In [1]:

```
%matplotlib inline
import matplotlib.pyplot as plt
```

PH2150- Scientific Computing Skills

Numpy - multidimensional data arrays

The `numpy` package is used in almost all numerical computation using *Python*. If you are looking at a *Python* textbook and it does not contain a section on *NumPy* then it is probably the wrong book for you. *NumPy* is a package that provides high-performance vector, matrix and higher-dimensional data structures for *Python*. It is implemented in *C* and *Fortran* so when calculations are vectorized (formulated with vectors and matrices), performance can be very fast.

In the `numpy` package the terminology used for vectors, matrices and higher-dimensional data sets is an *array*. And the object that is created is the numpy.ndarray (N-dimensional array).

These notes contain a brief introduction to some of the properties and usage of the `numpy` array, for a more complete description:

https://docs.scipy.org/doc/numpy/index.html (https://docs.scipy.org/doc/numpy/index.html)

To use `numpy` you need to import the module, using for example:

In [19]:

```
import numpy as np # here we use the alias np which is common practice
```

Creating `numpy` arrays

There are a number of ways to create new numpy arrays, for example from:

- a *Python* list or tuples
- using functions to generate numpy arrays, such as `np.arange` , `np.linspace` .
- reading data from files, `np.loadtxt` see Problem sheet 3.

From lists

For example, to create new vector and matrix arrays from Python lists we can use the `numpy.array` function.

In [20]:

```
# a vector: the argument to the array function is a Python list
v = np.array([1,2,3,4])
v
```

Out[20]:

```
array([1, 2, 3, 4])
```

In [21]:

```
# a matrix: the argument to the array function is a nested Python list
M = np.array([[1, 2], [3, 4]])
M
```

Out[21]:

```
array([[1, 2],
       [3, 4]])
```

The `v` and `M` objects are both of the type `ndarray` that the `numpy` module provides.

In [22]:

```
type(v), type(M)
```

Out[22]:

```
(numpy.ndarray, numpy.ndarray)
```

The difference between the `v` and `M` arrays is only their shapes. We can get information about the shape of an array by using the `ndarray.shape` property.

In [23]:

```
print ('vshape=',v.shape,'\nMshape=', M.shape) #\n gives new line
```

```
vshape= (4,)
Mshape= (2, 2)
```

The number of elements in the array is available through the `ndarray.size` property:

In [24]:

```
print ('v_size=',v.size,'\nM_size=', M.size) #\n gives new line
```

```
v_size= 4
M_size= 4
```

Equivalently, we could use the function `numpy.shape` and `numpy.size`

In [26]:

```
print('Shape of M =',np.shape(M),'\nSize of M =',np.size(M))
```

```
Shape of M = (2, 2)
Size of M = 4
```

Using the `dtype` (data type) property of an `ndarray`, we can see what type the data of an array has:

In [27]:

```
M.dtype,v.dtype
```

Out[27]:

```
(dtype('int32'), dtype('int32'))
```

We can explicitly define the type of the array data when we create it, using the `dtype` keyword argument:

In [30]:

```
M = np.array([[1, 2], [3, 4]], dtype=complex)

print(M)

M.dtype
```

```
[[1.+0.j 2.+0.j]
 [3.+0.j 4.+0.j]]
```

Out[30]:

```
dtype('complex128')
```

The `numpy.ndarray` has the following properties:

- The number of elements in an array is fixed.
- You cannot add elements to an array once it is created, or remove them.
- The elements of an array must all be of the same type, such as all floats or all integers.
- You cannot mix elements of different types in the same array and you cannot change the type of the elements once an array is created.

This is much more restrictive than the *list* data container that we have already seen, why would we ever use an array if lists are more flexible?

- Lists do not support mathematical functions such as matrix and dot multiplications, etc.
- Implementing such functions for Python lists would not be very efficient because of the dynamic typing (the data type at each element can be changed).
- The restriction that Numpy arrays are **statically typed** (fixed when created) and **homogeneous** (all the same data type) allows them to be more memory efficient.
- The static typing, also allows fast implementation of mathematical functions such as multiplication and addition of `numpy` arrays, which can be implemented in a compiled language (C and Fortran is used).

Common data types that can be used with `dtype` are: `int`, `float`, `complex`, `bool`, `object`, etc.

We can also explicitly define the bit size of the data types, for example: `int64`, `int16`, `float128`, `complex128`.

