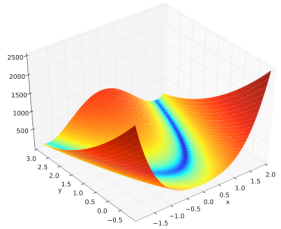
NumPy / SciPy / Matplotlib

NumPy is an extension to Python adding support for arrays and matrices, along with a large library of high-level mathematical functions to operate on them.
SciPy is a library of algorithms and tools for common tasks in science and engineering. It includes modules for optimization, interpolation, linear algebra, integration, FFT, etc.
Matplotlib is a plotting library for producing high-quality graphs and figures from your data.



Why NumPy / SciPy?

Well-written algorithms for commonly-used mathematical tools: don't reinvent the wheel!
Many of these algorithms need careful thought to implement safely: don't reinvent the wheel!
NumPy / SciPy is *fast*: the underlying algorithms are written in C (accessed through Python *interfaces*)

Why not MATLAB / Mathematica?

Python / NumPy / SciPy is more powerful
Benefits to integration with a widely-used, well-supported *general-purpose* programming language
Python / NumPy / SciPy is **FREE**

A Tour of NumPy

NumPy `arrays` are more efficient than Python's `lists` for numerical operations ...
... but they can hold only numbers (or boolean values) and there are no `append()` or `extend()` methods
NumPy has lots of helpful methods for dealing with multidimensional arrays (e.g. for calculating the transpose or inverse of a matrix
Mathematical operations can be applied in a clean way to the whole array (or a slice of it) at once. For example,

```
>>> import numpy as np
>>> a = np.arange(5)
array([0, 1, 2, 3, 4])

>>> np.exp(a)
array([  1.        ,   2.71828183,
7.3890561 ,  20.08553692,  54.59815003])
```

Initializing a NumPy array

In the following it is assumed that numpy has been imported as `np`:

```
>>> import numpy as np

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

>>> b = np.linspace(-2, 2, 5)
array([-2., -1.,  0.,  1.,  2.])

>>> c = np.array([[11,12],[21,22], [31,32]])
array([[11, 12],
       [21, 22],
       [31, 32]])
```

Initializing a NumPy array

There are also convenience methods for initializing commonly-used arrays:

```
>>> I = np.eye(2,2)
array([[ 1.,  0.],
       [ 0.,  1.]])
```

the identity matrix

```
>>> z = np.zeros((2, 3, 2))
array([[[ 0.,  0.],
        [ 0.,  0.],
        [ 0.,  0.]],

       [[ 0.,  0.],
        [ 0.,  0.],
        [ 0.,  0.]]])
```

a three-dimensional 2×3×2 array of zeros

Useful NumPy array methods

There are also convenience methods for initializing commonly-used arrays:

```
>>> z.dtype
dtype('float64')
```

ie double precision

```
>>> z.shape
(2, 3, 2)
>>> z.reshape(3,4)
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]])
>>> z.flatten()
array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,
0.,  0.,  0.,  0.,  0.])
```

There is also a method, `resize()`, which is like `reshape()` but changes the array it acts on in place instead of returning a new array

Useful NumPy array methods

```
>>> c
array([[11, 12],
       [21, 22],
       [31, 32]])
>>> c.transpose()
array([[11, 21, 31],
       [12, 22, 32]])
>>> a = np.array([1., 4., 1.])
>>> b = np.array([2., 0., 2.5])
>>> a + b
array([ 3. ,  4. ,  3.5])
>>> a * b
array([ 2. ,  0. ,  2.5])
>>> a.dot(b)
4.5
>>> np.mean(a)
2.0
```

or just `c.T`

or `np.dot(a, b)`; the scalar product

Useful NumPy array methods

```
>>> a = np.array([1., 4., 1.])
>>> b = np.array([2., 0., 2.5])
>>> np.max(a)
4.0
>>> a > b
array([False,  True, False], dtype=bool)
>>> b > 0.1
array([ True, False,  True], dtype=bool)
```

even:

```
>>> (a > b) | (b > 0.1)
array([ True,  True,  True], dtype=bool
```

Useful NumPy array methods

```
>>> c
array([[11, 12],
       [21, 22],
       [31, 32]])
>>> np.ones_like(c)
array([[1, 1],
       [1, 1],
       [1, 1]])
```

also, `zeros_like()`

... and many, many more: see the NumPy documentation:
http://docs.scipy.org/doc/numpy/reference/

