9. Introduction to the Numpy library.

We have briefly introduced the Numpy library in earlier notebooks where we have seen the similarity between the standard python list and its numpy equivalent, what we will now formally describe as a numpy array. While useful, the list object that we have used so far is limited and not particularly useful for advanced mathematical or numerical methods. As we shall see, the use of multi-dimensional arrays sits naturally as way of doing much of the formal mathematics you have been doing so far. For example a simple 1D array with three components is a natural way of defining the $\mathbf{i}, \mathbf{j}, \mathbf{k}$ components of a vector in 3 dimensions. Furthermore, if we can define a series of functions that, for example, calculate the dot and cross product of two vectors based on these arrays, then we can simplify greatly many of the numerical caclulations we wish to carry out. The python numpy library (or for more advanced functions the scipy library) has become the de facto library of choice for these matrix and linear algebra methods.

We do not have time, in this course to give a comprehensive discussion of the numpy library but at the end of this notebook you should begin to get a feel for its use and the type of problems you might solve. At the heart of the library is the concept of the numpy array and from this point we will use these arrays exclusively rather than the basic python list we have discussed so far.

For detailed information about the numpy and scipy libraries goto to either https://www.scipy.org/ (https://www.scipy.org/) or https://numpy.org/ (https://numpy.org/).

In this notebook you will learn,

- how to create and maniuplate 1D numpy arrays by direct use of the array index and by vectorisation,
- how to slice, dice and append 1D arrays using numpy,
- how to use 3 element 1D numpy arrays as vectors and to generate dot and cross products,
- how to do statistical operations on 1D arrays, i.e. mean, standard deviation etc.,
- how to create 2D (and higher dimensional arrays),
- how to index higher dimensional arrays,
- about the extension of vectorisation to higher dimensional arrays.
- how numpy may be used to do basic matrix calculations.

Remember that in order to use the numpy library it must be imported. It is almost taken as standard that it is named _np_ although this is more a matter of convention rather than a requirement.

i.e.

import numpy as np

9.1 The 1D numpy array.

9.1.1 Creating 1D numpy arrays

We have already come across the numpy 1D array in earlier notebooks but we will re-iterate some of the main points here. There are several ways in which we can create a numpy array. A few are shown below. Run the code block below and make sure you understand the output.

In [ ]:

```
import numpy as np

a=np.array([1,2,3,4,5])
b=np.zeros(5)
c=np.ones(6)
d=np.arange(10)
#
# Now print out the arrays that have been created
#
print("a= ",a)
print("b= ",b)
print("c= ",c)
print("d= ",d)
#
print(type(a))
```

In these examples, note that by default, the function arange creates a list of integers, whereas *ones* and *zeros* generate floats by default. As we saw previously we can overwrite these defaults by using the *dtype* declaration the function calls. i.e.

In [ ]:

```
import numpy as np
e=np.zeros(5,dtype="int")
f=np.ones(6,dtype="complex")
#
print("e= ",e)
print("f= ",f)
#
g=np.empty(5,dtype="float")
print("g=", g)
```

The last statement *np.empty(5)* is unusual. In this case python reserves the memory needed to store the numbers for the array, but it does not set their value. As you see, it often returns zeros, but this does not have to be the case. Why bother? Well, it takes time to allocate values to the array so if you are using big arrays you save computer time as you should be allocating values later in the code anyway (if you have written the programme properly). There are other array creation functions that can save you time, for example, the *linspace* and *logsapce* functions shown below

In [ ]:

```
import numpy as np
h=np.linspace(1,2,11)     # note the first argument is the start, the second the last and th
m=np.logspace(1,-1,3)
#
print("h= ",h)
print("m= ",m)
```

We saw this earlier when we used *linspace* to define the _x_ coordinates for a graph.

A few other useful functions you can use are given below. Make a note of the output in this case of a 1D array. We will see how the results change later when we discuss multi-dimensional arrays.

In [ ]:

```
print("The dimension of a is ", np.ndim(a))      # find the dimension of the array
print("The size of a is ",np.size(a))            # find the size of the array
print("The shape of a is ",np.shape(a))          # find the shape of the array
```

9.1.2 Indexing and slicing arrays

In the previous notebooks you have seen how to address, by the index, the different elements of a 1D numpy array. For example, if we wish to generate the first 13 integers (including 0) squared in an array we can do:

In [ ]:

```
import numpy as np
n=13
sqs=np.zeros(n)
for i in range(n):
    sqs[i]=i**2

print(sqs)
```

There are also very powerful ways of slicing arrays in to sub-arrays. A few examples are given below where you can see how the ':' is used to subdivide the array. The easiest way to understand this is to experiment with some simple arrays of your own. The syntax is very similar to the way you can slice strings in python, so it is also useful to look up 'cutting and slicing strings in python'.

