## What is SciPy

It is used to solve the complex scientific and mathematical problems. It is built on top of the Numpy extension, which means if we import the SciPy, there is no need to import Numpy. The **Scipy** is pronounced as **Sigh pi**, and it depends on the Numpy, including the appropriate and fast N-dimension array manipulation.

It provides many user-friendly and effective numerical functions for numerical integration and optimization.

The **SciPy** library supports **integration, gradient optimization, special functions, ordinary differential equation solvers, parallel programming tools**, and many more. We can say that **SciPy** implementation exists in every complex numerical computation.

The **scipy** is a data-processing and system-prototyping environment as similar to MATLAB. It is easy to use and provides great flexibility to scientists and engineers.

## Numpy vs. SciPy

Numpy and SciPy both are used for mathematical and numerical analysis. Numpy is suitable for basic operations such as sorting, indexing and many more because it contains array data, whereas SciPy consists of all the numeric data.

Numpy contains many functions that are used to resolve the linear algebra, Fourier transforms, etc. whereas SciPy library contains full featured version of the linear algebra module as well many other numerical algorithms.

# SciPy Sub - Packages

SciPy has the number of sub-packages for the various scientific computing domains.

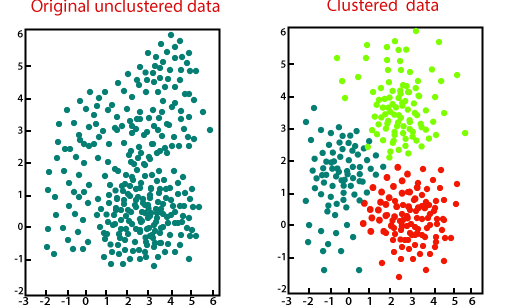
| **Sr** | **Sub-Package** | **Description** |
| --- | --- | --- |
| **1.** | scipy.cluster | Cluster algorithms are used to vector quantization/ Kmeans. |
| **2.** | scipy.constants | It represents physical and mathematical constants. |
| **3.** | scipy.fftpack | It is used for **Fourier transform**. |
| **4.** | scipy.integrate | Integration routines |
| **5.** | scipy.interpolation | Interpolation |
| **6.** | scipy.linalg | It is used for linear algebra routine. |
| **7.** | scipy.io | It is used for data input and output. |
| **8.** | scipy.ndimage | It is used for the n-dimension image. |
| **9.** | scipy.odr | Orthogonal distance regression. |
| **10.** | scipy.optimize | It is used for optimization. |
| **11.** | scipy.signal | It is used in signal processing. |
| **12.** | scipy.sparse | Sparse matrices and associated routines. |
| **13.** | scipy.spatial | Spatial data structures and algorithms. |
| **14.** | scipy.special | Special Function. |
| **15.** | scipy.stats | Statistics. |
| **16.** | scipy.weaves | It is a tool for writing. |

# SciPy Cluster

# Clustering is the procedure of dividing the datasets into groups consisting of similar data-points. There are two types of the cluster, which are:

* Central
* Hierarchy

The k- means clustering algorithm is a simple unsupervised algorithm that is used to predict groupings from within an unlabeled dataset. The prediction is based on the number of cluster centers present(k) and the nearest mean value (measured in Euclidian distance between observations).



**K-means Algorithm**

The steps are as follows, suppose we have an input x1,x2, x3,....xn, data and value K.

**Step - 1:** Select K random points as a cluster center called centroid. Suppose these are c1,c2,...ck, and it can be written as follows:

c1,c2,...ck

C is the set of all centroid.

**Step-2:** Assign each input value xi to the nearest center by calculating its Euclidean (L2) distance between the point and each centroid.

**Step-3:** In this step, we get the new centroid by calculating the average of all the points assigned to the cluster.

**Step-4:** We repeat steps 2 and 3 until none of the clusters remains unstable.

### K-means cluster and vector quantization (scipy.cluster.vq)

SciPy provides functions for K-means clustering, generating codebooks from k-means models, and quantizing vector by comparing them with centroid in a codebook.

| **Function** | **Description** |
| --- | --- |
| scipy.cluster.vq.whiten(obs, check\_finite=True ) | It normalizes a group of observation on features. |
| scipy.cluster.vq.vq(obs, code\_book,check\_finite=True) | It assigns codes from a codebook to observation. |
| scipy.cluster.vq.kmeans(obs, k\_or\_guess, iter=20, thresh=1e-05, check\_finite=True) | It performs k-means on a set of observation vectors forming k clusters. |
| scipy.cluster.vq.kmeans2(data,k,iter=10, thresh=1e-05, minit='random', missing='warn', check\_finite=True) | It classifies a set of observations into k clusters using the k-means algorithm. |

### K-Means Implementation in SciPy

Here, we will understand the implementation of K- means in SciPy

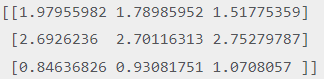
**Import K-means :** following statement is used to implement the K-means algorithm:AD

1. from SciPy.cluster.vq **import** kmeans,vq,white

**Computing K-means with three Clusters**

The K-means algorithm iterates again and again and adjusts the centroid until necessary progress cannot made change in distortion, since the last iteration is less than some threshold. Consider the following example:

1. from numpy **import** vstack,array
2. from numpy.random **import** rand
3. from scipy.cluster.vq **import** kmeans,vq,whiten
4. # data generation with three features
5. data1 = vstack((rand(100,3) + array([.5,.5,.5]),rand(100,3)))
6. data1 = whiten(data) # whitening of data
7. centroids,\_ = kmeans(data1,3) # computing K-Means with clusters
8. print(centroids)



# SciPy Constant

The scipy.constant package is available with a wide range of constants, which is used extensively in the scientific field. There are various physical, mathematical constants and units that we can import the required constants and use them as per needed.

**List of Mathematical constant**

The scipy.constant provides the following list of mathematical constants.

| **Sr. No.** | **Constants** | **Description** |
| --- | --- | --- |
| 1. | pi | pi |
| 2. | golden | Golden ratio |

Consider the following example of **scipy.constant**. Here we compare the 'pi' value by importing different modules.