Indexing and Slicing Arrays

index numpy arrays as, e.g. `c[i, j]` not `c[i][j]`

```
>>> h = np.arange(48).reshape(3, 4, 4)
>>> h
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11],
        [12, 13, 14, 15]],

       [[16, 17, 18, 19],
        [20, 21, 22, 23],
        [24, 25, 26, 27],
        [28, 29, 30, 31]],

       [[32, 33, 34, 35],
        [36, 37, 38, 39],
        [40, 41, 42, 43],
        [44, 45, 46, 47]]])
```

Indexing and Slicing Arrays

```
>>> h[1,2,0]
24
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11],
        [12, 13, 14, 15]],

       [[16, 17, 18, 19],
        [20, 21, 22, 23],
        [24, 25, 26, 27],
        [28, 29, 30, 31]],

       [[32, 33, 34, 35],
        [36, 37, 38, 39],
        [40, 41, 42, 43],
        [44, 45, 46, 47]]])
```

2 → [24, 25, 26, 27]

1 ←

Indexing and Slicing Arrays

```
>>> h[1,:,0]
array([16, 20, 24, 28])
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11],
        [12, 13, 14, 15]],

       [[16, 17, 18, 19],
        [20, 21, 22, 23],
        [24, 25, 26, 27],
        [28, 29, 30, 31]],

       [[32, 33, 34, 35],
        [36, 37, 38, 39],
        [40, 41, 42, 43],
        [44, 45, 46, 47]]])
```

: → [16, 17, 18, 19], [20, 21, 22, 23], [24, 25, 26, 27], [28, 29, 30, 31]

1 ←

Indexing and Slicing Arrays

```
>>> h[1,:,:]
```

an alternative to repeating the `:` is to use *ellipsis*: `h[1,...]`

```
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11],
        [12, 13, 14, 15]],

       [[16, 17, 18, 19],
        [20, 21, 22, 23],
        [24, 25, 26, 27],
        [28, 29, 30, 31]],

       [[32, 33, 34, 35],
        [36, 37, 38, 39],
        [40, 41, 42, 43],
        [44, 45, 46, 47]]])
```

← 1

Polynomials

(One dimensional) polynomials in numpy are conveniently described by the `poly1d` class: e.g. `P(x) = x² - x - 6`:

```
>>> p = np.poly1d([1,-1,-6])
```

enter the coefficients in *decreasing* order of power-of-*x*

```
>>> p.order
2

>>> p(0.5)
-6.25
```

evaluate the polynomial at 0.5

```
>>> p.r
array([ 3., -2.])
```

roots

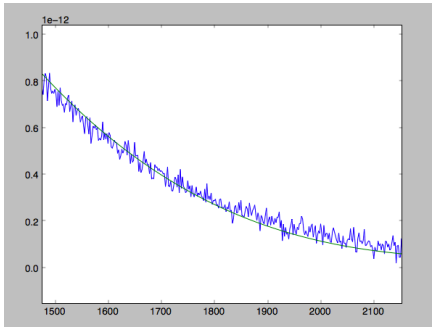
```
>>> p * p
poly1d([  1,  -2, -11,  12,  36])
```

also: polynomial division, addition, subtraction, etc.: see the numpy documentation: http://docs.scipy.org/doc/numpy/reference/routines.poly.html

Polynomials

Polynomial fitting. Given some data (arrays of *x*,*y* points), find the best-fit polynomial of a particular order to the data.

```
>>> fit_coeffs = np.polyfit(x, y, 3)
>>> fit_poly = np.poly1d(fit_coeffs)
```