In [ ]:

```
sequence=np.arange(10,dtype="float")
print(sequence)
print(sequence[4])             # print the element with index 4, note the index starts with
i=7
print(sequence[i])             # print the element using an index variable (must be integer)
print(sequence[4:8])           # print the elements with indices between 4 and 8 (note caref
print(sequence[2::3])          # print the element with index 2 and every third element afte
print(sequence[::2])           # print the elements with index 0,2,4 ..
print(sequence[::-1])          # prints the elements in reverse order.
print(sequence[:3])            # print the first 3 elements
print(sequence[4:])            # print from the fourth element onwards
i=2
j=6
newseq=sequence[i:j]           # NB. You cou can create new arrays using these slicing funct
print(newseq)
```

Take care to make sure you fully understand this syntax for 1D arrays as, as we shall see shortly, it is readily extended to multidimensional arrays.

9.1.3 adding and removing array elements.

In most cases you should try to create arrays with the size you need for your calculation. However, it is possible to delete, insert or append elements. The examples below show you how to do this. Note, you will need to use the numpy 's' function if you wish to delete slices of arrays (see variable c).

In [ ]:

```
import numpy as np
a=np.arange(21)
print(a)
b=np.delete(a,[1,3,5])                # elements 1,3,5 of a are removed
c=np.delete(a,np.s_[1:5])             # elements 1 to 5 of a are removed.
print("b elements 1,3,5 removed",b)
print("c elements 1 to 4 removed",c)
d=np.insert(a,10,[25,27,29])          # insert the numbers 25,27,29 after 10th element
print("d 25,27 and 29 inserted ",d)
e=np.append(a,[24,26,28])             # 24,26 and 28 appended at the end
print("e 24,26,28 appended at the end",e)
```

9.2 mathematical operations on 1D arrays.

9.2.1 Basic functions

Numpy includes all the standard functions that you will find in the math library so you do not need to import the math library separately. However, it also extends these functions to operate smoothly on arrays (rather than simple variables). You do need to make sure that you use the numpy version (ie. np.sin ...) as a call to the corresponding function in the math library using an array will fail (you might test this yourself). Hence it is very easy to plot for example a function once you've defined the _x_ points you wish to use. Look at this example, it automatically creates an array _y_ with elements that correspond to each element in the array _x_. The ability to do these operations on multiply elements in an array without using loops is known as vectorisation.

In [ ]:

```
import numpy as np
from matplotlib import pyplot as plt
x=np.linspace(-10.,10.,100)
y=np.cos(x)
plt.plot(x,y)
plt.show()
```

Other examples of vectorisation are shown in the examples below. You might compare the first with the loop we showed in section 9.1.2.

In [ ]:

```
import numpy as np
x=np.arange(13)
y=x*x                          # generate an array of squares using vectorisation
print(y)
print(" ")
z=x*y                          # now generate an array of cubes from the squares - again usi
print(z)
```

9.2.2 Array functions

Vectors

In addition to the mathematical functions you expect, numpy , also has functions that operate specifically on arrays. Let's take some examples of operations on vectors stored with numpy as 1D vectors of 3 elements. See how it is much easier to do cross-products in python (and less prone to error) rather than doing them by hand. Also note how numpy correctly returns a scalar for the dot product and a vector (as an array) for the cross product.

In [ ]:

```
import numpy as np
a=[2,5,3]
b=[1,7,4]
c=np.dot(a,b)
d=np.cross(a,b)                          # NB. This is only valid for 2D or 3D vectors (There
print("dot product of a and b is", c)
print("cross product of a and b is",d)
```

Statistical

If I have a 1D array of numbers, then it is easy to carry obtain statistical functions, see the examples below.

In [ ]:

```
import numpy as np
npts=10
data=np.empty(npts)
for i in range(npts):
    data[i]=np.random.rand()*100.              # Generate a set of random numbers between 0

meanvalue=np.mean(data)
medianvalue=np.median(data)
standarddev=np.std(data)
variance=np.var(data)
maxval=np.max(data)
minval=np.min(data)
print(data)
print("Mean Value = ", meanvalue)
print("Median value = ", medianvalue)
print("Standard deviation = ",standarddev)
print("variance =",variance)
print("Maximum = ",maxval)
print("Minimum = ", minval)
```

There are other functions that will find the sum of all the elements in the array, the product of all the elements, the index of the maximum and minimum values in the list etc. In short, if there is an operation you wish to carry out on a 1D numpy array there is almost certainly a numpy function that will do the job for you. There is also a useful *np.histogram()* function but we do not have the time to discuss it here. You might look it up if you wish to produce histograms of data.