1. from scipy.constants **import** pi
2. from math **import** pi
3. print("sciPy - pi Value = %.18f"%scipy.constants.pi) //3.141592653589793116
4. print("math - pi Value = %.18f"%math.pi) //3.141592653589793116

### Physical Constants

The **scipy.constant** package provides the number of physical constants. The most commonly used physical constants are the following:

| **Sr. No.** | **Physical Constants** | **Description** |
| --- | --- | --- |
| 1. | c | Speed of light in vaccum |
| 2. | speed\_of\_light | Speed of light in vaccum |
| 3. | G | Standard acceleration of gravity |
| 4. | G | Newton Constant of gravitation |
| 5. | E | Elementry charge |
| 6. | R | Molar gas constant |
| 7. | Alpha | Fine-structure constant |
| 8. | N\_A | Avagadro constant |
| 9. | K | Boltzmann constant |
| 10 | Sigma | Stefan-Boltzmann constant σ |
| 11. | m\_e | Electron mass |
| 12. | m\_p | Proton mass |
| 13. | m\_n | Neutron Mass |
| 14. | H | Plank Constant |
| 15. | Plank constant | Plank constant h |

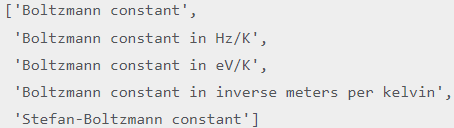
### Other important Constant

It is difficult to remember all units; the few essential constants are listed below:

| **Sr. No.** | **Units** | **Value** |
| --- | --- | --- |
| **1.** | **Mass** |  |
|  | Gram | One gram in Kilogram. |
|  | Grain | One grain in Kilogram. |
|  | Pound | One Pound in Kilogram. |
|  | Ounce | One Ounce in Kilogram. |
|  | automic\_mass | Atomics mass constant in Kilogram. |
| **2.** | **Time** |  |
|  | Minute | One minute in seconds. |
|  | Hour | One hour in seconds. |
|  | Day | One day in seconds. |
|  | Year | One year in seconds. |
| **3.** | **Length** |  |
|  | Inch | One inch in meters. |
|  | Foot | One foot in meters. |
|  | Yard | One yard in meters. |
|  | Pt | One point in meters. |
|  | Micron | One Micron in meters. |
| **4.** | **Pressure** |  |
|  | Atm | The standard atmosphere in pascals. |
|  | Atmosphere | The standard atmosphere in pascals. |
|  | Bar | One bar in Pascals. |
|  | Torr | One torr(mmHg) in pascals. |
| **5.** | **Area** |  |
|  | Hectare | One hectare in square meters. |
|  | Acre | One acre in square meters. |
| **6.** | **Speed** |  |
|  | Kmh | Kilometer per hour in meter per second. |
|  | Mph | Miles per hour in meter per second. |
|  | Mach | One Match in meter per second. |

The **scipy.constant** provides the **find()** function, which returns a list of **physical\_constant** keys containing a given string.

1. from scipy.constants **import** find, physical\_constants
2. find('boltzmann')



# 

# SciPy FFTpack

The FFT stands for **Fast Fourier Transformation**. The Fourier transformation converts the time-domain signal into the frequency domain. It breaks a waveform (a function or signal) into another replacement characterized by sine and cosine. It can convert the periodic time signal whereas the Laplace transform converts both periodic and aperiodic signal.

There is a limitation in the Fourier transformation, it can only convert the stable time signal. SciPy provides the **fftpack** module, which is used to calculate Fourier transformation.

### Fast Fourier Transform

**One Dimensional Discrete Fourier Transform**

The discrete Fourier transformation (DFT) is the most crucial discrete transform, which is used to perform Fourier analysis in many practical applications.

The FFT of length N sequence x[n] is calculated by fft() function and the inverse transform is calculated using ifft().

1. # importing the fft and inverse fft functions from fftpackage
2. from scipy.fftpack **import** fft
3. x = np.array([4.0, 2.0, 1.0, -3.0, 1.5]) #create an array with random n numbers
4. y = fft(x) #Applying the fft function



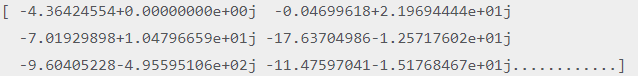
The **scipy.fftpack** module allows to compute fast Fourier transforms. We can use it for noisy signal because these signals require high computation. An example of the noisy input signal is given below:

1. time\_step = 0.02
2. period = 5.
3. time\_vector = np.arange(0, 20, time\_step)
4. sig = np.sin(2 \* np.pi / period \* time\_vec) + 0.5 \*np.random.randn(time\_vector.size)
5. print("The Signal Size is:", sig.size) //The Signal Size is: 1000

we have created a signal with a time step of 0.02 seconds. The statement prints the signal size as we can see in the output. We do not have idea about the signal frequency; we only familiar the sampling time step of the signal sig.

The **scipy.fftpack.fftfreq()** function will create the sampling frequencies and **scipy.fftpack.fft()** will compute the fast Fourier transform.

1. from scipy **import** fftpack
2. sample\_freq = fftpack.fftfreq(sig.size, d = time\_step)
3. sig\_fft = fftpack.fft(sig)
4. print(sig\_fft)



### Discrete Cosine Transform

The Discrete Cosine Transform represents data by summation of the cosine function. It transforms a sequence of the real data points into its real spectrum and therefore avoids the problem of the redundancy. DCT is mostly used in the image compressing.

DCT is similar to discrete Fourier transformation (DFT), but it uses only real numbers.

1. from scipy.fftpack **import** dct
2. **import** numpy as np
3. print(dct(np.array([6., 2., 5., 11., 5., 3.])))



**Usage of DCT:** The DCT is used in various fields. These fields are given below:AD

* Image Processing- Compression, Scientific Analysis.
* Audio Processing- Compression (MPEG or Mp3).
* Scientific Computing- Partial Differential Equation Solvers.

# SciPy Integrate

Sometimes a function is very complicated to integrate or cannot be integrated analytically; then, it can be solved by the numerical integration method. SciPy provides some routines for performing numerical integration. The **scipy.integrate** library contains most of these functions.

