SUMMER SCHOOL 2021

Numpy Tutorial

Andrew Gill

Summary of commands used in this lecture

Command	Meaning
array	Create a numpy array from a list
arange	Similar to range, but works with floating point numbers. Returns a numpy array.
linspace	Similar to arange, but the third parameter specifies number of points rather than step size. Endpoint is included
ones	creates an array specified dimensions with all elements 1
zeros	creates an array specified dimensions with all elements 0
eye	Creates an identity matrix
linalg.det	Calculate determinant of a square matrix
linalg.inv	Calculate inverse matrix of a square matrix
dot or @ operator	Calculate dot product (AKA inner product) of two numpy arrays - matrix multiplication
reshape	Changes the shape of a numpy array. The number of elements must not change.
sum	calculates the sum of an array, or each row or column of the array
mean	calculate the mean (average) of the entire array, or by rows or columns
polyfit	Do a polinomial curve-fit of specified order on x-y data
polyval	Calculate the value of a polinomial for a data range x
random.random	Calculates an array of random floating point numbers
random.randint	Calculates an array of random integers
readtxt	Read in data from a tabular text file, optionally ignoring headers and specifying the column delimiter
savetxt	Write out an array to a tabular text file, optionally specifying headers and number format, specifying the column delimiter

Why do we use Numpy?
Ordinary Python can do quite a bit “out of the box”.

The list data structure is very powerful, and much more flexible than a C or Fortran array.

Also, the built-in math libraries contain all the common mathematical functions, even for complex numbers (cmath). Finally, we have the ability to read in the contents of a data file with a for-loop.

So what more could we want?

The answer is that Numpy is built to make it easier to handle numerical data and do calculations for scientific applications.

The core of numpy is the numpy array. This behaves a little like a list, but adds a huge amount of functionality which wouldn’t make sense in the general-purpose list structure, but which is invaluable if you’re working with large amounts of numerical data.

Coupled with this, numpy allows efficient matrix arithmetic and includes a lot of efficiently implemented linear algebra and matrix manipulation functions to use with the Numpy array – inverting matrices, finding eigen values or just transposing a matrix. It even includes some powerful one-line functions for reading and writing data from or to formatted text files that spreadsheets like, or even some binary formats.

NB: Convention for These Notes

Grey-outlined black text boxes are scripts or pieces of code that can be copied whole and executed (either by copying and pasting into a text editor and save and run as a python file (the better way), or just by pasting directly into an interactive python session (ugly, but it should work):

```
print("hello there")
```

Green outlined black text boxes represent code entered into a python interactive session (which is always in **bold** - copy or type *only* these parts) together with the output of the python interpreter (which you should not copy.

Also, don't copy the interpreter prompt ">>>". *It will cause errors!*

```
>>> print("hello")
hello
```

Also remember: the interactive mode is only for quick throwaway tasks and testing - anything more than 3 lines should probably go in a script.

How do We Use Numpy?

To get access to all of this, we import numpy as for any python module:

```
import numpy as np
```

You've already used the range() function a lot, and its very useful, but it only works with integers.

However, in numpy there's a function called arange (array range) which can take arbitrary floating-point increments:

```
>>> x = np.arange(-0.1,0.5,0.007)
>>> x
array([-0.1  , -0.093, -0.086, -0.079, -0.072, -0.065, -0.058, -0.051,
       -0.044, -0.037, -0.03 , -0.023, -0.016, -0.009, -0.002,  0.005,
        0.012,  0.019,  0.026,  0.033,  0.04 ,  0.047,  0.054,  0.061,
        0.068,  0.075,  0.082,  0.089,  0.096,  0.103,  0.11 ,  0.117,
        0.124,  0.131,  0.138,  0.145,  0.152,  0.159,  0.166,  0.173,
        0.18 ,  0.187,  0.194,  0.201,  0.208,  0.215,  0.222,  0.229,
        0.236,  0.243,  0.25 ,  0.257,  0.264,  0.271,  0.278,  0.285,
        0.292,  0.299,  0.306,  0.313,  0.32 ,  0.327,  0.334,  0.341,
        0.348,  0.355,  0.362,  0.369,  0.376,  0.383,  0.39 ,  0.397,
        0.404,  0.411,  0.418,  0.425,  0.432,  0.439,  0.446,  0.453,
        0.46 ,  0.467,  0.474,  0.481,  0.488,  0.495])
```

