Numpy Arrays

We start by importing the module numpy as *import numpy as np*
Let x = [[1, 2], [3, 4]] be a two dimensional list (lists within a list). We convert it into an numpy array as *x = np.array (x)*

Now, *x.shape* gives the shape of this array which is (2, 2) in this case.

[In case of $1 \times n$ arrays, the result is printed as (n,)]

x.ndim gives the dimension of this array which is 2 in this case.

y = x.reshape (4,1) converts the array into a 4×1 array. The result is stored in y, but the array **x remains unchanged**.

x.resize (4,1) converts the array into a 4×1 array. The array **x itself changes**.

If x = [[1, 2], [3, 4]] and y = [[0, 1], [2, 3]] are two numpy arrays :
x + y, x – y, x* y, x/y produce another array with term by term addition, subtraction, etc. For example, x + y = [[1, 3], [5,7]], x*y = [[0, 2], [6, 12]].
Clearly, x + y and x – y give the results of matrix addition and matrix subtraction. However, x*y is **not the same as matrix multiplication**.

np.add (x, y) gives us the result of **matrix addition** which is the same as x + y.

np.dot (x, y) gives us the result of **matrix multiplication** which is not the same as x * y.

np.trace(x) gives the trace of x, i.e., sum of the diagonal elements : $x_{11} + x_{22}$

x.T produces the matrix transpose of x. i.e. x^T.

If x = [1, 1, 2] and y = [1, 2, 1] are two **one dimensional** numpy arrays, they may be treated as two vectors.

np.inner (x, y) or *np.vdot (x, y)* gives the inner product (i.e. the dot product) of these two vectors, which equals 5 in this case. Note that *np.dot* **does not** produce the so-called dot product of two vectors.

np.cross (x, y) gives the cross product x and y.

linspace (a, b, n) creates an array with ‘n’ number of entries, starting with ‘a’ and finishing with ‘b’ (both the end points being included), e.g., *linspace (1, 5, 5)* = [1, 2, 3, 4, 5]

arange (a, b, h) creates an array as [a, a + h, a + 2h, · · ·], which finishes **before ‘b’ is reached**, e.g., *linspace (1, 4, 0.5)* = [1.0, 1.5, 2.0, 2.5, 3.0, 3.5]

Slicing of arrays

Let x = [5, 4, 0, 1, 2, 6] be a numpy array.

x [m : n: p] produces a sliced array starting with the m-th element, finishing at the (n – 1)-th element and jumping with steps of p. For example, *x [1 : 4: 2] = [4, 1]*

x [: 4: 2] starts from the beginning, i.e. yields *[5, 0]*

x [1: : 2] goes up to the end i.e. yields *[4, 1, 6]*

x [1: 4:] takes the step size = 1, by default, i.e. yields *[4, 0, 1]*

Eigen value Problem

Let us import a module linalg from numpy as *import numpy.linalg as lin*
Let x = [[0, 1], [1, 0]] be a two dimensional numpy array, which is equivalent to a matrix.

lin.eigvals (x) gives the eigenvalues of x as a tuple, i.e., as *(1, – 1)*.

We can unpack them as *a, b = lin.eigvals (x)*

lin.eig (x) gives the eigenvalues and normalized eigenvectors as a tuple of arrays. The eigen values are packed as one array and the eigen vectors are packed as another 2 - dimensional array. We can unpack the tuple as *l, v = lin.eig (x)*
Now, *l* contains the eigen values and *v* contain the eigen vectors. We can further separate the eigen vectors by slicing as *v1 = v[: , 0]* (this collects the 0-th column of all the rows)
v2 = v[: , 1] (this collects the 1-th column of all the rows)

Trapezoidal rule

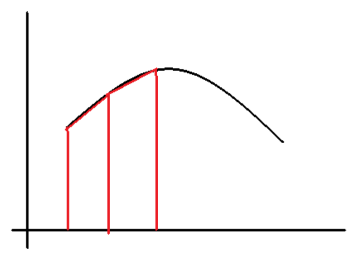
The interval is divided into small segments. The points on the curve are joined by **straight lines**, so that these joining lines and the ordinates form thin **trapeziums** (hence the name).

Area of the first trapezium = $(y_0 + y_1)/2 \times dx$

Area of the second trapezium = $(y_1 + y_2)/2 \times dx$

. . . .