* **Single Integrals**

Numerical integrate is sometimes called **quadrature**. The Quad function is essential for SciPy's integration functions. The syntax of the **quad()** function is the following:

1. scipy.integrate.quad(f,a,b),

**f -** Function name to be integrate

**a-**It is a lower limit.

**b-** It is a upper limit.

Let's consider the Gaussian function, integrated over a range from a to b. We define the function**f(x)**= e-x2, this can be done using a lambda expression and apply the quad method on the given function.

1. **import** scipy.integrate
2. from numpy **import** exp
3. f= lambda x:exp(-x\*\*2)
4. i = scipy.integrate.quad(f, 0, 1)

//(0.7468241328124271, 8.291413475940725e-15)

In above program, we have used the **quad()** function that returns the two values. The first value is the integral, and the second value is an estimate of the absolute error in the value of an integer.

#### **Note:** Since quad() function requires function as the first argument, we cannot directly pass expression as the argument. It allows positive and negative infinity as limits.

### Multiple integrals

The multiple integral such as double and triple integration conclude into the functions **dblquad(),** **tplquad(),** and **nquad().** Here we consider the double integral problem to solve using the **scipy.integrate.dblquad(func,a,b,gfun,hfun).** The first argument **func** is the name of the function to be integrated and a and b are the lower and upper limit of the x variable. While gfun and hfun are names of the functions that define the lower and upper limit of the y variable.

1. **import** scipy.integrate
2. from numpy **import** exp
3. from math **import** sqrt
4. f = lambda x, y : 16\*x\*y
5. g = lambda x : 0
6. h = lambda y : sqrt(1-4\*y\*\*2)
7. i = scipy.integrate.dblquad(f, 0, 0.5, g, h) //(-0.5, 4.412025764622231e-14)

The **scipy.integarte** contains the number of other integration functions, including nquad(), which perform n-fold multiple integrations.

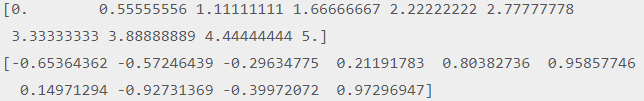
# SciPy Interpolation

Interpolation is defined as finding a value between two points on a line or a curve. first part of the word is "inter" as meaning "enter", which indicates us to look inside the data. In other words, "**The estimation of intermediate value between the precise data points is called as interpolation**". Interpolation is very useful in statistics, science, & business or when there is a need to predict the value that exists within two existing data points.

Let's have a look how the interpolation work using the **scipy.interpolation** package.

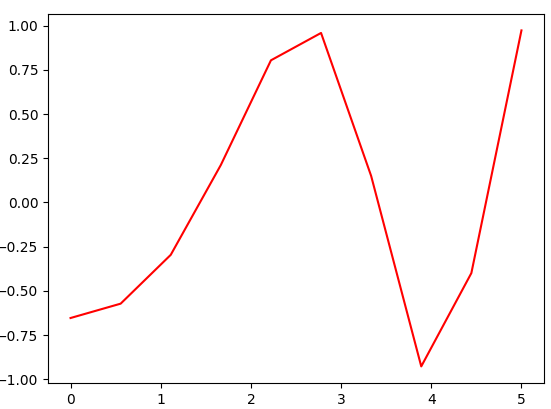
1. from scipy **import** interpolate
2. **import** matplotlib.pyplot as plt
3. x = np.linspace(0, 5, 10)
4. y = np.cos(x\*\*2/3+4)
5. print(x,y)

The **np.linspace()** function returns the interval between the given number.



We can plot those arrays as two dimension of points in space, let's consider example:

1. from scipy **import** interpolate
2. **import** matplotlib.pyplot as plt
3. x = np.linspace(0, 5, 10)
4. y = np.cos(x\*\*2/3+4)
5. plt.plot(x,y,'r')
6. plt.show()

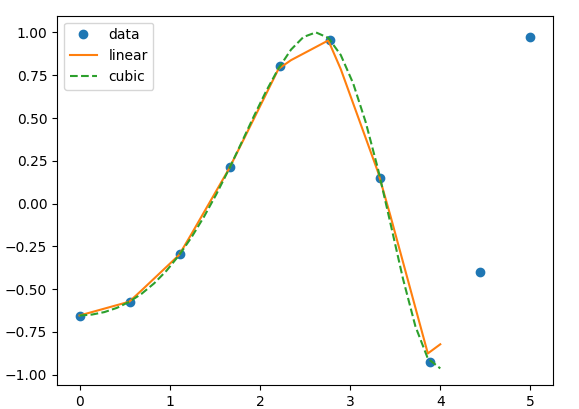


### 1-D Interpolation

scipy.interpolation provides interp1d class which is an useful method to create a function based on fixed data points.

By using above data, creating an interpolate function & draw a new interpolated graph.

1. from scipy.interpolate **import** interp1d
2. **import** matplotlib.pyplot as plt
3. fun1 = interp1d(x, y,kind = 'linear')
4. fun2 = interp1d(x, y, kind = 'cubic')
5. xnew = np.linspace(0, 4,30)
6. plt.plot(x, y, 'o', xnew, fun1(xnew), '-', xnew, fun2(xnew), '--')
7. plt.legend(['data', 'linear', 'cubic','nearest'], loc = 'best')
8. plt.show()

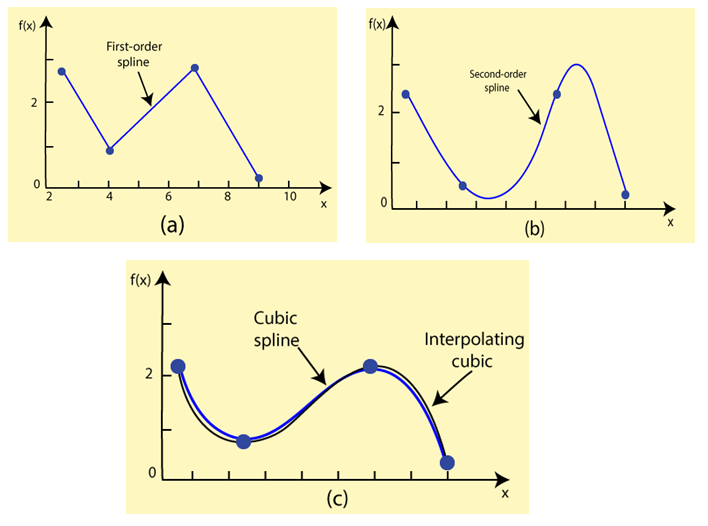


In the above program, we have created two function fun1 and fun2. The variable x contains the sample points, and variable y contains the corresponding values. The third variable **kind** represents the types of interpolation techniques. There are various methods of interpolation. These methods are the following:

* Linear
* Nearest
* Zero
* S-linear
* Quadratic
* Cubic

### Splines

The spline is a flexible strip used to generate a smooth curve through a designated set of points. Spline interpolation requires two essential steps. In the first step, a spline representation of the curve is computed, and in the second step, the spline is evaluated at the desired points.



