SciPy - Guia rápido

SciPy - Introdução

SciPy, pronunciado como Sigh Pi, é um python científico de código aberto, distribuído sob a biblioteca licenciada BSD para realizar cálculos matemáticos, científicos e de engenharia.

A biblioteca SciPy depende do NumPy, que fornece manipulação de array N-dimensional conveniente e rápida. A biblioteca SciPy foi construída para funcionar com arrays NumPy e fornece muitas práticas numéricas eficientes e fáceis de usar, como rotinas para integração e otimização numérica. Juntos, eles funcionam em todos os sistemas operacionais populares, são rápidos de instalar e gratuitos. NumPy e SciPy são fáceis de usar, mas poderosos o suficiente para serem usados por alguns dos principais cientistas e engenheiros do mundo.

Subpacotes SciPy

O SciPy está organizado em subpacotes que cobrem diferentes domínios da computação científica. Eles estão resumidos na tabela a seguir -

scipy.cluster	Quantização vetorial / Kmeans
scipy.constantes	Constantes físicas e matemáticas
scipy.fftpack	transformada de Fourier
scipy.integrate	Rotinas de integração
scipy.interpolar	Interpolação
scipy.io	Entrada e saída de dados
scipy.linalg	Rotinas de álgebra linear
scipy.ndimage	pacote de imagem n-dimensional
scipy.odr	Regressão de distância ortogonal
scipy.optimize	Otimização
scipy.signal	Processamento de sinal
scipy.esparso	Matrizes esparsas
scipy.espacial	Estruturas e algoritmos de dados espaciais

scipy.especial	Quaisquer funções matemáticas especiais
scipy.stats	Estatisticas

Estrutura de dados

A estrutura de dados básica usada pelo SciPy é um array multidimensional fornecido pelo módulo NumPy. NumPy fornece algumas funções para Álgebra Linear, Transformadas de Fourier e Geração de Números Aleatórios, mas não com a generalidade das funções equivalentes no SciPy.

SciPy - Configuração do ambiente

A distribuição padrão do Python não vem com nenhum módulo SciPy. Uma alternativa leve é instalar o SciPy usando o popular instalador de pacotes Python,

```
pip install pandas
```

Se instalarmos o **pacote Anaconda Python** , o Pandas será instalado por padrão. A seguir estão os pacotes e links para instalá-los em diferentes sistemas operacionais.

janelas

Anaconda (de https://www.continuum.io) é uma distribuição Python gratuita para a pilha SciPy. Também está disponível para Linux e Mac.

Canopy (https://www.enthought.com/products/canopy/) está disponível gratuitamente, bem como para distribuição comercial com uma pilha SciPy completa para Windows, Linux e Mac.

Python (x,y) - É uma distribuição Python gratuita com pilha SciPy e Spyder IDE para sistema operacional Windows. (Pode ser baixado em https://python-xy.github.io/)

Linux

Os gerenciadores de pacotes das respectivas distribuições Linux são usados para instalar um ou mais pacotes na pilha SciPy.

Ubuntu

Podemos usar o seguinte caminho para instalar o Python no Ubuntu.

```
sudo apt-get install python-numpy python-scipy
python-matplotlibipythonipython-notebook python-pandas python-sympy python-nose
```

Fedora

Podemos usar o seguinte caminho para instalar o Python no Fedora.

```
sudo yum install numpyscipy python-matplotlibipython python-pandas
sympy python-nose atlas-devel
```

SciPy - Funcionalidade Básica

Por padrão, todas as funções NumPy estão disponíveis através do namespace SciPy. Não há necessidade de importar explicitamente as funções do NumPy quando o SciPy é importado. O objeto principal do NumPy é o array multidimensional homogêneo. É uma tabela de elementos (geralmente números), todos do mesmo tipo, indexados por uma tupla de inteiros positivos. No NumPy, as dimensões são chamadas de eixos. O número de **eixos** é chamado de **classificação** .