Matrix Tools

For example,

```
>>> A = np.array(((3., 2., 6.), (2., 2., 5.),
                  (-2., -1., -4.)))
>>> b = np.array((0., 1., 2.,))
>>> np.trace(A)
1.0
>>> np.linalg.eigvals(A)
array([-1.        ,  0.99999995,  1.00000005])
>>> np.linalg.det(A)
-0.99999999999999989
```

note the finite precision here

```
>>> np.linalg.solve(A,b)
array([ 2.,  6., -3.])
```

ie solve for `x = (x1,x2,x3)`:

$$\begin{bmatrix} 3 & 2 & 6 \\ 2 & 2 & 5 \\ -2 & -1 & -4 \end{bmatrix} \cdot x = \begin{bmatrix} 0 \\ 1 \\ 2 \end{bmatrix}$$

Matrix Tools

For example,

```
>>> A = np.array(((3., 2., 6.), (2., 2., 5.),
                  (-2., -1., -4.)))
>>> np.linalg.inv(A)
array([[ 3., -2.,  2.],
       [ 2.,  0.,  3.],
       [-2.,  1., -2.]])
```

matrix inverse

if A is not invertible, a `SingularMatrix` exception is raised

... and many more. See the `linalg` documentation:
http://docs.scipy.org/doc/numpy/reference/routines.linalg.html

Matrix Tools

Since the `inv()` method returns an array object, it can be chained to another method. For example, to solve the set of linear equations:

$$\begin{aligned} 3x + 2y + 6z &= 5 \\ 2x + 2y + 5z &= 2.75 \\ -2x - y - 4z &= -4 \end{aligned}$$

```
>>> A = np.array(((3., 2., 6.), (2., 2., 5.),
                  (-2., -1., -4.)))
>>> b = np.array((1.5, -2., 0.75))
>>> x = np.linalg.inv(A).dot(b)
>>> x
array([ 1.5 , -2.  ,  0.75])
```

Statistical Methods

For example, given two one-dimensional arrays, `a` and `b`, each with the same number of elements:

```
>>> a = np.array([1., 2., 3., 4.])
>>> b = np.array([0., 11., 19., 32.])
>>> np.mean(b)
15.5
>>> np.std(b)
11.672617529928752
>>> np.corrcoef(a, b)
array([[ 1.        ,  0.99613934],
       [ 0.99613934,  1.        ]])
```

the correlation coefficient matrix: diagonal elements are `1.`

the covariance matrix can be obtained with `np.cov(a, b)`

Data Input

NumPy provides a very useful method, loadtxt, to read in data from a text file. The most useful arguments are:

```
numpy.loadtxt(fname, comments='#',
delimiter=None, skiprows=0, usecols=None,
unpack=False)
```

`fname`	file name
`comments`	the character indicating a comment
`delimiter`	the string separating values in a row (default is to use whitespace as the separator)
`skiprows`	skip the first `skiprows` lines (which often contain header information)
`usecols`	the column numbers to use (starting at 0)
`unpack`	if `True`, return the data in a transposed array (columns go to rows) if

Data Input

Example: UK Met Office monthly weather data for Heathrow airport since 1948:

```
Heathrow (London Airport)
Location 5078E 1767N 25m amsl
Estimated data is marked with a * after the value.
Missing data (more than 2 days missing in month) is marked by  ---.
Sunshine data taken from an automatic Kipp & Zonen sensor marked with a #...
   yyyy  mm   tmax    tmin      af    rain     sun
              degC    degC    days      mm   hours
   1948   1    8.9     3.3    ---     85.0    ---
   1948   2    7.9     2.2    ---     26.0    ---
   1948   3   14.2     3.8    ---     14.0    ---
   1948   4   15.4     5.1    ---     35.0    ---
   ...
   2012   8   23.5    14.3       0    36.4   182.6#  Provisional
```

```
>>> maxT, minT, rain = np.loadtxt(data_file,
      skiprows=7, usecols=(2, 3, 5), unpack=True)
   maxT: [8.9, 7.9, 14.2, ...]
   minT: [3.3, 2.2, 3.8, ...]
   rain: [85.0, 26.0,  14.0, ...]
```