There are two methods to represent the curve. First, the direct spline method is used to find the spline representation of a curve in a two- dimensional plane using the function **splrep()**. default spline order is cubic, but it can be changed with the input keyword, k.

**splprep()** allows us to define the curve **parametrically** in N-dimension space. keyword argument is used to specify the amount of smoothing to perform during the spline fit.

The standard output is a 3-tuple, (t,c,k), where, t represents the knot-point, c represents coefficient and k represents the order of the spline.

### Univariate Spline

The **scipy.interpolate** provides **UnivariateSpline** class, a suitable method to create a function, based on fixed data points. The syntax is as following:

**Syntax:-** scipy.interpolate.UnivariateSpline(x, y, w = None, bbox = [None, None], k = 3, s = None, ext = 0, check\_finite = False).

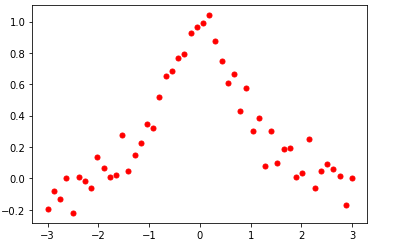
**w-** It specifies the weight for spline fitting. It must be positive.

**s-** It specifies the number of knots(points) by specifying a smoothing condition.

**k-** It denotes the degree of the smoothing spline. Must be k<= 5. By default, k = 3, a cubic spline.

* **Ext** - It controls the extrapolation mode for elements not in the interval defined by the knot sequence.
  + if ext = 0 or 'extrapolate', returns the extrapolated value.
  + if ext = 1 or 'zero', returns 0
  + if ext = 2 or 'raise', raises a ValueError
  + if ext = 3 of 'const', returns the boundary value.
* **check\_finite** -To check whether the input arrays contain only finite numbers.

1. from scipy.interpolate **import** UnivariateSpline
2. x = np.linspace(-3, 3,50)
3. y = np.exp(-x\*\*2) + 0.1 \* np.random.randn(50)
4. plt.plot(x, y, 'ro', ms = 5)
5. plt.show()



# SciPy Input and Output

The SciPy.io (input and output) package supports a vast range of functions to work with a different format of files. Few of these formats are the following:

* Matlab
* IDL
* Matrix Market
* Wave
* Arff
* Netcdf etc.

1. **import** scipy.io as sio
2. vect = np.arange(10) #Save a mat file
3. sio.savemat('array.mat', {'vect':vect})
4. mat\_file\_content = sio.loadmat('array.mat') #Now Load the File
5. print(mat\_file\_content) // {'\_\_header\_\_': b'MATLAB 5.0 MAT-file Platform: nt, Created on: Wed Nov 13 14:38:57 2019', '\_\_version\_\_': '1.0', '\_\_globals\_\_': [], 'vect': array([[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]])}

We can see the array consisted with the information. If we want to inspect the contents of a MATLAB file without reading the data into memory, use **the whosmat** command as given below:

1. **import** scipy.io as sio
2. mat\_file\_content = sio.whosmat('array.mat')
3. print(mat\_file\_content) //[('vect', (1, 10), 'int64')]

# SciPy Linear Algebra

SciPy is built upon the **ATLAS LAPACK** and **BLAS** library, and it provides very fast linear algebra capabilities. Linear algebra routine accepts two-dimension array object and output is also given as a two-dimension array. If we want more speed in computation, then we have to dig deep in this scenario.

**Syntax:-** linalg.solve()

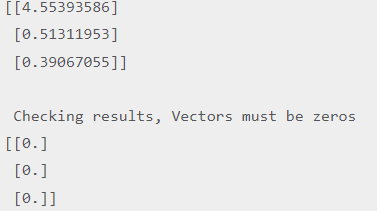
### Linear Equation

**linalg.solve** is used to solve linear equation a\*x + b\*y = Z, for the unknown x, y values.

x + 3y +10z = 10  
2x + 12y + 7z = 18  
5x + 8y + 8z = 30

Here we will solve above linear equation by using the **linear.solve** command for the faster calculation.

1. from scipy **import** linalg
2. # We are trying to solve a linear algebra system which can be given as
3. # x + 3y +10z = 10 , 2x + 12y + 7z = 18 , 5x + 8y + 8z = 30
4. a = np.array([[1, 3, 10], [2, 12, 7], [5, 8, 8]]) # Creating input array
5. b = np.array([[10], [18], [30]]) # Solution Array
6. x = linalg.solve(a, b) # Solve the linear algebra
7. print(x) # Print results
8. print("\n Checking results, Vectors must be zeros") # Checking Results
9. print(a.dot(x) - b)



In above program, we have declared a and b as variable where a stored coefficients of equation and b stored right-hand-side value. Variable x stored the evaluated solution.

### Finding the determinants

The determinant of the square matrix is found by using the **linalg.det()** function. The determinate A is often denoted as |A| in the linear algebra. It accepts a matrix and returns a scalar value.

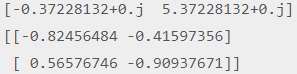
1. from scipy **import** linalg
2. A = np.array([[5,9],[8,4]]) //#Declaring the numpy array
3. x = linalg.det(A) #Passing the values to the det function
4. print(x) // -52

### Eigenvalues and Eigenvectors

Finding eigenvalues and eigenvector problems are the most common problem in linear algebra. We can find the Eigenvalues (?) and the corresponding Eigenvectors (v) of a square matrix (A) by **linalg.eig()** function.