Agora, vamos revisar a funcionalidade básica de vetores e matrizes no NumPy. Como o SciPy é construído sobre arrays NumPy, é necessário compreender os fundamentos do NumPy. Como a maior parte da álgebra linear trata apenas de matrizes.

Vetor NumPy

Um vetor pode ser criado de várias maneiras. Alguns deles são descritos abaixo.

Convertendo objetos semelhantes a array Python em NumPy

Consideremos o seguinte exemplo.

```
import numpy as np
list = [1,2,3,4]
arr = np.array(list)
print arr
```

A saída do programa acima será a seguinte.

```
[1 2 3 4]
```


Criação intrínseca de array NumPy

NumPy possui funções integradas para criar arrays do zero. Algumas dessas funções são explicadas abaixo.

Usando zeros()

A função zeros(shape) criará um array preenchido com valores 0 com a forma especificada. O tipo padrão é float64. Consideremos o seguinte exemplo.

```
import numpy as np
print np.zeros((2, 3))
```

A saída do programa acima será a seguinte.

```
array([[ 0., 0., 0.],
[ 0., 0., 0.]])
```

Usando uns()

A função ones(shape) criará um array preenchido com valores 1. É idêntico a zeros em todos os outros aspectos. Consideremos o seguinte exemplo.

```
import numpy as np
print np.ones((2, 3))
```

A saída do programa acima será a seguinte.

```
array([[ 1., 1., 1.],
[ 1., 1., 1.]])
```

Usando intervalo()

A função arange() criará arrays com valores incrementais regularmente. Consideremos o seguinte exemplo.

```
import numpy as np
print np.arange(7)
```

O programa acima irá gerar a seguinte saída.

```
array([0, 1, 2, 3, 4, 5, 6])
```

Definindo o tipo de dados dos valores

Consideremos o seguinte exemplo.

```
import numpy as np
arr = np.arange(2, 10, dtype = np.float)
print arr
print "Array Data Type :",arr.dtype
```

O programa acima irá gerar a seguinte saída.

```
[ 2. 3. 4. 5. 6. 7. 8. 9.]
Array Data Type : float64
```

Usando linspace()

A função linspace() criará arrays com um número especificado de elementos, que serão espaçados igualmente entre os valores inicial e final especificados. Consideremos o seguinte exemplo.

```
import numpy as np
print np.linspace(1., 4., 6)
```

O programa acima irá gerar a seguinte saída.

```
array([ 1. , 1.6, 2.2, 2.8, 3.4, 4. ])
```


Matriz

Uma matriz é uma matriz 2D especializada que mantém sua natureza 2D por meio de operações. Possui certos operadores especiais, como * (multiplicação de matrizes) e ** (potência de matrizes). Consideremos o seguinte exemplo.

```
import numpy as np
print np.matrix('1 2; 3 4')
```

O programa acima irá gerar a seguinte saída.

```
matrix([[1, 2],
```

```
[3, 4]])
```

Transposição Conjugada de Matriz

Este recurso retorna a transposta conjugada (complexa) de **self** . Consideremos o seguinte exemplo.

```
import numpy as np
mat = np.matrix('1 2; 3 4')
print mat.H
```

O programa acima irá gerar a seguinte saída.

```
matrix([[1, 3],
     [2, 4]])
```

Transposição de Matriz

Este recurso retorna a transposição de si mesmo. Consideremos o seguinte exemplo.

```
import numpy as np
mat = np.matrix('1 2; 3 4')
mat.T
```

O programa acima irá gerar a seguinte saída.

```
matrix([[1, 3],
     [2, 4]])
```

Quando transpomos uma matriz, criamos uma nova matriz cujas linhas são as colunas da original. Uma transposição conjugada, por outro lado, troca o índice da linha e da coluna para cada elemento da matriz. O inverso de uma matriz é uma matriz que, se multiplicada pela matriz original, resulta em uma matriz identidade.