9.3 Multi-dimensional arrays - 2D arrays.

As we have seen, numpy is very useful for handling 1D arrays of data and the inbuilt functions save a great deal of time and effort compared to writing them for yourself. However, the utility of the numpy library becomes more apparent when we wish to carry out calculations on data in multidimensional arrays. For example, you will see 3 x 3 matrices have an important part to play in many physical problems and basic operations like transposition, inversion and finding determinants of matrices are some of the important operations we may wish

to carry out on them. To do the calculations by hand is tedious and error prone but these are exactly the kind of calculations that are well suited to our computational methods. In the next sections we wil look at basic array manipulations and calculations. Towards the end of the numerical modelling course we will look at more complex problems involving linear algebra that will show the true utility of this library. We will also introduce matrix methods with numpy. If you have not covered matrices in your maths course yet you may omit the relevant section and come back to it later.

9.3.1 Creating 2 dimensional arrays explicitly.

The simplest way for creating 2D arrays may be seen in the following:

In [ ]:

```
import numpy as np
a=np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
b=np.array([[1,2,3,4],[5,6,7,8]])
c=np.array([[1,2],[3,4],[5,6],[7,8]])
print("a = ")
print(a)
print(" ")
print("b = ")
print(b)
print(" ")
print("c = ")
print(c)
print(" ")
d=np.array([[1,2,3],[1,2]])
print("d = ")
print(d)
#
# Note also how we can check the sahpe of our ararys, i.e.
#
print("The shape of a is",np.shape(a))
print("The shape of b is",np.shape(b))
print("The shape of c is",np.shape(c))
print("The shape of d is",np.shape(d))
#
# Note also
#
print("The dimension of a is ", np.ndim(a))     # find the dimension of the array
print("The size of a is ",np.size(a))           # find the size of the array
```

If you look at the syntax carefully you can see that we have created an array of 1D arrays! Note the outer set of brackets for the 2D array as a whole and the inner set for the 1D arrays that make the rows of the 2D array. The 2D array is hence arranged by adding row after row. Note, how the 3 x 3, 4 x 2 and 2 x 4 arrays have been created. Note also that the output for the last array, _d_, cannot be represented as an _n_ by _m_ array.

Note also how we can establish the dimension of the array with the *np.ndim()* function and its size (the total number of elements) with the *np.size()* that we used previously for 1D arrays.

9.3.2 - creating 2D arrays in a programme.

We can create 2D arrays in much the same way, with the same functions that we used for creating 1D arrays. Note the slightly different syntax and make sure you include double brackets (i.e. the brackets for the function and the brackes for a tuple. You will mpost likely use *np.eye* when doing matrix operations as it is a quick way

of creating the indentity matrix.

In [ ]:

```
a=np.ones((4,4))
print("a = ")
print(a)
b=np.zeros((3,4))
print("b = ")
print(b)
c=np.full((3,3),10)
print("c = ")
print(c)
d=np.eye(6)
print("d = ")
print(d)
```

If we wish, we can also start with 1D arrays and change their 'shape' to a 2 or higher dimensional array later. For example, look at the code below where we start ith a 9 element 1D array that we reshape into a 3x3 array.

In [ ]:

```
import numpy as np
a=np.arange(9)
print(a)
print(np.shape(a))
a=np.reshape(a,(3,3))
print(a)
print(np.shape(a))
```

We shall not use them on this course, but for completeness, we can define higher dimensional arrays by an extension of the methods we have used for 1D and 2D. Look at the code below to see some examples.

In [ ]:

```
import numpy as np
a=np.array([[[1,2],[3,4]],[[5,6],[7,8]]])
print(" a is")                  # create a 3D array
print(a)
b=np.ones((3,3,3))              # create a 3D array of ones.
print("b is")
print(b)
c=np.arange(27)
c=np.reshape(c,(3,3,3))         # create a 3D array
print("c is")
print(c)
print(" ")
d=np.zeros((2,2,2,2))
print(" d is a 4D array - very hard to express in 2 dimensions on paper!")
print(d)
```

9.3.3 Indexing a 2D array

In the same way that we can index a 1D array, we can also do the same for a 2D array as long as we know the syntax. The important point to remember is the index of our rows and columns starts at zero (as in the case of a 1D array). Hence, for a 3 x 3 array the row index will be 0,1, or 2 and similarly the column index will be 0,1 or 2.

To index an element in the array is then straight forward using the syntax [*row index* , *column index*]. Look at the array, _a_ , created below and make sure that you understand how the elements in the array have been indexed.