**Av = λv**

1. from scipy **import** linalg
2. a = np.array([[3,2],[4,6]]) #Declaring the numpy array
3. l, v = linalg.eig(a) #Passing the values to the eig function
4. print(l) #printing the result **for** eigenvalues
5. print(v) #printing the result **for** eigenvectors



### SciPy svd

The svd is stands for **single value decomposition**. The unique value decomposition of a matrix A is the factorization of A into the product of three matrices A = UDVT, where the columns of U & V are orthonormal, & the matrix D is diagonal with real positive entries.

# SciPy Ndimage

The **SciPy** provides the **ndimage** (n-dimensional image) package, that contains the number of general image processing and analysis functions. It is dedicated to image processing. We can perform several tasks in image processing such as input/output image, classification, Feature extraction, Registration, etc.

**Opening and Writing to Image Files**

The **scipy.ndimage** provides the **misc** package, which comes with some images. We will use those images and perform image manipulation.

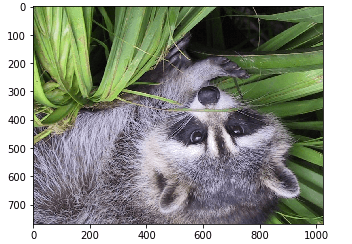
1. from scipy **import** misc
2. f = misc.face()
3. misc.imsave('face.jpg', f)
4. plt.imshow(f)
5. plt.show()



The number in the matrix format represents any images and their color combinations. A machine uses those numbers for the manipulation. There is two way to represent the image, grayscale and RGB. RGB is the most popular way of representation.

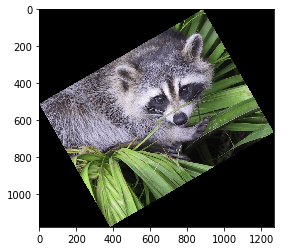
We can perform the some basic operations such as image rotation, image up-side down. Consider the following example of image upside down:

1. from scipy **import** misc
2. face = misc.face()
3. flip\_ud\_face = np.flipud(face)
4. plt.imshow(flip\_ud\_face)
5. plt.show()



The SciPy provide the **rotate()** function, which rotates the image to the specified angle.

1. from scipy **import** misc,ndimage
2. face = misc.face()
3. rotate\_face = ndimage.rotate(face, 30) #rotating the image 30 degree
4. plt.imshow(rotate\_face)
5. plt.show()



## Filters

Filtering is the process where we modify and enhance an image. For example, the filter can be applied to an image to highlight certain feature or eliminate other features. Image processing operations implemented with filtering including **Smoothing and Edge Enhancement**. Consider the following operations using SciPy ndimage.

* **Blurring**

Blurring is the technique that is used to reduce the noise in the image. We can perform a filter operation and observe the change in the image.

1. from scipy **import** misc
2. from scipy **import** ndimage
3. face = misc.face()
4. blurred\_image = ndimage.gaussian\_filter(face, sigma=4)
5. plt.imshow(blurred\_image )
6. plt.show()

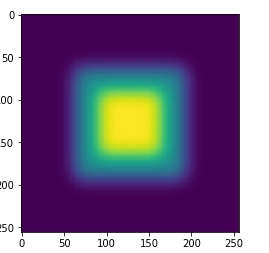


The sigma value denotes the level of blur on a scale of five. You can change the sigma value and see the difference.

* **Edge Detection**

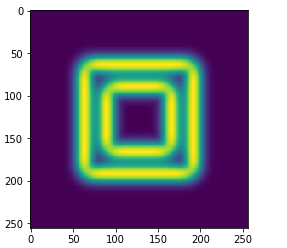
Edge detection is an image processing term which is used for finding the boundaries of objects within the image. It is used for image segmentation and data extraction in fields such as Image Processing, Computer Vision, and Machine Vision. To understand the edge detection more specifically, visit our tutorial - [click here](https://www.javatpoint.com/opencv#Edge-detection).

1. **import** scipy.ndimage as nd
2. im = np.zeros((256, 256))
3. im[64:-64, 64:-64] = 1
4. im[90:-90,90:-90] = 2
5. im = ndimage.gaussian\_filter(im, 10)
6. plt.imshow(im)
7. plt.show()



The output image appears like a square block of color. Now, we will find the edges of those colored block. The **ndimage** provides the **sobel()** function to perform this operation. Whereas, NumPy provides the **hypot()** function which is used to combine the two resultant matrices to one. Consider the following example:

1. **import** scipy.ndimage as nd
2. im = np.zeros((256, 256))
3. im[64:-64, 64:-64] = 1
4. im[90:-90,90:-90] = 2
5. im = ndimage.gaussian\_filter(im, 8)
6. zx = ndimage.sobel(im, axis = 0, mode = 'constant')
7. zy = ndimage.sobel(im, axis = 1, mode = 'constant')
8. sobl = np.hypot(zx, zy)
9. plt.imshow(sobl)
10. plt.show()



# SciPy Optimize

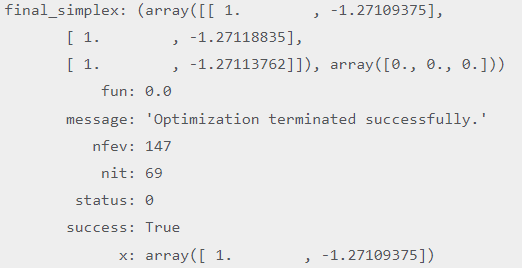
The optimize package provides various commonly used optimization algorithms. This module contains the following aspects:

* Global optimization routines(brute-force, anneal(), basinhopping())
* Unconstrained and constrained minimization of the multivariate scalar functions(minimize()) using various algorithms (BFGS, Nelders-Mead simplex, Newton Conjugate Gradient, COBLYA).
* Least-squares minimization algorithms(leastsq() and curve fit())
* Scalar univariate function minimizers (minimizer\_scalar() &root finders newton())

### Nelder- Mead Simplex Algorithm

The **Nelder- Mead Simplex** algorithm provides **minimize()** function which is used for minimization of scalar function of one or more variables.

