SciPy, pronounced as Sigh Pi, is a scientific python open source, distributed under the BSD licensed library to perform Mathematical, Scientific and Engineering Computations.The SciPy library depends on NumPy, which provides convenient and fast N-dimensional array manipulation. The SciPy library is built to work with NumPy arrays and provides many user-friendly and efficient numerical practices such as routines for numerical integration and optimization. Together, they run on all popular operating systems, are quick to install and are free of charge. NumPy and SciPy are easy to use, but powerful enough to depend on by some of the world's leading scientists and engineers.

SciPy Sub-packages

SciPy is organized into sub-packages covering different scientific computing domains. These are summarized in the following table –

scipy.cluster	Vector quantization / Kmeans
scipy.constants	Physical and mathematical constants
scipy.fftpack	Fourier transform
scipy.integrate	Integration routines
scipy.interpolate	Interpolation
scipy.io	Data input and output
scipy.linalg	Linear algebra routines
scipy.ndimage	n-dimensional image package
scipy.odr	Orthogonal distance regression
scipy.optimize	Optimization
scipy.signal	Signal processing
scipy.sparse	Sparse matrices
scipy.spatial	Spatial data structures and algorithms
scipy.special	Any special mathematical functions
scipy.stats	Statistics

SciPy is built using the optimized **ATLAS LAPACK** and **BLAS** libraries. It has very fast linear algebra capabilities. All of these linear algebra routines expect an object that can be converted into a two-dimensional array. The output of these routines is also a two-dimensional array.

SciPy.linalg vs NumPy.linalg

A scipy.linalg contains all the functions that are in numpy.linalg. Additionally, scipy.linalg also has some other advanced functions that are not in numpy.linalg. Another advantage of using scipy.linalg over numpy.linalg is that it is always compiled with BLAS/LAPACK support, while for NumPy this is optional. Therefore, the SciPy version might be faster depending on how NumPy was installed.

Linear Equations

The **scipy.linalg.solve** feature solves the linear equation a * x + b * y = Z, for the unknown x, y values.

As an example, assume that it is desired to solve the following simultaneous equations.

$$x + 3y + 5z = 10$$

$$2x + 5y + z = 8$$

$$2x + 3y + 8z = 3$$

To solve the above equation for the x, y, z values, we can find the solution vector using a matrix inverse as shown below.

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

However, it is better to use the **linalg.solve** command, which can be faster and more numerically stable.

The solve function takes two inputs 'a' and 'b' in which 'a' represents the coefficients and 'b' represents the respective right hand side value and returns the solution array.

Let us consider the following example.

```
#importing the scipy and numpy packages
from scipy import linalg
import numpy as np

#Declaring the numpy arrays
a = np.array([[3, 2, 0], [1, -1, 0], [0, 5, 1]])
b = np.array([2, 4, -1])

#Passing the values to the solve function
x = linalg.solve(a, b)

#printing the result array
print x
```

The above program will generate the following output.

```
array([ 2., -2., 9.])
```

Finding a Determinant

The determinant of a square matrix A is often denoted as |A| and is a quantity often used in linear algebra. In SciPy, this is computed using the **det()** function. It takes a matrix as input and returns a scalar value.

Let us consider the following example.

```
#importing the scipy and numpy packages
from scipy import linalg
import numpy as np

#Declaring the numpy array
A = np.array([[1,2],[3,4]])

#Passing the values to the det function
x = linalg.det(A)

#printing the result
print x
```

The above program will generate the following output.

```
-2.0
```

Eigenvalues and Eigenvectors

The eigenvalue-eigenvector problem is one of the most commonly employed linear algebra operations. We can find the Eigen values (λ) and the corresponding Eigen vectors (v) of a square matrix (A) by considering the following relation −

$$Av = \lambda v$$

scipy.linalg.eig computes the eigenvalues from an ordinary or generalized eigenvalue problem. This function returns the Eigen values and the Eigen vectors.