Total area = $[y_0 + 2y_1 + 2y_2 + \ldots y_n] \times dx /2$



```
# Integration by Trapezoidal Rule
# Integrating sin (x) from 0 to pi = 3.14159
import numpy as np
x = np.linspace (0, 3.14159, 21)
y = np.sin (x)
h = 3.14159/20
sum = y[0] + y[20]
for i in range (1, 20) :
   sum = sum + 2*y[i]
area = sum * h/2
print ('Area = ', area)
```

Or,

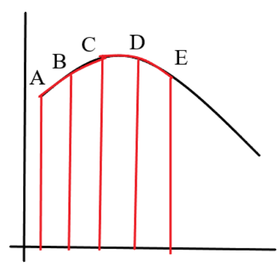
```
# Integrating sin (x) from 0 to pi
import math
def f(x) :
   return math.sin(x)
a = 0.0
b = 3.14159
n = 20
h = (b - a)/n
sum = f(a) + f(b)
for i in range (1, 20) :
   sum = sum + 2 * f(a + i*h)
area = sum *h/2
print ("Area = ", area)
```

```
# Integration by Trapezoidal Rule, using numpy.trapz
# Integrating (1 – x) from 0 to 1
import numpy as np
x = np.linspace (0, 1, 10)
y = 1 - x
area = np.trapz (y, x)
print ('Area = ', area)
```

```
# Integration by Trapezoidal Rule, using scipy.integrate.trapz
import numpy as np
import scipy.integrate as sc
x = np.arange (0, 1.1, 0.1)
y = 1 - x
area = sc.trapz (y, x)
print ('Area = ', area)
```

Simpson's 1/3 Rule

The interval is divided into small segments. Every three successive points on the curve are joined by a small **parabola, i.e. a second degree curve** of the form : $y = a + bx + cx^2$, in contrast to the Trapezoidal rule, where every two successive points are joined by a small **straight line, i.e. a first degree curve** of the form : $y = a + bx$. For example, the points A, B, C are on a parabola, while the points C, D, E are on a separate parabola.



The area under the first parabola ABC = $(y_0 + 4y_1 + y_2) \times dx/3$

The area under the first parabola CDE = $(y_2 + 4y_3 + y_4) \times dx/3$

. . . .

Total area = $[y_0 + 4y_1 + 2y_2 + 4y_3 + 2y_4 + \ldots y_n] \times dx/3$

(Hence the name)

The odd numbered ordinates are multiplied by 4, while the even numbered ordinates are multiplied by 2 and the ordinates at the two end are multiplied by 1.

- Note that the number of intervals must be even, i.e., the number of points should be odd.

```
# Integration by Simpson's Rule
def f(x) :
   return (1 - x*x)
a = input ("Initial x ")
a = float (a)
b = input ("Final x ")
b = float (b)
n = input ('No. of intervals ')
n = int (n)
```

```
h = (b - a)/n
sum = f(a) + f(b)
for i in range (1, n, 2) :
    sum = sum + 4 * f(a + i*h)
for i in range (2, n-1, 2) :
    sum = sum + 2 * f(a +i*h)
area = sum *h/3
print ("Area = ", area)
```

```
# Integration by Simpson’s 1/3  Rule, using scipy.integrate.simps
import numpy as np
import scipy.integrate as sc
x = np.arange (0, 1.1, 0.1)
y = 1 - x
area = sc.simps (y, x)
print ('Area = ', area)
```

Lagrange Interpolation

For fitting, say three data points, we take a polynomial of the form :

$$y = A\,(x - x_2)\,(x - x_3) + B\,(x - x_1)\,(x - x_3) + C\,(x - x_1)\,(x - x_2)$$

Note that we have avoided the factor $(x - x_1)$ in the first term, the factor $(x - x_2)$ in the second term and so on. Setting $x = x_1$ immediately kills all the factors except the first.

$$\text{So, } y = y_1 \text{ at } x = x_1 \;\Rightarrow\; A = y_1/(x_1 - x_2)\,(x_1 - x_3)$$
$$y = y_2 \text{ at } x = x_2 \;\Rightarrow\; B = y_2/(x_2 - x_1)\,(x_2 - x_3)$$
$$y = y_3 \text{ at } x = x_3 \;\Rightarrow\; C = y_3/(x_3 - x_1)\,(x_3 - x_2)$$

```
# Lagrange's Interpolation
n = input ('Tell me the number of data points')
n = int (n)
x = []
y = []
print ('Input the values of xi and yi')
for i in range (n) :
    xi = float (input())
    x.append (xi)
    yi = float (input())
    y.append (yi)
xx = input ('Give me the interpolating point')
xx = float (xx)
sum = 0.0
for i in range in range (n) :
      prod = 1.0
      for j in range (n) :
          if j != i :
             prod = prod *(xx - x[j])/(x[i] - x[j])
      sum = sum + y[i]*prod
print ('At x = ', xx, 'y = ', sum)
```

```
''' Lagrange Interpolation
    using scipy.interpolate.lagrange '''
import scipy.interpolate as sc
x = [1, 2, 3]
y = [1, 4, 9]
p = sc.lagrange (x, y)
print (p)
l = p.coef
print (l)
```