1. **import** scipy
2. from scipy.optimize **import** minimize
3. def f(x): #define function f(x)
4. **return** .2\*(1 - x[0])\*\*2
5. scipy.optimize.minimize(f, [2, -1], method="Nelder-Mead")

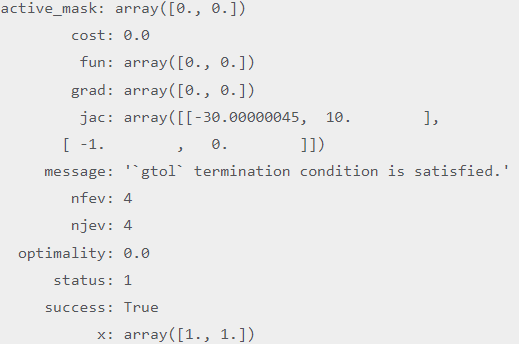


### Least Square Minimization

It is used to solve the nonlinear least-square problems with bound on the variables. Given the residuals (**difference between observed and predicted value of data**) f(x) (an n-dimension real function of n real variables) and the loss function rho(s) (a scalar

function), **least\_square** finds a local minimum of the cost function f(x):

1. from scipy.optimize **import** least\_squares
2. input = np.array([2, 2])
3. def rosenbrock(x):
4. **return** np.array([10 \* (x[1] - x[0]\*\*3), (1 - x[0])])
5. res = least\_squares(rosenbrock, input)



### Root Finding

* **Scalar functions**AD

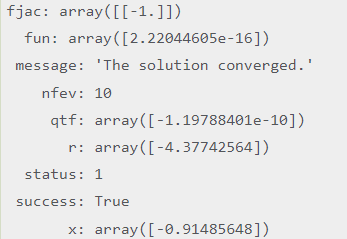
There are four different root-finding algorithms for a single value equation. Each algorithm needs the endpoints of an interval in which a root is expected (because the function changes signs).

* **Sets of Equations**

The **root()** function is used to find the root of the nonlinear equation. There are various methods such as **hybr** (the default) and the **Levenberg-Marquardt method** from the MINPACK. Let's consider the below equation

x2 + 3cos(x)=0

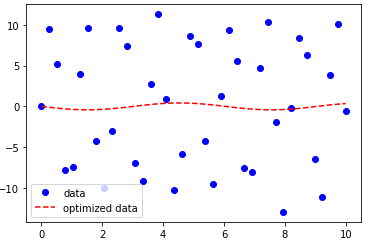
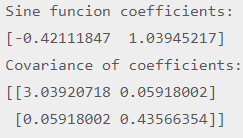
1. from scipy.optimize **import** root
2. def func(x):
3. **return** x\*2 + 3\* np.cos(x)
4. a = root(func, 0.3)



### Optimize Curve Fitting

Curve fitting is the technique of creating a curve. It is a mathematical function that has the best fit to a series of data points, possibly subject to constraints.

1. from scipy.optimize **import** curve\_fit
2. x = np.linspace(0, 10, num = 40)
3. # The coefficients are much bigger.
4. y = 10.35 \* np.sin(5.330 \* x) + np.random.normal(size = 40)
5. def test(x, a, b):
6. **return** a \* np.sin(b \* x)
7. param, param\_cov = curve\_fit(test, x, y)
8. print("Sine funcion coefficients:")
9. print(param)
10. print("Covariance of coefficients:")
11. print(param\_cov)
12. ans = (param[0]\*(np.sin(param[1]\*x)))
13. plt.plot(x, y, 'o', color ='red', label ="data")
14. plt.plot(x, ans, '--', color ='blue', label ="optimized data")
15. plt.legend()
16. plt.show()



### SciPy fsolve

**scipy.optimize** library provides the **fsolve()** function, which is used to find the root of the function. It returns roots of the equation defined by fun(x) = 0 given a starting estimate.

1. from scipy.optimize **import** fsolve
2. sqrt = np.emath.sqrt
3. a = 132712000000
4. T = 365.35 \* 86337 \* 2 / 3
5. e = 580.2392124070273
6. def f(x):
7. **return** np.abs((T \* a \*\* 2 / (2 \* np.pi)) \*\* (1 / 3) \* sqrt(1 - x \*\* 2)
8. - sqrt(.5 \* a \*\* 2 / e \* (1 - x \*\* 2)))
9. x = fsolve(f, 0.01)
10. x, f(x)



# SciPy Stats

The **scipy.stats** contains a large number of statistics, probability distributions functions. The list of statistics functions can be obtained by **info(stats)**. A list of a random variable can also be acquired from the **docstring** for the stat sub-package.

| **Function** | **Description** |
| --- | --- |
| rv\_continuos | It is a base class to construct specific distribution classes and instances for continuous random variable. |
| rv\_discrete | It is a base class to construct specific distribution classes and instances for discrete random variables. |
| rv\_histogram | It can be inherited from **rv\_continuous** class. It generates a distribution given by a histogram. |

### Normal Continuous Random Variable

There are two general distribution classes which have been implemented for encapsulating continuous random variables and discrete random variable. Here we will

discuss about the continuous Random Variables:

1. from scipy.stats **import** norm
2. print(norm.cdf(np.array([3,-1., 0, 1, 2, 4, -2, 5])))



In the above program, first, we need to import the **norm** module from the **scipy.stats**, then we passed the data as Numpy array in the cdf() function.

To get the median of the distribution, we can use the **Percent Point Function (PPF)**, this is the inverse of the CDF.

We can generate the sequence of the random numbers; the size argument is necessary to pass the size parameter.

1. from scipy.stats **import** norm
2. print(norm.rvs(size = 4)) //[-0.42700905 1.0110461 0.05316053 -0.45002771]

The output can vary when we run the program every time. We can use the **seed()** function to generate the same random numbers.