Let us consider the following example.

```
#importing the scipy and numpy packages
from scipy import linalg
import numpy as np

#Declaring the numpy array
A = np.array([[1,2],[3,4]])

#Passing the values to the eig function
l, v = linalg.eig(A)

#printing the result for eigen values
print l

#printing the result for eigen vectors
print v
```

The above program will generate the following output.

```
array([-0.37228132+0.j, 5.37228132+0.j]) #--Eigen Values
array([[-0.82456484, -0.41597356], #--Eigen Vectors
       [ 0.56576746, -0.90937671]])
```

Singular Value Decomposition

A Singular Value Decomposition (SVD) can be thought of as an extension of the eigenvalue problem to matrices that are not square.

The **scipy.linalg.svd** factorizes the matrix 'a' into two unitary matrices 'U' and 'Vh' and a 1-D array 's' of singular values (real, non-negative) such that a == U*S*Vh, where 'S' is a suitably shaped matrix of zeros with the main diagonal 's'.

Let us consider the following example.

```
#importing the scipy and numpy packages
from scipy import linalg
import numpy as np

#Declaring the numpy array
a = np.random.randn(3, 2) + 1.j*np.random.randn(3, 2)

#Passing the values to the eig function
U, s, Vh = linalg.svd(a)

# printing the result
print U, Vh, s
```

The above program will generate the following output.

```
(
   array([
      [ 0.54828424-0.23329795j, -0.38465728+0.01566714j,
      -0.18764355+0.67936712j],
      [-0.27123194-0.5327436j , -0.57080163-0.00266155j,
      -0.39868941-0.39729416j],
      [ 0.34443818+0.4110186j , -0.47972716+0.54390586j,
      0.25028608-0.35186815j]
   ]),

   array([ 3.25745379, 1.16150607]),

   array([
      [-0.35312444+0.j , 0.32400401+0.87768134j],
      [-0.93557636+0.j , -0.12229224-0.33127251j]
   ])
)
```

Fourier Transformation is computed on a time domain signal to check its behavior in the frequency domain. Fourier transformation finds its application in disciplines such as signal and noise processing, image processing, audio signal processing, etc. SciPy offers the fftpack module, which lets the user compute fast Fourier transforms.

Following is an example of a sine function, which will be used to calculate Fourier transform using the fftpack module.

Fast Fourier Transform

Let us understand what fast Fourier transform is in detail.

One Dimensional Discrete Fourier Transform

The FFT y[k] of length N of the length-N sequence x[n] is calculated by fft() and the inverse transform is calculated using ifft(). Let us consider the following example

```
#Importing the fft and inverse fft functions from fftpackage
from scipy.fftpack import fft

#create an array with random n numbers
x = np.array([1.0, 2.0, 1.0, -1.0, 1.5])

#Applying the fft function
y = fft(x)
print y
```

The above program will generate the following output.

```
[ 4.50000000+0.j            2.08155948-1.65109876j   -1.83155948+1.60822041j
 -1.83155948-1.60822041j    2.08155948+1.65109876j ]
```

Let us look at another example

```
#FFT is already in the workspace, using the same workspace to for inverse
transform

yinv = ifft(y)

print yinv
```

The above program will generate the following output.

```
[ 1.0+0.j   2.0+0.j   1.0+0.j   -1.0+0.j   1.5+0.j ]
```

The **scipy.fftpack** module allows computing fast Fourier transforms. As an illustration, a (noisy) input signal may look as follows −

```
import numpy as np
time_step = 0.02
period = 5.
time_vec = np.arange(0, 20, time_step)
sig = np.sin(2 * np.pi / period * time_vec) + 0.5
*np.random.randn(time_vec.size)
print sig.size
```

We are creating a signal with a time step of 0.02 seconds. The last statement prints the size of the signal sig. The output would be as follows –

```
1000
```

We do not know the signal frequency; we only know the sampling time step of the signal sig. The signal is supposed to come from a real function, so the Fourier transform will be symmetric. The **scipy.fftpack.fftfreq()** function will generate the sampling frequencies and **scipy.fftpack.fft()** will compute the fast Fourier transform.

