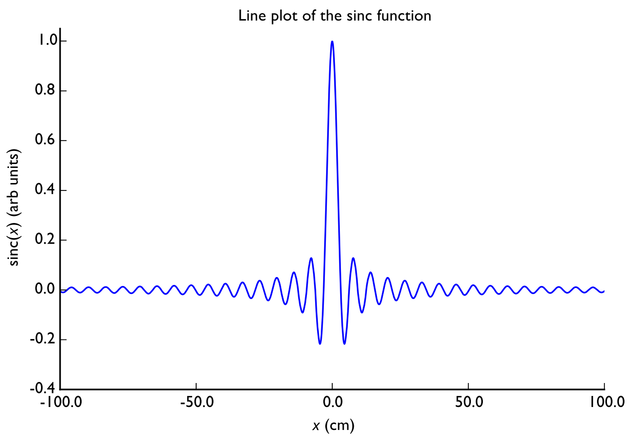
We have now covered the details of NumPy that we will need for our numerical investigations. Having numbers in an array is an important step in engineering analysis, but understanding what comes out of a calculation can be much easier if we can visualize the results. In the next section we discuss a method for this visualization.

4.2 MATPLOTLIB BASICS

Matplotlib is a library for Python that allows you to plot the arrays from NumPy, as well as many other features. It is designed to be intuitive and easy to use, and it mimic the plotting interface of MATLAB, a widely used toolkit and language for applied mathematics and computation. Therefore, if you have experience with MATLAB, using Matplotlib will be very familiar.

In the following example, a plot is created, and properly annotated using Matplotlib.

```
In [32]: import matplotlib.pyplot as plt
         import numpy as np
         #make a simple plot
         x = np.linspace(-100,100,1000)
         y = np.sin(x)/x
         #plot x versus y
         plt.plot(x,y)
         #label the y axis
         plt.ylabel("sinc(x) (arb units)");
         #label the x axis
         plt.xlabel("x (cm)")
         #give the plot a title
         plt.title("Line plot of the sinc function")
         #show the plot
         plt.show()
```

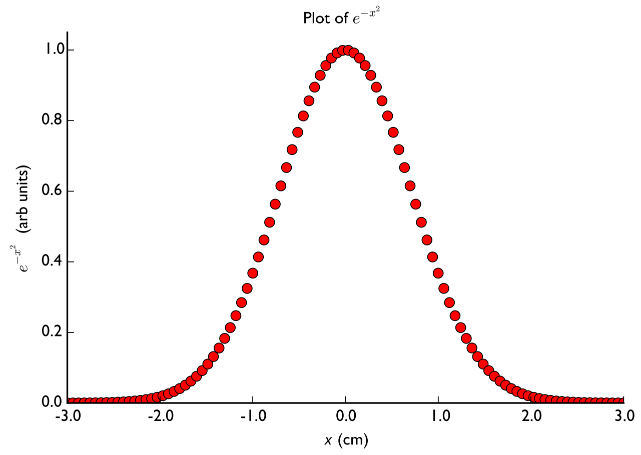


Reviewing what the code above did we see that the `plot` function took in two arguments that served as the data for the *x* and *y* axes. The `ylabel` and `xlabel` functions take in a string to print on the respective axes; the `title` function prints a string as the plot title. Finally, the `show` function tells Python to show us the plot. Notice how I labeled each axis and even gave the plot a title. I also included the units for each axis.

