Numerical Calculations

NumPy and SciPy

NumPy and SciPy are core libraries for scientific computing in Python, referred to as *The SciPy Stack*

NumPy package provides a high-performance multidimensional array object, and tools for working with these arrays

SciPy package extends the functionality of Numpy with a substantial set of useful algorithms for statistics, linear algebra, optimization, …

NumPy and SciPy

The SciPy stack is not shipped with Python by default

- You need to install the packages manually.

It can be installed using Python's standard pip package manager

```
$ pip install --user numpy scipy matplotlib
```

On windows, you can instead install WinPython

- It is a free Python distribution including scientific packages
- Download from https://winpython.github.io/

NumPy and SciPy

Importing the NumPy module

- The standard approach is to use a simple import statement:

```
>>> import numpy
```

- The recommended convention to import numpy is:

```
>>> import numpy as np
```

- This statement will allow you to access NumPy objects using np.X instead of numpy.X

NumPy Arrays

The central feature of NumPy is the array object class

- Similar to lists in Python
- Every element of an array must be of the same type (typically numeric)
- Operations with large amounts of numeric data are very fast and generally much more efficient than lists

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

NumPy Arrays

Manual construction of arrays

- 1-D:

```
>>> a = np.array([0, 1, 2, 3])
>>> a
array([0, 1, 2, 3])
>>> a.ndim
1
>>> a.shape
(4,)
>>> len(a)
4
```

NumPy Arrays

Manual construction of arrays

- 2-D, 3-D, …

```
>>> b = np.array([[0, 1, 2], [3, 4, 5]]) # 2 x 3 array
>>> b
array([[0, 1, 2],
       [3, 4, 5]])
>>> b.ndim
2
>>> b.shape
(2, 3)
>>> len(b) # returns the size of the first dimension
2
```

NumPy Arrays

Functions for creating arrays

- Evenly spaced:

```
>>> a = np.arange(10) # 0 .. n-1  (!)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> b = np.arange(1, 9, 2) # start, end (exclusive), step
>>> b
array([1, 3, 5, 7])
```

- By number of points:

```
>>> c = np.linspace(0, 1, 6)   # start, end, num-points
>>> c
array([ 0. ,  0.2,  0.4,  0.6,  0.8,  1. ])
>>> d = np.linspace(0, 1, 5, endpoint=False)
>>> d
array([ 0. ,  0.2,  0.4,  0.6,  0.8])
```

NumPy Arrays

Functions for creating arrays

- Common arrays:

```
>>> a = np.ones((3, 3))  # reminder: (3, 3) is a tuple
>>> a
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
>>> b = np.zeros((2, 2))
>>> b
array([[ 0.,  0.],
       [ 0.,  0.]])
>>> c = np.eye(3)
>>> c
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
>>> d = np.diag(np.array([1, 2, 3, 4]))
>>> d
array([[1, 0, 0, 0],
       [0, 2, 0, 0],
       [0, 0, 3, 0],
       [0, 0, 0, 4]])
```

NumPy Arrays

Functions for creating arrays

- Random numbers:

```
>>> a = np.random.rand(4)       # uniform in [0, 1]
>>> a
array([ 0.95799151,  0.14222247,  0.08777354,  0.51887998])

>>> b = np.random.randn(4)      # Gaussian
>>> b
array([ 0.37544699, -0.11425369, -0.47616538,  1.79664113])

>>> np.random.seed(1234)        # Setting the random seed
```

NumPy Arrays

Array element type

- NumPy arrays comprise elements of a single data type
- The type object is accessible through the .dtype attribute
- NumPy auto-detects the data-type from the input

```
>>> a = np.array([1, 2, 3])
>>> a.dtype
dtype('int64')

>>> b = np.array([1., 2., 3.])
>>> b.dtype
dtype('float64')
```

NumPy Arrays

Array element type

- You can explicitly specify which data-type you want:

```
>>> c = np.array([1, 2, 3], dtype=float)
>>> c.dtype
dtype('float64')
```

- The default data type is floating point:

```
>>> a = np.ones((3, 3))
>>> a.dtype
dtype('float64')
```

NumPy Arrays

Indexing

- The items of an array can be accessed and assigned to the same way as other Python sequences:

```
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a[0], a[2], a[-1]
(0, 2, 9)
```

NumPy Arrays

Indexing

- For multidimensional arrays, indexes are tuples of integers:

```
>>> a = np.diag(np.arange(3))
>>> a
array([[0, 0, 0],
       [0, 1, 0],
       [0, 0, 2]])
>>> a[1, 1]
1
>>> a[2, 1] = 10 # third line, second column
>>> a
array([[ 0,  0,  0],
       [ 0,  1,  0],
       [ 0, 10,  2]])
>>> a[1]
array([0, 1, 0])
```