Let us understand this with the help of an example.

```
from scipy import fftpack
sample_freq = fftpack.fftfreq(sig.size, d = time_step)
sig_fft = fftpack.fft(sig)
print sig_fft
```

The above program will generate the following output.

```
array([
   25.45122234 +0.00000000e+00j,    6.29800973 +2.20269471e+00j,
   11.52137858 -2.00515732e+01j,    1.08111300 +1.35488579e+01j,
   ……..])
```

Discrete Cosine Transform

A **Discrete Cosine Transform (DCT)** expresses a finite sequence of data points in terms of a sum of cosine functions oscillating at different frequencies. SciPy provides a DCT with the function **dct** and a corresponding IDCT with the function **idct**. Let us consider the following example.

```
from scipy.fftpack import dct
print dct(np.array([4., 3., 5., 10., 5., 3.]))
```

The above program will generate the following output.

```
array([ 60.,  -3.48476592,  -13.85640646,  11.3137085,  6.,  -6.31319305])
```

The inverse discrete cosine transform reconstructs a sequence from its discrete cosine transform (DCT) coefficients. The idct function is the inverse of the dct function. Let us understand this with the following example.

```
from scipy.fftpack import dct
print idct(np.array([4., 3., 5., 10., 5., 3.]))
```

The above program will generate the following output.

```
array([ 39.15085889, -20.14213562, -6.45392043, 7.13341236,
8.14213562, -3.83035081])
```

SciPy constants package provides a wide range of constants, which are used in the general scientific area.

SciPy Constants Package

The **scipy.constants package** provides various constants. We have to import the required constant and use them as per the requirement. Let us see how these constant variables are imported and used.

To start with, let us compare the 'pi' value by considering the following example.

```
#Import pi constant from both the packages
from scipy.constants import pi
from math import pi

print("sciPy - pi = %.16f"%scipy.constants.pi)
print("math - pi = %.16f"%math.pi)
```

The above program will generate the following output.

```
sciPy - pi = 3.1415926535897931
math - pi = 3.1415926535897931
```

List of Constants Available

The following tables describe in brief the various constants.

Mathematical Constants

Sr. No.	Constant	Description
1	pi	pi
2	golden	Golden Ratio

Physical Constants

The following table lists the most commonly used physical constants.

Sr. No.	Constant & Description
1	**c** Speed of light in vacuum
2	**speed_of_light** Speed of light in vacuum
3	**h**

	Planck constant
4	**Planck** Planck constant h
5	**G** Newton's gravitational constant
6	**e** Elementary charge
7	**R** Molar gas constant
8	**Avogadro** Avogadro constant
9	**k** Boltzmann constant
10	**electron_mass(OR) m_e** Electronic mass
11	**proton_mass (OR) m_p** Proton mass
12	**neutron_mass(OR)m_n** Neutron mass

Units

The following table has the list of SI units.

Sr. No.	Unit	Value

1	milli	0.001
2	micro	1e-06
3	kilo	1000

These units range from yotta, zetta, exa, peta, tera .....kilo, hector, ...nano, pico, ... to zepto.

Other Important Constants

The following table lists other important constants used in SciPy.

Sr. No.	Unit	Value
1	gram	0.001 kg
2	atomic mass	Atomic mass constant
3	degree	Degree in radians
4	minute	One minute in seconds
5	day	One day in seconds
6	inch	One inch in meters
7	micron	One micron in meters
8	light_year	One light-year in meters
9	atm	Standard atmosphere in pascals
10	acre	One acre in square meters
11	liter	One liter in cubic meters

12	gallon	One gallon in cubic meters
13	kmh	Kilometers per hour in meters per seconds
14	degree_Fahrenheit	One Fahrenheit in kelvins
15	eV	One electron volt in joules
16	hp	One horsepower in watts
17	dyn	One dyne in newtons
18	lambda2nu	Convert wavelength to optical frequency

Remembering all of these are a bit tough. The easy way to get which key is for which function is with the **scipy.constants.find()** method. Let us consider the following example.

```
import scipy.constants
res = scipy.constants.physical_constants["alpha particle mass"]
print res
```