4.2.1 Customizing Plots

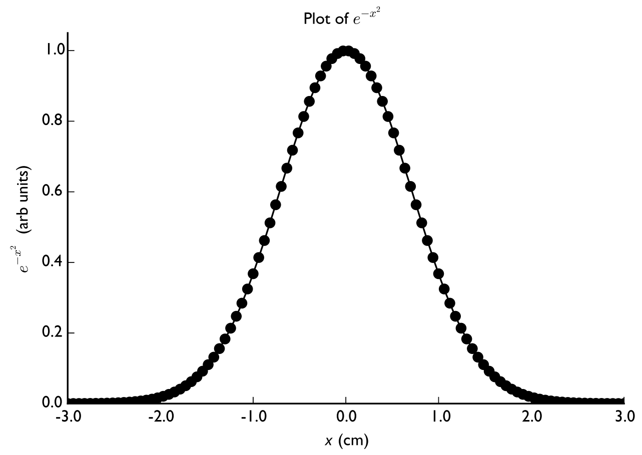
You can also change the plot type. Here we will change from line plotting to using red dots:

```
In [33]: x = np.linspace(-3,3,100)
         y = np.exp(-x**2)
         plt.plot(x,y,"ro"); #red dots on the plot
         plt.ylabel("$e^{-x^2}$ (arb units)");
         plt.xlabel("x (cm)")
         plt.title("Plot of $e^{-x^2}$")
         plt.show()
```



We could also use dots and a line in our plot:

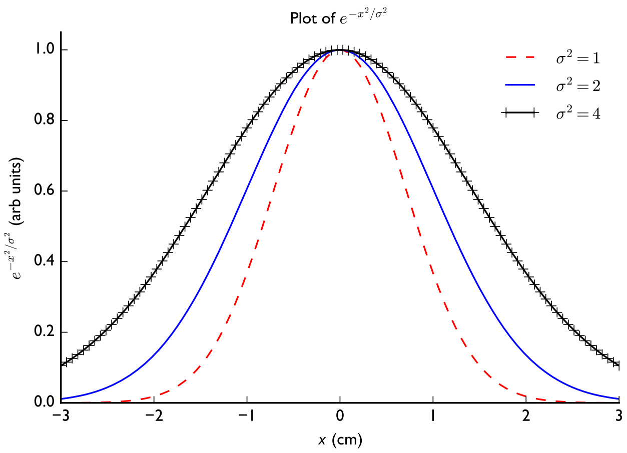
```
In [34]: x = np.linspace(-3,3,100)
         y = np.exp(-x**2)
         plt.plot(x,y,"ko-"); #black dots and a line on the plot
         plt.ylabel("$e^{-x^2}$ (arb units)");
         plt.xlabel("x (cm)")
         plt.title("Plot of $e^{-x^2}$")
         plt.show()
```



Notice these examples have included LaTeX mathematics in the labels by enclosing it in dollar signs. LaTeX mathematics is a markup language for mathematical characters. For example, you can make characters superscript by enclosing the characters in curly braces preceded by ^. In LaTeX to use Greek letters you use a backslash before the name of the letter, and other usually obvious characters. We will use LaTeX, extensively to annotate figures throughout the work.

It is also possible to plot several lines on a plot and include a legend. The legend text is passed into the `plot` function through the parameter `label` which takes in a string, potentially with LaTeX. Calling the `legend` function will add the legend to the figure. Here is an example of multiple lines:

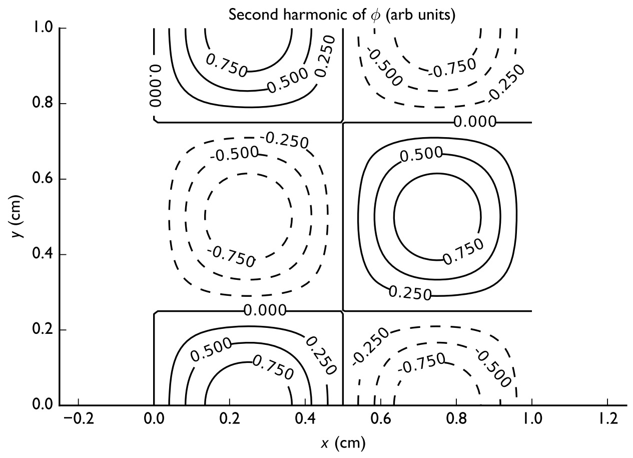
```
In [35]: x = np.linspace(-3,3,100)
         y = np.exp(-x**2)
         y1 = np.exp(-x**2/2)
         y2 = np.exp(-x**2/4)
         plt.plot(x,y,marker="", color="r",
                         linestyle="--", label="$\sigma^2 = 1$")
     plt.plot(x,y1,color="blue",
                         label="$\sigma^2 = 2$")
     plt.plot(x,y2,color="black",
                   marker = "+", label="$\sigma^2 = 4$")
         plt.ylabel("$e^{-x^2/\sigma^2}$ (arb units)");
         plt.xlabel("x (cm)")
         plt.title("Plot of $e^{-x^2/\sigma^2}$")
         plt.legend()
         plt.show()
```