SciPy - Cluster

O agrupamento K-means é um método para encontrar clusters e centros de cluster em um conjunto de dados não rotulados. Intuitivamente, podemos pensar em um cluster como – composto por um grupo de pontos de dados, cujas distâncias entre pontos são pequenas em comparação com as distâncias a pontos fora do cluster. Dado um conjunto inicial de K centros, o algoritmo K-means itera as duas etapas a seguir -

- Para cada centro, é identificado o subconjunto de pontos de treinamento (seu cluster) que está mais próximo dele do que qualquer outro centro.
- A média de cada recurso para os pontos de dados em cada cluster é calculada, e esse vetor médio se torna o novo centro desse cluster.

Estas duas etapas são iteradas até que os centros não se movam mais ou as atribuições não mudem mais. Então, um novo ponto **x** pode ser atribuído ao cluster do protótipo mais próximo. A biblioteca SciPy fornece uma boa implementação do algoritmo K-Means por meio do pacote cluster. Vamos entender como usá-lo.

Implementação K-Means no SciPy

Vamos entender como implementar K-Means no SciPy.

Importar K-médias

Veremos a implementação e uso de cada função importada.

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

Geração de dados

Temos que simular alguns dados para explorar o agrupamento.

```
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)))
```

Agora, temos que verificar os dados. O programa acima irá gerar a seguinte saída.

```
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 um grupo de observações por recurso. Antes de executar K-Means, é benéfico redimensionar cada dimensão de recurso do conjunto de observação com branqueamento.

Cada característica é dividida por seu desvio padrão em todas as observações para fornecer variância unitária.

Branquear os dados

Temos que usar o código a seguir para branquear os dados.

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

Calcular K-Means com três clusters

Vamos agora calcular K-Means com três clusters usando o código a seguir.

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

O código acima executa K-Means em um conjunto de vetores de observação formando K clusters. O algoritmo K-Means ajusta os centróides até que não seja possível fazer progresso suficiente, ou seja, a mudança na distorção, uma vez que a última iteração é inferior a algum limite. Aqui, podemos observar o centróide do cluster imprimindo a variável centróides usando o código fornecido a seguir.

```
print(centroids)
```

O código acima irá gerar a seguinte saída.

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

Atribua cada valor a um cluster usando o código fornecido a seguir.

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

A função **vq** compara cada vetor de observação na matriz 'M' por 'N' **obs** com os centróides e atribui a observação ao cluster mais próximo. Ele retorna o cluster de cada observação e a distorção. Podemos verificar a distorção também. Vamos verificar o cluster de cada observação usando o código a seguir.

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

O código acima irá gerar a seguinte saída.

```
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, 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, 0, 2, 2, 1, 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, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2], dtype=int32)
```

Os valores distintos 0, 1, 2 da matriz acima indicam os clusters.

SciPy - Constantes

O pacote de constantes SciPy fornece uma ampla gama de constantes, que são utilizadas na área científica geral.

Pacote de constantes SciPy

O **pacote scipy.constants** fornece várias constantes. Temos que importar a constante necessária e usá-la conforme o requisito. Vamos ver como essas variáveis constantes são importadas e usadas.

Para começar, vamos comparar o valor 'pi' considerando o exemplo a seguir.