The above program will generate the following output.

```
[
   'alpha particle mass',
   'alpha particle mass energy equivalent',
   'alpha particle mass energy equivalent in MeV',
   'alpha particle mass in u',
   'electron to alpha particle mass ratio'
]
```

K-means clustering is a method for finding clusters and cluster centers in a set of unlabelled data. Intuitively, we might think of a cluster as – comprising of a group of data points, whose inter-point distances are small compared with the distances to points outside of the cluster. Given an initial set of K centers, the K-means algorithm iterates the following two steps –

- For each center, the subset of training points (its cluster) that is closer to it is identified than any other center.
- The mean of each feature for the data points in each cluster are computed, and this mean vector becomes the new center for that cluster.

These two steps are iterated until the centers no longer move or the assignments no longer change. Then, a new point **x** can be assigned to the cluster of the closest prototype. The SciPy library provides a good implementation of the K-Means algorithm through the cluster package. Let us understand how to use it.

K-Means Implementation in SciPy

We will understand how to implement K-Means in SciPy.

Import K-Means

We will see the implementation and usage of each imported function.

```
from SciPy.cluster.vq import kmeans,vq,whiten
```

Data generation

We have to simulate some data to explore the clustering.

```
from numpy import vstack,array
from numpy.random import rand

# data generation with three features
data = vstack((rand(100,3) + array([.5,.5,.5]),rand(100,3)))
```

Now, we have to check for data. The above program will generate the following output.

```
array([[ 1.48598868e+00, 8.17445796e-01, 1.00834051e+00],
       [ 8.45299768e-01, 1.35450732e+00, 8.66323621e-01],
       [ 1.27725864e+00, 1.00622682e+00, 8.43735610e-01],
       ..............
```

Normalize a group of observations on a per feature basis. Before running K-Means, it is beneficial to rescale each feature dimension of the observation set with whitening. Each feature is divided by its standard deviation across all observations to give it unit variance.

Whiten the data

We have to use the following code to whiten the data.

```
# whitening of data
data = whiten(data)
```

Compute K-Means with Three Clusters

Let us now compute K-Means with three clusters using the following code.

```
# computing K-Means with K = 3 (2 clusters)
centroids,_ = kmeans(data,3)
```

The above code performs K-Means on a set of observation vectors forming K clusters. The K-Means algorithm adjusts the centroids until sufficient progress cannot be made, i.e. the change in distortion, since the last iteration is less than some threshold. Here, we can observe the centroid of the cluster by printing the centroids variable using the code given below.

```
print(centroids)
```

The above code will generate the following output.

```
print(centroids)[ [ 2.26034702  1.43924335  1.3697022 ]
                  [ 2.63788572  2.81446462  2.85163854]
                  [ 0.73507256  1.30801855  1.44477558] ]
```

Assign each value to a cluster by using the code given below.

```
# assign each sample to a cluster
clx,_ = vq(data,centroids)
```

The **vq** function compares each observation vector in the 'M' by 'N' **obs** array with the centroids and assigns the observation to the closest cluster. It returns the cluster of each observation and the distortion. We can check the distortion as well. Let us check the cluster of each observation using the following code.

```
# check clusters of observation
print clx
```

The above code will generate the following output.

```
array([1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 0, 0, 1, 0, 2, 0, 2,
0, 1, 1, 1,
0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 0, 0, 0,
1, 1, 0, 0,
0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0,
0, 0, 0, 0,
0, 1,  0, 0, 0, 0, 1, 0, 0, 1, 2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 0, 2, 0, 2, 2, 2,
2, 2, 0, 0,
2, 2, 2, 1, 0, 2, 0, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 0, 2, 2, 1, 2, 2, 2, 2, 2, 2,
2, 2, 2, 2,
2, 2, 0, 2, 0, 2, 2, 2, 2, 2, 2, 2, 2, 2, 0, 2, 2, 2, 2, 0, 2, 2, 2, 2, 2, 2,
2, 2, 2, 2,
2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2],
dtype=int32)
```

The distinct values 0, 1, 2 of the above array indicate the clusters.