With MatPlotLib it is also possible to make plots that are more than just lines. This example here we make a contour plot of a function of two variables. We will not go into detail about these plots here, rather we will demonstrate additional plotting features as we need them.

```
In [36]: phi_m = np.linspace(0, 1, 100)
         phi_p = np.linspace(0, 1, 100)
```

```
X,Y = np.meshgrid(phi_p, phi_m)
Z = np.sin(X*2*np.pi)*np.cos(Y*2*np.pi)
CS = plt.contour(X,Y,Z, colors='k')
plt.clabel(CS, fontsize=9, inline=1)
plt.xlabel("x (cm)");
plt.ylabel("y (cm)");
plt.title("Second harmonic of $\phi$ (arb units)");
```



FURTHER READING

For a less nuclear engineering-specific tutorial on NumPy see http://wiki.scipy.org/. For a list, see http://matplotlib.org/users/pyplot_tutorial.html. LaTeX has several books available that are useful references such as [5,6]. Additionally, there are many tutorials available on the Internet, including one from the LaTeX project https://www.latex-project.org/.

PROBLEMS

Short Exercises

4.1. Write a simple dot product calculator. Ask the user for the size of the vector, and then to input the value for each element in the vector. Print to the user the value of the dot product of the two vectors.

4.2. Ask the user for a matrix size, `N`. Print to the user a random matrix of size `N` by `N`. Then print to the user the vector that contains the sum of each column.

4.3. Ask the user for a vector size, `N`. Print to the user a random vector of size `N`. Then print the sorted matrix. NumPy has a function for sorting a vector: `x.sort()`, where `x` is the name of a vector.

4.4. Ask the user for a vector size, `N`. Print to the user a random vector of size `N`. Then print the maximum value of the vector. NumPy has a function that returns the index of the maximum value: `x.argmax()`, where `x` is the name of a vector.

4.5. Ask the user for a vector size, `N`. Print to the user a random vector of size `N` where the elements of `N` are from a normal distribution with mean 0 and standard deviation 1. To generate this vector you should use `np.random.normal`. Then print the mean value of the vector and the standard deviation. NumPy has a function that returns the mean value and standard deviation: `x.mean()` and `x.std()`, where `x` is the name of a vector.

4.6. Ask the user for a vector size, `N`. Print to the user two random vectors of size `N`; call these vectors `x` and `y`. If we consider these vectors being the coordinates of points (x_i, y_i), compute the matrix $\mathbf{r}$ that contains the distances between each point and each of other points. This matrix will have elements

$$r_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}.$$

The diagonal of this matrix should be all zeros.

Programming Projects

1. Inhour Equation

The inhour equation (short for inverse hour) describes the growth or decay of the neutron population in a reactor as described by the point kinetics equations. The equation relates the reactivity, ρ, the mean generation time, Λ, the fraction of fission neutrons born from each of six delayed neutron precursor groups, β_i, and the decay constant for those groups, λ_i. The inhour equation is

$$\rho = s\left(\Lambda + \sum_{i=1}^{6} \frac{\beta_i}{s + \lambda_i}\right).$$

The seven values of s that satisfy this equation are used to express the neutron population as function of time, $n(t)$, as

$$n(t) = \sum_{\ell=1}^{7} A_\ell e^{s_\ell t}.$$

The decay constant of a delayed neutron precursor group is related to the half-life for neutron emission by the group is