```
#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)
```

O programa acima irá gerar a seguinte saída.

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

Lista de constantes disponíveis

As tabelas a seguir descrevem resumidamente as diversas constantes.

Constantes Matemáticas

Sr. Não.	Constante	Descrição
1	pi	pi
2	dourado	Proporção áurea

Constantes Físicas

A tabela a seguir lista as constantes físicas mais comumente usadas.

Sr. Não.	Constante e Descrição
1	**c** Velocidade da luz no vácuo
2	**velocidade da luz** Velocidade da luz no vácuo
3	**h** Constante de Planck
4	**Prancha** Constante de Planck h
5	**G** Constante gravitacional de Newton
6	**e** Carga elementar
7	**R** Constante molar do gás
8	**Avogrado** Constante de Avogrado
9	**k** Constante de Boltzmann
10	**massa_elétron(OR) m_e** Massa eletrônica
11	**massa_próton (OU) m_p** Massa de prótons
12	**massa_nêutron(OR)m_n** Massa de nêutrons

Unidades

A tabela a seguir contém a lista de unidades SI.

Sr. Não.	Unidade	Valor
1	mili	0,001
2	micro	1e-06
3	quilo	1000

Essas unidades variam de yotta, zetta, exa, peta, tera…quilo, hector,…nano, pico,… a zepto.

Outras constantes importantes

A tabela a seguir lista outras constantes importantes usadas no SciPy.

Sr. Não.	Unidade	Valor
1	grama	0,001kg
2	massa atômica	Constante de massa atômica
3	grau	Grau em radianos
4	minuto	Um minuto em segundos
5	dia	Um dia em segundos
6	polegada	Uma polegada em metros
7	mícron	Um mícron em metros
8	ano luz	Um ano-luz em metros
9	caixa eletrônico	Atmosfera padrão em pascais
10	Acre	Um acre em metros quadrados
11	litro	Um litro em metros cúbicos
12	galão	Um galão em metros cúbicos
13	km/h	Quilômetros por hora em metros por segundos
14	grau_Fahrenheit	Um Fahrenheit em Kelvins
15	eV	Um elétron-volt em joules

16	HP	Um cavalo-vapor em watts
17	dinâmico	Um dine em newtons
18	lambda2nu	Converter comprimento de onda em frequência óptica

Lembrar que tudo isso é um pouco difícil. A maneira mais fácil de obter qual chave é para qual função é com o método **scipy.constants.find()** . Consideremos o seguinte exemplo.

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

O programa acima irá gerar a seguinte saída.

```
[
  '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'
]
```

Este método retorna a lista de chaves, caso contrário, nada se a palavra-chave não corresponder.

SciPy - FFTpack

A Transformada de Fourier é calculada em um sinal no domínio do tempo para verificar seu comportamento no domínio da frequência. A transformação de Fourier encontra sua aplicação em disciplinas como processamento de sinal e ruído, processamento de imagem, processamento de sinal de áudio, etc. SciPy oferece o módulo fftpack, que permite ao usuário calcular transformadas rápidas de Fourier.

A seguir está um exemplo de função seno, que será usada para calcular a transformada de Fourier usando o módulo fftpack.

Transformação rápida de Fourier

Vamos entender em detalhes o que é a transformada rápida de Fourier.

Transformada Discreta de Fourier Unidimensional

A FFT y[k] de comprimento N da sequência de comprimento N x[n] é calculada por fft() e a transformada inversa é calculada usando ifft(). Consideremos o seguinte exemplo

```
#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
```

O programa acima irá gerar a seguinte saída.

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

Vejamos outro exemplo

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

yinv = ifft(y)

print yinv
```

O programa acima irá gerar a seguinte saída.

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

O módulo **scipy.fftpack** permite calcular transformadas rápidas de Fourier. Como ilustração, um sinal de entrada (ruidoso) pode ter a seguinte aparência -

```
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
```

Estamos criando um sinal com intervalo de tempo de 0,02 segundos. A última instrução imprime o tamanho do sinal sig. A saída seria a seguinte -

```
1000
```

Não sabemos a frequência do sinal; conhecemos apenas o intervalo de tempo de amostragem do sinal sinal. Supõe-se que o sinal venha de uma função real, portanto a transformada de Fourier será simétrica. A função **scipy.fftpack.fftfreq()** irá gerar as frequências de amostragem e **scipy.fftpack.fft()** irá calcular a transformada rápida de Fourier.

Vamos entender isso com a ajuda de um exemplo.

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

O programa acima irá gerar a seguinte saída.