### Descriptive Statistics

The descriptive statistics describe the values of observation in a variable. There are various stats such as Min, Max, and Variance, that take the Numpy array as input and returns the particular results. Some essential functions provide by **scipy.stats** package are described in the following image.

| **Function** | **Description** |
| --- | --- |
| describe() | Computes various descriptive statistics of the input array. |
| gmean() | Computes geometric mean along with the specified. |
| hmean() | Calculates the harmonic mean along the specified axis. |
| kurtosis() | Computes the Kurtosis. |
| mode() | Returns the mode value. |
| skew() | Tests the skewness of the data |
| zscore() | It calculates the z score of each value in the sample, relative to the sample mean and standard deviation. |

1. **import** scipy as sp
2. from scipy.stats **import** norm
3. number\_of\_data = 100
4. random\_data\_set = sp.randn(number\_of\_data)
5. print(random\_data\_set.mean()) //0.006283818005153084
6. print(sp.median(random\_data\_set)) // -0.03008382588766136
7. min\_max = np.array([random\_data\_set.min(),random\_data\_set.max()])
8. print(min\_max) //[-2.1865825 2.47537921]

// sp.stats.describe(random\_data\_set) // DescribeResult(nobs=100, minmax= (-2.1865824992721987, 2.475379209985273), mean=0.006283818005153084, variance=1.0933102537156147, skewness=0.027561719919920322, kurtosis= -0.6958272633471831)

### T-Test

The t-test is used to compare two averages (means) and tells that if these averages are different from each other. The t-test is also described as significant in the differences between the groups.

### T-score

The t-score is a ratio between two groups and the difference within the groups. The smaller the t-score shows that the groups are relatively similar, and the more significant t-score indicates, the more difference between the groups.

**Comparing two samples**

The two samples are given that can come either from the same or from difference distributions and we want to test whether these samples have same statistical properties.

1. from scipy **import** stats
2. rvs = stats.norm.rvs(loc = 6, scale = 10, size = (50,2))
3. print(stats.ttest\_1samp(rvs,5.0)) Ttest\_1sampResult(statistic=array([0.42271098, 1.1463823 ]), pvalue=array([0.67435547, 0.25720448]))

In the above output, a p-value is a **probability** that the results from your sample data occurred by chance. P-values are from 0% to 100%.

### SciPy Linear Regression

Linear regression is used to find the relationship between the two variables. The SciPy provides **linregress()** function to perform linear regression. The syntax is given below:

**Syntax:** scipy.stats.linregress(x,y=None)

**x, y:** These two parameters should be an array and have the same length.

There are two types of linear regression.

* Simple regression
* Multivariable regression

**Simple Regression**

Simple linear regression is a method for predicting a response using a single feature. It is assumed that the two variables are linearly related, which means the other variable can accurately predict one variable. For example, using temperature in the degree Celsius, it is correctly predicted in Fahrenheit.

**Multivariable Regression**

Multiple linear regression is described as the relationship between one continuous dependent variable and two or more independent variables.

1)price(dependent variable)= m1\*area + m2\*bedrooms + m2\*age(independent variable)

The variable price is dependent on the other variables.

# SciPy Sparse Matrix

The sparse matrix allows the data structure to store large sparse matrices, and provide the functionality to perform complex matrix computations. In simple words, suppose you have a 2-D matrix with hundreds of elements, where only a few of them contain a non-zero value. When sorting this matrix using the sorting approach, we would waste a lot of space for zeros.

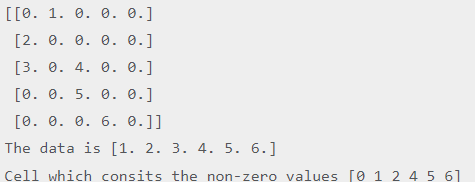
The sparse data structure allows us to store only non-zero values assuming the rest of them are zeros.

### Sparse matrix types in SciPy

There are various ways to represent a sparse matrix; SciPy provides seven of them.

* Block Sparse Row matrix (BSR)
* Coordinate list matrix (COO)
* Compressed Sparse Column matrix (CSC)
* Compressed Sparse Row matrix (CSR)
* Sparse matrix with Diagonal storage (DIA)
* Dictionary Of Keys based sparse matrix (DOK)
* Row-based linked list sparse matrix (LIL)

1. from scipy.sparse **import** random
2. np.random.seed(10)
3. # Generate a random binary sparse matrix
4. matrix = random(5, 5, format='csr', density=0.25)
5. # Substitute all non zero values with index number
6. matrix.data[:] = np.arange(1, matrix.data.shape[0]+1)
7. print(matrix.toarray()) # We can access and modify these arrays:
8. print("The data is",matrix.data)
9. print("Cell which consits the non-zero values",matrix.indptr)



# SciPy Spatial

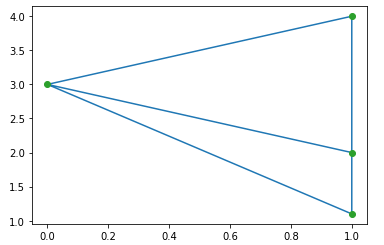
The **scipy.spatial** package can calculate Triangulation, Voronoi Diagram and Convex Hulls of a set of points, by leveraging the **Qhull** library. Likewise, it contains **KDTree** implementations for nearest-neighbor point queries and utilities for distance computations in various metrics.

* **Delaunay Triangulations**

In mathematics and computation geometry, The Delaunay Triangulation defines that the three points from triangle create a vertex when these vertex touch the path of the circle.

The center of the circle is determined by the radius of the three points or the triangle.

1. from scipy.spatial **import** Delaunay
2. **import** matplotlib.pyplot as plt
3. points = np.array([[0, 3], [1, 1.1], [1, 4], [1, 2]])
4. triang = Delaunay(points)
5. plt.triplot(points[:,0], points[:,1], triang.simplices.copy())
6. plt.plot(points[:,0], points[:,1], 'o')
7. plt.show()



### Coplaner Points

Coplaner points are three or more points exist in the same plane. The plane is a flat surface, which can expand without end in all directions.

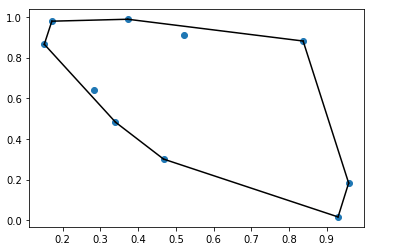
1. from scipy.spatial **import** Delaunay
2. points = np.array([[0, 0], [0, 1], [1, 0], [1, 1],[1,1]])
3. tri = Delaunay(points)
4. print(tri.coplanar) //[[4 0 3]]

In above O/P, point 4 is not included in triangulation; it exists near triangle 0 & vertex 3.