$$\lambda_i = \frac{\ln 2}{t_{1/2}}.$$

Common values of the necessary constants in this equation are [7]: $\Lambda = 5 \times 10^{-5}$ s

$$\beta = \{0.00021, 0.00142, 0.00128, 0.00257, 0.00075, 0.00027\},$$

where the total delayed fraction $\bar{\beta} = 0.0065$, and

$$t_{1/2} = \{56, 23, 6.2, 2.3, 0.61, 0.23\} \text{ s}.$$

Using this data, plot the right-hand side of the inhour equation as a function of s, and plot a horizontal line corresponding to ρ to graphically illustrate the roots of the inhour equation. Do this for $\rho = -1, 0, 0.1\bar{\beta}, \bar{\beta}$, and discuss the results. Make sure that the scale of your plot makes sense given that there will be singularities in the plot.

2. *Fractal Growth*

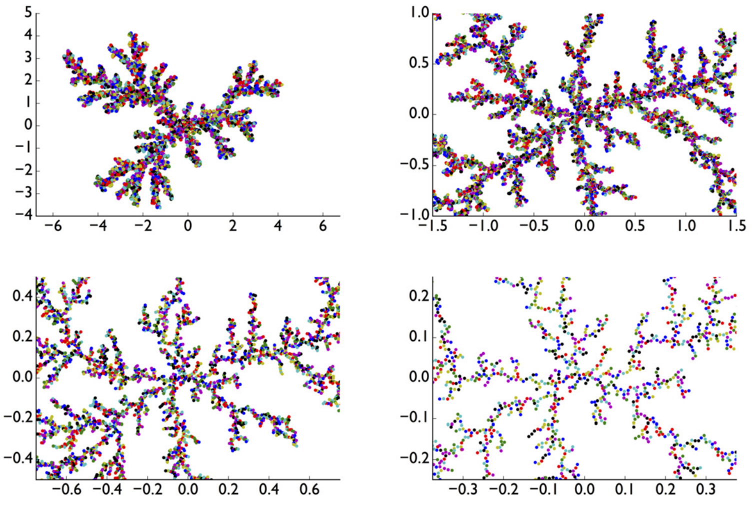
In this problem you will code a program that grows a cluster of particles that stick together using a diffusion process known as a random walk. The resulting structure will be a fractal, that is a structure that is self-similar, where the structure looks the same at every scale. To build these structures we start with a single particle at the position $(0,0)$, then introduce a particle at "infinity". The new particle undergoes a random walk where it jumps a random distance in a random direction. The particle undergoes random jumps until it strikes the center particle. Then, we repeatedly introduce particles at "infinity" and follow each until it sticks to the existing particles.

To make the algorithm work quickly, we will define "infinity" by placing a particle randomly on a circle that circumscribes the existing structure. We also, allow the distance traveled by the particle in each step to be the minimum distance between the particle and the structure. The following code will perform this algorithm, but there are expressions missing in key positions. The comments will tell you how to fill them in; these are denoted by `??`.. Get the code working, and make plots of the resulting structures with matplotlib. Make figures of various sizes by changing the value of `N`, i.e., `N = 100, 500, 1000, 500, 10000`, and even higher values if the code is fast enough. Comment on the similarity of the structures of different size.

```
import matplotlib.pyplot as plt
import numpy as np
#initialize list of particles
#column 0 is x position
#column 1 is y position
particles = np.zeros([1,2])
#how many particles to simulate
N = 500
#how close do we need to be to "stick"
tol = 1.0e-2
#how many steps can a particle take before we stop tracking
maxsteps = 1e3
#radius of structure, initially small
r = 2.0*tol
#pick the angle on the circle in [0,2pi]
theta = np.random.uniform(low=0,
                          high=2*np.pi,
                          size = 1)
for i in range(N):
    #make the initial position of the new particles
    newposx = r*np.sin(theta)
```

```
newposy = r*np.cos(theta)
steps = 0
#how close is the new particle to the cluster
dist = r
while (dist > tol) and (steps<maxsteps):
    #how close is the new particle to the cluster
    dist_vect = np.sqrt(
                (particles[:,0]-newposx)**2 + (particles[:,1]-newposy)**2)
    dist = np.min(dist_vect)
    #compute the jump distance randomly in [0,dist]
    rho = ??
    #compute the direction of the jump randomly in [0,2 pi]
    theta = ??
    #move the particle
    newposx += ??
    newposy += ??
    steps += 1
#if the while loop is exited with fewer steps than
#the maximum, then add the particle to the list
#otherwise forget it
if (steps < maxsteps):
    tmp = np.ndarray(shape = [1,2])
    tmp[0,0] = newposx
    tmp[0,1] = newposy
    particles = np.append(particles, tmp,axis=0)
#make the starting point for the next particle
#by calculating the radius and angle
r = (1+tol)*np.max(np.sqrt(particles[:,0]**2 + particles[:,1]**2))
theta = ??
```