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

Transformada Discreta de Cosseno

Uma **Transformada Discreta de Cosseno (DCT)** expressa uma sequência finita de pontos de dados em termos de uma soma de funções cosseno oscilando em diferentes frequências. SciPy fornece um DCT com a função **dct** e um IDCT correspondente com a função **idct** . Consideremos o seguinte exemplo.

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

O programa acima irá gerar a seguinte saída.

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

A transformada discreta inversa de cosseno reconstrói uma sequência a partir de seus coeficientes de transformada discreta de cosseno (DCT). A função idct é o inverso da função dct. Vamos entender isso com o exemplo a seguir.

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

O programa acima irá gerar a seguinte saída.

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

SciPy - Integrar

Quando uma função não pode ser integrada analiticamente, ou é muito difícil de integrar analiticamente, geralmente recorre-se a métodos de integração numérica. O SciPy possui diversas rotinas para realizar integração numérica. A maioria deles é encontrada na mesma biblioteca **scipy.integrate** . A tabela a seguir lista algumas funções comumente usadas.

Sr. Não.	Descrição da função
1	**quádruplo** Integração única
2	**dblquad** Integração dupla
3	**tplquad** Integração tripla
4	**nquad** n-fold multiple integration
5	**fixed_quad** Gaussian quadrature, order n
6	**quadrature** Gaussian quadrature to tolerance
7	**romberg** Romberg integration
8	**trapz** Trapezoidal rule
9	**cumtrapz** Trapezoidal rule to cumulatively compute integral
10	**simps** Simpson’s rule
11	**romb** Romberg integration
12	**polyint** Analytical polynomial integration (NumPy)

13	**poly1d** Helper function for polyint (NumPy)

Single Integrals

The Quad function is the workhorse of SciPy's integration functions. Numerical integration is sometimes called **quadrature**, hence the name. It is normally the default choice for performing single integrals of a function f(x) over a given fixed range from a to b.

$$\int_{a}^{b} f(x)dx$$

The general form of quad is **scipy.integrate.quad(f, a, b)**, Where 'f' is the name of the function to be integrated. Whereas, 'a' and 'b' are the lower and upper limits, respectively. Let us see an example of the Gaussian function, integrated over a range of 0 and 1.

We first need to define the function → $f(x) = e^{-x^2}$, this can be done using a lambda expression and then call the quad method on that function.

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

The above program will generate the following output.

```
(0.7468241328124271, 8.291413475940725e-15)
```

The quad function returns the two values, in which the first number is the value of integral and the second value is the estimate of the absolute error in the value of integral.

Note – Since quad requires the function as the first argument, we cannot directly pass exp as the argument. The Quad function accepts positive and negative infinity as limits. The Quad function can integrate standard predefined NumPy functions of a single variable, such as exp, sin and cos.

Multiple Integrals

The mechanics for double and triple integration have been wrapped up into the functions **dblquad, tplquad** and **nquad**. These functions integrate four or six arguments, respectively. The limits of all inner integrals need to be defined as functions.

Double Integrals

The general form of **dblquad** is scipy.integrate.dblquad(func, a, b, gfun, hfun). Where, func is the name of the function to be integrated, 'a' and 'b' are the lower and upper limits of the x variable, respectively, while gfun and hfun are the names of the functions that define the lower and upper limits of the y variable.

As an example, let us perform the double integral method.

$$\int_{0}^{1/2} dy \int_{0}^{\sqrt{1-4y^2}} 16xy \:dx$$

We define the functions f, g, and h, using the lambda expressions. Note that even if g and h are constants, as they may be in many cases, they must be defined as functions, as we have done here for the lower limit.

```
import scipy.integrate
from numpy import exp
from math import sqrt
f = lambda x, y : 16*x*y
g = lambda x : 0
h = lambda y : sqrt(1-4*y**2)
i = scipy.integrate.dblquad(f, 0, 0.5, g, h)
print i
```

The above program will generate the following output.

```
(0.5, 1.7092350012594845e-14)
```

Além das rotinas descritas acima, scipy.integrate possui uma série de outras rotinas de integração, incluindo nquad, que realiza integração múltipla n-fold, bem como outras rotinas que implementam vários algoritmos de integração. No entanto, quad e dblquad atenderão à maioria das nossas necessidades de integração numérica.