Using array-generating functions

For larger arrays it is inpractical to initialize the data manually, using explicit python lists. Instead we can use one of the many functions in `numpy` that generate arrays of different forms. Some of the more common are:

arange

In [14]:

```
# create a range

x = np.arange(0, 10, 1) # arguments: start, stop, step

x
```

Out[14]:

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [15]:

```
x = np.arange(-1, 1, 0.1) # arguments: start, stop, step

x
```

Out[15]:

```
array([ -1.00000000e+00,  -9.00000000e-01,  -8.00000000e-01,
        -7.00000000e-01,  -6.00000000e-01,  -5.00000000e-01,
        -4.00000000e-01,  -3.00000000e-01,  -2.00000000e-01,
        -1.00000000e-01,  -2.22044605e-16,   1.00000000e-01,
         2.00000000e-01,   3.00000000e-01,   4.00000000e-01,
         5.00000000e-01,   6.00000000e-01,   7.00000000e-01,
         8.00000000e-01,   9.00000000e-01])
```

linspace and logspace

In [16]:

```
# using linspace, both end points ARE included
np.linspace(0, 10, 25) # (start, stop, num=50, endpoint=True, retstep=False, dtype=Non
e)
```

Out[16]:

```
array([  0.        ,   0.41666667,   0.83333333,   1.25      ,
         1.66666667,   2.08333333,   2.5       ,   2.91666667,
         3.33333333,   3.75      ,   4.16666667,   4.58333333,
         5.        ,   5.41666667,   5.83333333,   6.25      ,
         6.66666667,   7.08333333,   7.5       ,   7.91666667,
         8.33333333,   8.75      ,   9.16666667,   9.58333333,  10.
])
```

In [17]:

```
logspace(0, 10, 10, base=e) # (start, stop, num=50, endpoint=True, base=10.0, dtype=None)
```

Out[17]:

```
array([  1.00000000e+00,   3.03773178e+00,   9.22781435e+00,
         2.80316249e+01,   8.51525577e+01,   2.58670631e+02,
         7.85771994e+02,   2.38696456e+03,   7.25095809e+03,
         2.20264658e+04])
```

mgrid

In [35]:

```
x, y = np.mgrid[0:5, 0:5] # similar to meshgrid in MATLAB

print ('x=\n',x)

print('\ny=\n',y)
```

```
x=
 [[0 0 0 0 0]
 [1 1 1 1 1]
 [2 2 2 2 2]
 [3 3 3 3 3]
 [4 4 4 4 4]]

y=
 [[0 1 2 3 4]
 [0 1 2 3 4]
 [0 1 2 3 4]
 [0 1 2 3 4]
 [0 1 2 3 4]]
```

random data

Numpy contains a sub package `random` that contains many function for generating arrays of random numbers, that we will return to later in the course, below are a few examples.

In [21]:

```
from numpy import random
```

In [22]:

```
# uniform random numbers in [0,1]
random.rand(5,5)
```

Out[22]:

```
array([[ 0.92932506,  0.19684255,  0.736434  ,  0.18125714,  0.70905038],
       [ 0.18803573,  0.9312815 ,  0.1284532 ,  0.38138008,  0.36646481],
       [ 0.53700462,  0.02361381,  0.97760688,  0.73296701,  0.23042324],
       [ 0.9024635 ,  0.20860922,  0.67729644,  0.68386687,  0.49385729],
       [ 0.95876515,  0.29341553,  0.37520629,  0.29194432,  0.64102804]])
```

In [23]:

```
# standard normal distributed random numbers
random.randn(5,5)
```

Out[23]:

```
array([[ 0.117907  , -1.57016164,  0.78256246,  1.45386709,  0.54744436],
       [ 2.30356897, -0.28352021, -0.9087325 ,  1.2285279 , -1.00760167],
       [ 0.72216801,  0.77507299, -0.37793178, -0.31852241,  0.84493629],
       [-0.10682252,  1.15930142, -0.47291444, -0.69496967, -0.58912034],
       [ 0.34513487, -0.92389516, -0.216978  ,  0.42153272,  0.86650101]])
```

zeros and ones

Sometimes you just want to start with an array filled with either just zeros or ones. With a particular shape. As this is quite a common requirement functions are built in to do this.