NumPy Arrays

Slicing

- Arrays, like other Python sequences can also be sliced:

```
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a[2:9:3] # [start:end:step]
array([2, 5, 8])
```

- All three slice components are not required: by default, start is 0, end is the last and step is 1:

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

Operations on Arrays

Basic operations

- With scalars:

```
>>> a = np.array([1, 2, 3, 4])
>>> a + 1
array([2, 3, 4, 5])
>>> 2**a
array([ 2, 4, 8, 16])
```

- All arithmetic operates elementwise:

```
>>> b = np.ones(4) + 1
>>> a - b
array([-1., 0., 1., 2.])
>>> a * b
array([ 2., 4., 6., 8.])
```

Operations on Arrays

Other operations

- Comparisons

```
>>> a = np.array([1, 2, 3, 4])
>>> b = np.array([4, 2, 2, 4])
>>> a == b
array([False, True, False, True], dtype=bool)
>>> a > b
array([False, False, True, False], dtype=bool)
```

Operations on Arrays

Other operations

- Array-wise comparisons:

```
>>> a = np.array([1, 2, 3, 4])
>>> b = np.array([4, 2, 2, 4])
>>> c = np.array([1, 2, 3, 4])
>>> np.array_equal(a, b)
False
>>> np.array_equal(a, c)
True
```

Operations on Arrays

Other operations

- Transposition:

```
>>> a = np.array([[ 0., 1., 1.],
                  [ 0., 0., 1.],
                  [ 0., 0., 0.]])
>>> a.T
array([[ 0., 0., 0.],
       [ 1., 0., 0.],
       [ 1., 1., 0.]])
```

Polynomials

NumPy supplies methods for working with polynomials.

- We save the coefficients of a polynomial in an array
- For example: $x^3 + 4x^2 - 2x + 3$

```
>>> p = np.array([1, 4, -2, 3])
```

- Evaluation and root fining:

```
>>> np.polyval(p, [1, 2, 3])
array([6, 23, 60])
>>> np.roots(p)
array([-4.57974010+0.j        ,  0.28987005+0.75566815j,
        0.28987005-0.75566815j])
```

Polynomials

NumPy supplies methods for working with polynomials.

- We save the coefficients of a polynomial in an array
- For example: $x^3 + 4x^2 - 2x + 3$

```
>>> p = np.array([1, 4, -2, 3])
```

- Integration and derivation:

```
>>> np.polyint(p)
array([ 0.25 ,  1.33333333, -1. ,   3. ,   0. ])
>>> np.polyder(p)
array([ 3, 8, -2 ])
```

Polynomials

NumPy supplies methods for working with polynomials.

- We save the coefficients of a polynomial in an array
- For example: $x^3 + 4x^2 - 2x + 3$

```
>>> p = np.array([1, 4, -2, 3])
```

- Addition and subtraction:

```
>>> q = np.array([2, 7]) # 2x + 7
>>> np.polyadd(p, q)                 # addition
array([ 1, 4, 0, 10 ])
>>> np.polysub(p, q)                 # subtraction
array([ 1, 4, -4, -4 ])
```

Polynomials

NumPy supplies methods for working with polynomials.

- We save the coefficients of a polynomial in an array
- For example: $x^3 + 4x^2 - 2x + 3$

```
>>> p = np.array([1, 4, -2, 3])
```

- Multiplication and division

```
>>> q = np.array([2, 7]) # 2x + 7
>>> np.polymul(p, q)                  # multiplication
array([ 2, 15, 24, -8, 21])
>>> np.polydiv(p, q)                  # division
(array([ 0.5  ,  0.25 , -1.875]), array([ 16.125]))
```

Numerical Integration

Numerical integration is the approximate computation of an integral using numerical techniques

SciPy provides a number of integration routines. A general purpose tool to solve integrals of the kind:

$$I = \int_a^b f(x)dx$$

- It is provided by the **quad**() function of the **scipy.integrate** module

Numerical Integration

Suppose we want to evaluate the integral

$$I = \int_0^{2\pi} e^{-x} \sin(x)\, dx$$

```
>>> import numpy as np
>>> import scipy.integrate as si
>>> f = lambda x: np.exp(-x) * np.sin(x)
>>> I = si.quad(f, 0, 2 * np.pi)
>>> print(I)
(0.49906627863414593, 6.023731631928322e-15)
```

Numerical Integration

Infinite bound integral:

$$I = \int_0^{\infty} e^{-x} \sin(x)\, dx$$

```
>>> import numpy as np
>>> import scipy.integrate as si
>>> f = lambda x: np.exp(-x) * np.sin(x)
>>> I = si.quad(f, 0, np.inf)
>>> print(I)
(0.5000000000000002, 1.4875911931534648e-08)
```