### Convex hulls

In mathematics, the convex hull or convex envelope of a set of points X in **Euclidean plan** or in a **Euclidean space**. It is the smallest convex set that contains X.

1. from scipy.spatial **import** ConvexHull
2. **import** matplotlib.pyplot as plt
3. points = np.random.rand(10, 2) # 30 random points in 2-D
4. hull = ConvexHull(points)
5. plt.plot(points[:,0], points[:,1], 'o')
6. **for** simplex in hull.simplices:
7. plt.plot(points[simplex,0], points[simplex,1], 'k-')
8. plt.show()



### Scipy cdist

The SciPy provides the **spatial.distance.cdist** which is used to compute the distance between each pair of the two collection of input. following are the calling conventions:

**1. Y = cdist(XA, XB, 'euclidean')**

It calculates the distance between m points using Euclidean distance (2-norm) as the distance metric between the points. The points are organized as m n-dimensional row vectors in the matrix X.

**2. Y = cdist(XA, XB, 'minkowski', p=2.)**

It calculates the distances using the Minkowski distance ||u?v||p (p-norm) where p?1.

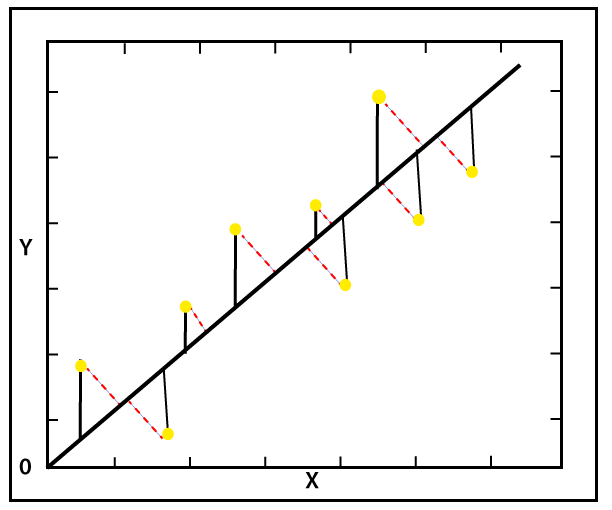
**3. Y = cdist(XA, XB, 'cityblock')**

It calculates the city block or Manhattan distance between the points.

# SciPy ODR

The ODR is an abbreviation form of **Orthogonal Distance Regression**. It is used in the regression studies. basic linear regression is used to estimate the relationship between two variables y and x by drawing the line of the best fit in the graph. Then the question arises why Orthogonal Distance Regression (ODR) needs? Sometimes measurement error occurred in the independent variable (x), not in the dependent variable (y).

The standard linear regression is focused on predict the Y value from the X value, so the useful thing to do is to calculate the error in the Y values (as we shown by dotted black lines in the below image). However, it is better to be account for the error in both X and Y (as shown by the dotted red lines in the following image).



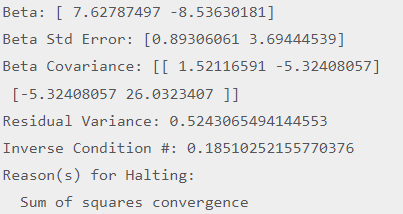
Orthogonal Distance Regression (ODR) is a method that is used to calculate the error perpendicular to the line rather than vertically.

Orthogonal Distance Regression provides ODRPACK to perform ODR with non-linear functions. It is basically a FORTRAN-77 library. It can do explicit or implicit ODR fits. It can also be used to solve the ordinary least square problem (OLS).

**Implementation of scipy.odr for Univariate Regression**

The Univariate regression can be defined as determining relationship between one independent variable and one dependent variable.

1. from scipy.odr **import** \*
2. **import** random
3. # Initiate some data, and generate the random number using random.random().
4. a = np.array([0, 1, 2, 3, 4, 5, 6, 7])
5. b = np.array([i\*\*2 + random.random() **for** i in x])
6. def linear\_func(z, a): # Define a quadratic function ( in **this** **case**) to fit data with.
7. m, p = z
8. **return** m\*a + p
9. linear\_model\_fit = Model(linear\_func) # Creating a model **for** fitting.
10. data=RealData(x, y) #Creating a RealData obj using our initiated data from above.
11. odr = ODR(data, linear\_model\_fit, beta0=[0., 1.])#Fixed up ODR with model & data.
12. out = odr.run() # Here we run the regression using the run().
13. out.pprint() # Use the in-built pprint method to give us results.



# SciPy Spectrogram

The signal processing toolbox consists of few filtering functions, a limited set of filter design tools. It also contains few B-spline interpolation algorithm for one-and two-dimensional data.

The **scipy.signal.spectrogram** computes a spectrogram with the consecutive Fourier transforms. Spectrograms can be used as a way of reflecting the change of frequency content of non-signal signals over time.

1. from scipy **import** signal
2. #Generate a test signal, a 2 Vrms sine wave whose frequency linearly changes with time from 1kHz to 2kHz, corrupted by 0.001 V\*\*2/Hz of white noise sampled at 10 kHz.
3. fs = 10e3 # Sampling Frequency
4. N = 1e5
5. amp = 2 \* np.sqrt(2)
6. noise\_power = 0.001 \* fs / 2
7. time = np.arange(N) / fs
8. freq = np.linspace(1e3, 2e3, N)
9. x = amp \* np.sin(2\*np.pi\*freq\*time)
10. x += np.random.normal(scale=np.sqrt(noise\_power), size=time.shape)
11. f, t, Sxx = signal.spectrogram(x, fs) # Compute and plot the spectrogram.
12. plt.pcolormesh(t, f, Sxx)
13. plt.ylabel('Frequency [Hz]')
14. plt.xlabel('Time [sec]')
15. plt.show()

The scipy.signal.spectogram() returns the following array:

**f: ndarray:** Array of sample frequencies.

**t: ndarray:** Array of segment times.

**Sxx: ndarry:** Spectogram of x. By default the last axis of **Sxx** corresponds to the segment times