There is a similar function called linspace, which instead of using a step size as a parameter, instead lets you specify how many steps you want to use:

```
>>> x = np.linspace(0,10.0,30)
>>> x
array([  0.        ,   0.34482759,   0.68965517,   1.03448276,
         1.37931034,   1.72413793,   2.06896552,   2.4137931 ,
         2.75862069,   3.10344828,   3.44827586,   3.79310345,
         4.13793103,   4.48275862,   4.82758621,   5.17241379,
         5.51724138,   5.86206897,   6.20689655,   6.55172414,
         6.89655172,   7.24137931,   7.5862069 ,   7.93103448,
         8.27586207,   8.62068966,   8.96551724,   9.31034483,
         9.65517241,  10.        ])
```

Note that this includes both the starting-point (0) and the endpoint(100), unlike range and arange.

If you're a matlab or octave user, this may seem familiar to you. That's deliberate, as much of numpy is written with the intention of making a transition from matlab as painless as possible.

So np.eye, np.ones and np.zeros will behave mostly as you expect (if you don't know matlab, a quick perusal of the docs will tell you what these do, if you can't guess from the names).

The good news is that numpy arrays are mostly compatible with lists, and conversion between the two is easy:

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

You can do a 2-d array by giving it a list of lists, like this:

```
>>> b = [[1,2,3],[4,5,6],[7,8,-1]]
>>> bb=np.array(b)
>>> bb
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8, -1]])
```

Now, if you want to find the inverse matrix of bb, first check if its possible:

np.linalg.det(bb)

```
>>> np.linalg.det(bb)
29.999999999999989
```

This tells us that the determinant is a positive number – so we can invert it

np.linalg.inv(bb)

```
>>> np.linalg.inv(bb)
array([[-1.76666667,  0.86666667, -0.1       ],
       [ 1.53333333, -0.73333333,  0.2       ],
       [-0.1       ,  0.2       , -0.1       ]])
```

Notice that the det (determinant) and inv (inverse matrix) are part of Numpy.linalg, a sub-module within numpy dealing with advanced linear algebra functions.

Some of the more simple matrix operations like matrix multiplication (dot product) are in base numpy, for instance using the array bb above:

np.dot(bb,np.linalg.inv(bb))

```
>>> np.dot(bb,np.linalg.inv(bb))
array([[  1.00000000e+00,   1.11022302e-16,  -5.55111512e-17],
       [ -6.66133815e-16,   1.00000000e+00,  -1.11022302e-16],
       [  4.16333634e-16,   1.38777878e-16,   1.00000000e+00]])
```

Which will give you an identity matrix, with ones on the diagonal and zeros (or very small numbers – computers make rounding errors) everywhere else.

A shorthand for doing matrix multiplication is the "@" symbol:

```
>>> bb@np.linalg.inv(bb)
array([[  1.00000000e+00,   1.11022302e-16,  -5.55111512e-17],
       [ -6.66133815e-16,   1.00000000e+00,  -1.11022302e-16],
       [  4.16333634e-16,   1.38777878e-16,   1.00000000e+00]])
```

Be careful, though! Numpy.invert doesn't do the same thing as numpy.linalg.inv . "invert: does a bitwise inversion, flipping ones to zeros and vice versa in each number in the array.