Numerical Integration

Case Study

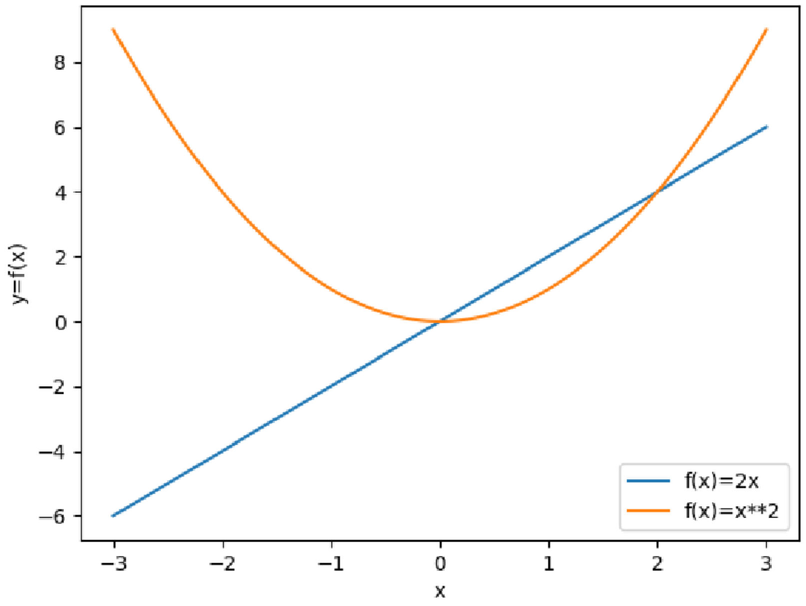
- Plot $f(x) = 2x$ and its integral

```
import numpy as np
import scipy.integrate as si
import matplotlib.pyplot as plt
f = lambda x: 2*x
g = lambda x: si.quad(f, 0, x)[0]
x = np.linspace(-3,3,1000)
yf = f(x)
yg = np.array([g(i) for i in x])
plt.plot(x,yf)
plt.plot(x,yg)
plt.legend(['f(x)=2x', 'f(x)=x**2'])
plt.xlabel('x')
plt.ylabel('y=f(x)')
plt.show()
```

Numerical Integration

Case Study

- Plot $f(x) = 2x$ and its integral



Linear Algebra

Finding determinant

- It is provided by the **det**() function of the **scipy.linalg** module

```
>>> import numpy as np
>>> from scipy import linalg
>>> A = np.array([[1,2],[3,4]])
>>> A
array([[1, 2],
      [3, 4]])
>>> linalg.det(A)
-2.0
```

Linear Algebra

Matrix inversion

- It is provided by the **inv**() function of the **scipy.linalg** module

```
>>> import numpy as np
>>> from scipy import linalg
>>> A = np.array([[1,3,5],[2,5,1],[2,3,8]])
>>> A
array([[1, 3, 5],
       [2, 5, 1],
       [2, 3, 8]])
>>> linalg.inv(A)
array([[-1.48,  0.36,  0.88],
       [ 0.56,  0.08, -0.36],
       [ 0.16, -0.12,  0.04]])
>>> A.dot(linalg.inv(A)) #double check
array([[  1.00000000e+00,  -1.11022302e-16,  -5.55111512e-17],
       [  3.05311332e-16,   1.00000000e+00,   1.87350135e-16],
       [  2.22044605e-16,  -1.11022302e-16,   1.00000000e+00]])
```

Linear Algebra

Solving linear system

- It is provided by the **solve**() function of the **scipy.linalg** module

Example

- Suppose it is desired to solve the following simultaneous equations:

$$\begin{aligned} x + 3y + 5z &= 10 \\ 2x + 5y + z &= 8 \\ 2x + 3y + 8z &= 3 \end{aligned}$$

- We could find the solution vector using a matrix inverse:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 5 & 1 \\ 2 & 3 & 8 \end{bmatrix}^{-1} \begin{bmatrix} 10 \\ 8 \\ 3 \end{bmatrix} = \frac{1}{25} \begin{bmatrix} -232 \\ 129 \\ 19 \end{bmatrix} = \begin{bmatrix} -9.28 \\ 5.16 \\ 0.76 \end{bmatrix}$$

Linear Algebra

Solving linear system

- It is provided by the **solve**() function of the **scipy.linalg** module

```
>>> import numpy as np
>>> from scipy import linalg
>>> A = np.array([[1, 2], [3, 4]])
>>> A
array([[1, 2],
       [3, 4]])
>>> b = np.array([[5], [6]])
>>> b
array([[5],
       [6]])
>>> np.linalg.solve(A, b)  # fast
array([[-4. ],
       [ 4.5]])
>>> A.dot(np.linalg.solve(A, b)) - b  # check
array([[ 0.],
       [ 0.]])
```