In [26]:

```
np.zeros((3,3))
```

Out[26]:

```
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
```

In [27]:

```
np.ones((3,3))
```

Out[27]:

```
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
```

Manipulating arrays

Indexing

We can index elements in an array using square brackets and indices:

In [39]:

```
# a matrix: the argument to the array function is a nested Python list
M = np.array([[1, 2], [3, 4]])
M
```

Out[39]:

```
array([[1, 2],
       [3, 4]])
```

In [40]:

```
# v is a vector, and has only one dimension, taking one index
v[0]
```

Out[40]:

```
1
```

In [41]:

```
# M is a matrix, or a 2 dimensional array, taking two indices
M[1,1]
```

Out[41]:

```
4
```

If we omit an index of a multidimensional array it returns the whole row (or, in general, a N-1 dimensional array)

In [42]:

```
M
```

Out[42]:

```
array([[1, 2],
       [3, 4]])
```

In [43]:

```
M[1]
```

Out[43]:

```
array([3, 4])
```

The same thing can be achieved with using : instead of an index:

In [45]:

```
M[1,:] # row 1
```

Out[45]:

```
array([ 0.60410063,  0.4791374 ,  0.8237106 ])
```

In [46]:

```
M[:,1] # column 1
```

Out[46]:

```
array([ 0.40043577,  0.4791374 ,  0.15459644])
```

We can assign new values to elements in an array using indexing:

In [47]:

```
M[0,0] = 1
```

In [48]:

```
M
```

Out[48]:

```
array([[ 1.        ,  0.40043577,  0.66254019],
       [ 0.60410063,  0.4791374 ,  0.8237106 ],
       [ 0.96856318,  0.15459644,  0.96082399]])
```

In [49]:

```
# also works for rows and columns
M[1,:] = 0
M[:,2] = -1
```

In [50]:

```
M
```

Out[50]:

```
array([[ 1.        ,  0.40043577, -1.        ],
       [ 0.        ,  0.        , -1.        ],
       [ 0.96856318,  0.15459644, -1.        ]])
```

Index slicing

Index slicing is the technical name for the syntax `M[lower:upper:step]` to extract part of an array:

In [51]:

```
A = array([1,2,3,4,5])
A
```

Out[51]:

```
array([1, 2, 3, 4, 5])
```

In [52]:

```
A[1:3]
```

Out[52]:

```
array([2, 3])
```

Array slices are *mutable*: if they are assigned a new value the original array from which the slice was extracted is modified:

In [53]:

```
A[1:3] = [-2,-3]

A
```

Out[53]:

```
array([ 1, -2, -3,  4,  5])
```

We can omit any of the three parameters in `M[lower:upper:step]` :

In [54]:

```
A[::] # lower, upper, step all take the default values
```

Out[54]:

```
array([ 1, -2, -3,  4,  5])
```

In [55]:

```
A[::2] # step is 2, lower and upper defaults to the beginning and end of the array
```

Out[55]:

```
array([ 1, -3,  5])
```

In [56]:

```
A[:3] # first three elements
```

Out[56]:

```
array([ 1, -2, -3])
```

In [57]:

```
A[3:] # elements from index 3
```

Out[57]:

```
array([4, 5])
```

Negative indices counts from the end of the array (positive index from the begining):

In [58]:

```
A = array([1,2,3,4,5])
```

In [59]:

```
A[-1] # the last element in the array
```

Out[59]:

```
5
```

In [60]:

```
A[-3:] # the last three elements
```

Out[60]:

```
array([3, 4, 5])
```

Index slicing works exactly the same way for multidimensional arrays:

In [61]:

```
A = array([[n+m*10 for n in range(5)] for m in range(5)])

A
```

Out[61]:

```
array([[ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14],
       [20, 21, 22, 23, 24],
       [30, 31, 32, 33, 34],
       [40, 41, 42, 43, 44]])
```

In [62]:

```
# a block from the original array
A[1:4, 1:4]
```

Out[62]:

```
array([[11, 12, 13],
       [21, 22, 23],
       [31, 32, 33]])
```

In [63]:

```
# strides
A[::2, ::2]
```

Out[63]:

```
array([[ 0,  2,  4],
       [20, 22, 24],
       [40, 42, 44]])
```

Further reading

- http://numpy.scipy.org (http://numpy.scipy.org)
- http://scipy.org/Tentative_NumPy_Tutorial (http://scipy.org/Tentative_NumPy_Tutorial)
- http://scipy.org/NumPy_for_Matlab_Users (http://scipy.org/NumPy_for_Matlab_Users) - A Numpy guide for MATLAB users.