Numpy arrays can be added together or subtracted just as you'd expect:

```
>>> p=np.array([1,2,3])
>>> q=np.array([-3,-4,-5])
>>> r=p+q
>>> s=p-q
>>> r
array([-2, -2, -2])
>>> s
array([4, 6, 8])
```

For this to work, however, all arrays have to be the same size and shape.

You can check the size (number of elements) in a numpy array like this:

```
p.size
3
```

The shape (number of rows and columns) is a list or tuple containing the number of elements along each axis of the array (eg. the number of rows and columns in a two-dimensional array)

```
>>> p.shape
(3,)
```

and number of dimensions (1D is a row, 2D has rows and columns, 3D has rows,columns and depth, and so on)

```
>>> p.ndim
1
```

Note that Numpy sometimes distinguishes between a 1-dimensional array and a two-dimensional array with only one column or row. This can cause problems, but can easily be fixed by manipulating the array shape.

The numpy function reshape allows you to rearrange the array so that the elements are arranged in however many rows and columns as you want, as long as the rows multiplied by the columns stays equal to the total number of elements in the array.

So to convert p from a 1d to a 2d array, we do this:

```
>>> p2d=np.reshape(p,[1,3])
>>> p2d
array([[1, 2, 3]])
```

The list parameter after p is a list that tells reshape how many rows and columns we want. In this case there is only one row, but three columns. Use ndim and shape to confirm the shape of p2d.

A last point about numpy array shapes: a 2D array can be transposed (rows and columns flipped) by putting a ".T" after the variable name, for instance the array p2d in the example would be transposed as p2d.T

Returning to the subject of numpy arithmetic, “*” and “/” do elementwise multiplication and division (again, only if the shape of the matrices are appropriate – they must have the same number of columns at least).

For example:

```
>>> f=np.array([1,2,3])
>>> g=np.array([2,2,-3])
>>> f*g
array([ 2,  4, -9])
```

Compare this with the non-mathematical way in which “+”, “-” and “*” operate on python lists.

Make sure you don’t get confused between the two!

Numpy has its own maths functions (sin, cos, tan, sqrt and so on) that operate on each element of an array.

Here’s a quick demonstration script, just to show you how powerful all this is:

```
import numpy as np
x = np.arange(0,10.1,0.1)
y1=x*x
y2=x**2*np.sin(x) # note that ** raises each element to a power,
in this case 2.
#just for fun, lets plot y1 and y2 relative to x using matplotlib
(we’ll do more on that later)
import matplotlib.pyplot as plt
plt.plot(x,y1,"r*")
plt.plot(x,y2,"g")
plt.show()
```

In python, you’d either have had to use a for-loop or the map() function to do that. Either way would have required a few more lines.

Note: We'll deal with plotting using matplotlib in the next section. However, it's not difficult to understand what it does here, and having a graph to look at makes this exercise a bit more interesting.

If you want to do a polynomial curve-fit to x-y data, you use polyfit.

You can then generate a data series to plot or otherwise use by means of polyval

```
import numpy as np
x = np.arange(0,11)
y=x**2+3*x*np.random.rand(x.size)-1
p = np.polyfitx,y,2)
xfit=np.arange(0,10.01,0.01)
yfit=np.polyval(p,xfit)
import matplotlib.pyplot as plt
plt.plot(x,y,"g*")
plt.plot(xfit,yfit,"k")
plt.show()
```

Note that for this example, we used the function rand from the Numpy submodule random to introduce some scattter to a polynomial. rand Generates a numpy array of random numbers. There are other functions such as ranf() or randint() which generate floating point numbers or or integers respectively, and can be given upper and lower bounds, etc. uniform() Will approximate a uniform random distribution.

Numpy also has functions to calculate the sum, the mean and other statistical quantities for an arbitrary array.