The figures below show the similarity of scales and were generated with this code, and `N` $= 5 \times 10^5$.



3. *Charges in a Plane*

Consider a collection of N charged point particles distributed in the xy plane. The particles have known masses, m, and charges, q. The Coulomb force on particle i is given by

$$F_i = \sum_{j=1,\, j\neq i}^{N} \frac{q_i q_j}{r^2},$$

where r is the distance between particle i and j. The x and y components of the force are

$$F_i^x = \sum_{j=1,\, j\neq i}^{N} \frac{q_i q_j}{r^2}\frac{x_i - x_j}{r}, \qquad F_i^y = \sum_{j=1,\, j\neq i}^{N} \frac{q_i q_j}{r^2}\frac{y_i - y_j}{r}.$$

From the forces, we can compute the acceleration in each direction using $\vec{F_i} = m_i\vec{a}$. Given the equations of motion of a particle

$$v_{x,i}^{l+1} = v_{x,i}^{l} + \Delta t a_{x,i}^{l}, \qquad v_{y,i}^{l+1} = v_{y,i}^{l} + \Delta t a_{y,i}^{l},$$
$$x_i^{l+1} = x_i^{l} + \Delta t v_{x,i}^{l+1}, \qquad y_i^{l+1} = y_i^{l} + \Delta t v_{y,i}^{l+1},$$

for $l = 1 \ldots L$ and $v_{x,i}^0 = v_{y,i}^0 = 0$. Where the superscripts indicate the time level, i.e., $x_i^l = x(l\Delta t)$.

Write a Python code that solves the above problem using $L = 1000$ and $\Delta t = 0.001$, and the following table of masses and initial positions as defined in the following code snippet:

```
#Code to advect point charges using Coulomb's law
import numpy as np
import matplotlib.pyplot as plt

#set up initial values of x,y,vx,vy,Fx,Fy
N = 20
x = np.zeros(N)
x[0:(N//2)] = 0*np.linspace(-N,-5,N//2)
x[(N//2):N] = np.linspace(1,10,N//2)
y = np.zeros(N)
y[0:(N//2)] = np.linspace(-10,10,N//2)
y[(N//2):N] = np.random.uniform(low = -1e-3, high = 1e-3, size = N//2)
mass = np.zeros(N)
mass[0:(N//2)] = 1e8
mass[(N//2):N] = 1
q = np.zeros(N)
q[0:(N//2)] = 10
q[(N//2):N] = 1
vx = np.zeros(N);
vx[0:(N//2)] = 0.0
vx[(N//2):N] = -10.0
vy = np.zeros(N)
Fx = np.zeros(N)
Fy = np.zeros(N)
```

To solve this problem you will need three loops: one to compute the forces, one to compute the velocities, and one to compute the new positions. These loops will be nested inside a loop that keeps track of the time.

Plot the solution at the final time for varying values of `N`: 20, 50, 100. Explain the differences in the results.