[This also extends to e.g 3D arrays as, for example, we would have [*row index* , *column index* , *sheet index*]]

In [ ]:

```
import numpy as np
a=np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(a)
print(a[0,2])
print(a[1,0])
print(a[2,1])
```

As we have also seen in previous notebooks the ability to refer to individual elements in 2D arrays by their indices is also important programmatically. In the code block below you can see how, with a nested loop, you can cycle through the individual elements in an array. You should also look carefully at the order in which the elements of the array are printed (i.e. inner loop first, then outer loop) and how it changes when for example, I swap the indices around (i.e. from a[i,j] to a [j,i].)

In [ ]:

```
import numpy as np
a=np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
for i in range(3):
    for j in range(3):
        print(a[i,j])                # comment and uncomment these print statements to see
#        print(a[j,i])               # 3 x 3 array are indexed.
```

9.3.4 Mathematical operations on 2D arrays

At this point it is important to realise that the 'standard' python mathematical operations on arrays are not the same as the matrix operations we will see later. So, when the arrays have the same shape, normal operations (+,-,*,/) are done element wise (i.e. $a_i * a_i$) and not by matrix methods (vectorisation again) . This is why we can do / operations (that are not allowed with matrix algebra). It is also faster, in python, than writing explicit loops. To understand these operations look at the effect of these operators on the 1D row and column arrays below. The results may appear strange at first but make sure you understand the results and the difference between a 1D array as a row and a 1D array as a column.

In [ ]:

```
import numpy as np
a=np.array([1,2,3])                  # a is a row vector
b=np.array([[2],[4],[6]])            # b is a column vector
c=np.array([2,4,6])                  # c is a row vector

print("a =")
print(a)
print("b = ")
print(b)
print("a*b = ")
print(a*b)
print("a*c = ")
print(a*c)
print("a/c =")
print(a/c)
print("a/b =")
print(a/b)
```

then look at the the 2D arrays below, again make sure that you understand how the elements in the arrays have been multiplied and divided.

In [ ]:

```
a=np.array([[2,4,6],[1,3,5],[3,6,9]])
b=np.array([[4,8,12],[1,3,5],[9,18,27]])
print("a = ")
print(a)
print("b= ")
print(b)
print("a * b =")
print(a*b)
print("a / b =")
print(a/b)
```

The only way to really see how this work is to try this out for arrays of different shapes and sizes and look at the results for yourself.

9.3.5 Matrix and linear algebra operations using numpy arrays.

In numpy, we need to take care when we wish to carry out explict matrix and linear algebra calculations. There is a sub package in numpy called the *linalg* package that you should always use the *linalg* functions in preference to similarly named functions in numpy itself (i.e. *np.linalg.inv()* rather than *np.invert()*) . Look at the following example:

In [ ]:

```
import numpy as np
a=np.array([[41,8,12],[1,16,5],[9,18,27]])
b=np.invert(a)
c=np.linalg.inv(a)
print(b)
print(c)
```

Clearly, the *np.invert()* function gives a completely different result to *np.linalg.inv()* ! Unfortunately, it is very easy

to misremember this and strange results will occur if you think this is a matrix inversion. You may wish to look up what this function is!

In [ ]:

```
import numpy as np
a=np.array([[1., .2, 3.], [4., 5., 6.], [7., 8., 9.]])
print("a = ")
print(a)
print("Transpose of a = ")
print(np.transpose(a))
b=np.array([[4.,3.,2.],[6.,4.,2.],[9.,5.,1.]])
print("b = ")
print(b)
print ("matrix multiply a and b")
print(np.matmul(a,b))
print("The determinant of a is")
print(np.linalg.det(a))
```

In [ ]:

```
import numpy as np
a=np.array([[4, 2, 3], [7, 5, 3], [1, 8, 6]])
b=np.array([[2],[3],[5]])
print("a = ")
print(a)
print("b = ")
print(b)
print("matrix a x matrix b =")
print(np.matmul(a,b))
c=np.linalg.inv(a)
print("The inverse matrix of a is c =")
print(c)
print("Let's check by multiplying a x c")
d=np.matmul(a,c)
print(d)
```

9.4 Summary

In this notebook you have learnt,

- how to create and maniuplate 1D numpy arrays by direct use of the array index and by vectorisation,
- how to slice, dice and append 1D arrays using numpy,
- how to use 3 element 1D numpy arrays as vectors and to generate dot and cross products,
- how to do statistical operations on 1D arrays, i.e. mean, standard deviation etc.,
- how to create 2D (and higher dimensional arrays,
- how to index higher dimensional arrays,
- about the extension of vectorisation to higher dimensional arrays.
- how numpy may be used to do basic matrix calculations.