One of the optional parameters to the mean and sum functions is “axis”. This allows you to average each column (axis=0) or row (axis=1) and so on for higher dimensional numpy arrays.

```
>>> a=np.array([[1,2,3],[4,5,6],[7,8,9]])
>>> np.mean(a)
5.0
>>> np.mean(a,axis=0)
array([ 4.,  5.,  6.])
>>> np.mean(a,axis=1)
array([ 2.,  5.,  8.])
```

There are many more interesting and useful tricks that you can do with numpy arrays; here is one that is very powerful but not obvious, which is comparing values elementwise:

```
>>> k=np.array([4,3,5])
>>> m=np.array([2,3,6])
>>> k>m
array([ True, False, False], dtype=bool)
>>> k<=m
array([False,  True,  True], dtype=bool)
>>> k==m
array([False,  True, False], dtype=bool)
```

You will see that the above returns an array of boolean values, which is the same size as k and m.

Each element from the two arrays is compared according to the inequality (or equality) operator, and a True or False value is put in the corresponding place in the boolean array.

This can be very useful, especially when you recall that in python, a Boolean value of True is really just 1, and False is 0.

Also, if you use an array of booleans as the “index” to an array of identical shape, it will return only those values that are at positions corresponding with a value of True in the “index” boolean array.

Now something slightly more mundane but a lot more generally useful.

Suppose you have an ascii (text) file containing a table of numbers – it might have headings or not, and there there might be spaces, tabs or commas between the columns.

Here's an example, which you can copy into a text-editor and save as "mydata.csv":

```
"Time(s)","Voltage(V)","Temperature (K)","Wind Speed(km/h)"
0,22.1,300.05,5.2
1,23.0,305.22,5.1
3,22.9,310.01,5.2
4,24.1,315.52,5.3
```

In order to use this data, you want to read it into a numpy array. You could go and open the file, read it line-by-line and split each line according to the delimiter (column separator), then convert it into a number and put it in an appropriate place in a list. But Numpy makes it easy to do this with a single function, called loadtxt:

```
>>> data1=np.loadtxt("mydata.csv",delimiter=",",skiprows=1)
>>> data1
array([[   0.  ,   22.1 ,  300.05,    5.2 ],
       [   1.  ,   23.  ,  305.22,    5.1 ],
       [   3.  ,   22.9 ,  310.01,    5.2 ],
       [   4.  ,   24.1 ,  315.52,    5.3 ]])
```

As you might expect, loadtxt has a friend called savetxt, which does the exact reverse – it writes a numpy array to a file, according to the format which you specified. Let's write a script to open the previous datafile, square each element, and write them out to a new file, with tabs instead of comma separators, a new format and a new header:

```
import numpy as np
data1=np.loadtxt("mydata.csv",delimiter=",",skiprows=1)
np.savetxt("outdata.txt",data1**2,fmt="%.6f",delimiter="\t",header="written by numpy")
```

If you run the above script and look at the files in your working directory, you should find the file "outdata.txt", which should contain this information:

```
# written by numpy
0.000000      488.410000    90030.002500  27.040000
1.000000      529.000000    93159.248400  26.010000
9.000000      524.410000    96106.200100  27.040000
16.000000     580.810000    99552.870400  28.090000
```

If your data file is “dirty” - that is, it has gaps or the number of columns changes from row to row, or a few bits of random text where numbers should appear, or other such nasty things, you should look at the genfromtxt function. It provides more ways of dealing with such data files. It works a lot like loadtxt, but gives you much more optional arguments to control its behaviour.

If even that isn’t enough, you need to consider using the pandas module, which you’ll be introduced to towards the end of this course.

Going further

There are many other things which Numpy can do, and not enough time in this course to cover all of them.

If you need to figure out how to do a particular task, have a look at the documentation and tutorial on the official numpy website:

https://numpy.org/doc/stable/

Also, for still more specialised mathematical functions used in scientific applications you should have a look at the scipy library, which I'll give a brief introduction to later in this course:

https://www.scipy.org/

Scipy is built on top of numpy, and designed to work well with it.