SciPy - Interpolar

Neste capítulo, discutiremos como a interpolação ajuda no SciPy.

O que é interpolação?

Interpolação é o processo de encontrar um valor entre dois pontos em uma linha ou curva. Para nos ajudar a lembrar o que significa, devemos pensar na primeira parte da palavra, 'inter', como significando 'entrar', o que nos lembra de olhar 'para dentro' dos dados que tínhamos originalmente. Esta ferramenta, a interpolação, não é útil apenas em estatísticas, mas também em ciência, negócios ou quando há necessidade de prever valores que se enquadram em dois pontos de dados existentes.

Vamos criar alguns dados e ver como essa interpolação pode ser feita usando o pacote **scipy.interpolate .**

```
import numpy as np
from scipy import interpolate
import matplotlib.pyplot as plt
x = np.linspace(0, 4, 12)
y = np.cos(x**2/3+4)
print x,y
```

O programa acima irá gerar a seguinte saída.

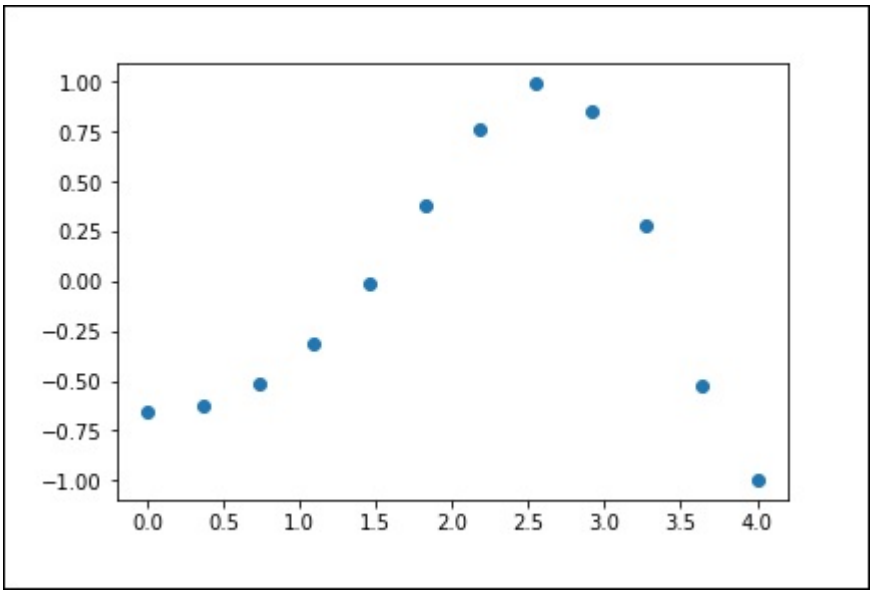
```
(
  array([0.,  0.36363636,  0.72727273,  1.09090909,  1.45454545, 1.81818182,
       2.18181818,  2.54545455,  2.90909091,  3.27272727,  3.63636364,  4.]),

  array([-0.65364362,  -0.61966189,  -0.51077021,  -0.31047698,  -0.00715476,
        0.37976236,   0.76715099,   0.99239518,   0.85886263,   0.27994201,
       -0.52586509,  -0.99582185])
)
```

Agora, temos duas matrizes. Assumindo essas duas matrizes como as duas dimensões dos pontos no espaço, vamos representar graficamente usando o programa a seguir e ver como elas se parecem.

```
plt.plot(x, y,'o')
plt.show()
```

O programa acima irá gerar a seguinte saída.



Interpolação 1-D

A classe interp1d em scipy.interpolate é um método conveniente para criar uma função baseada em pontos de dados fixos, que podem ser avaliados em qualquer lugar dentro do

domínio definido pelos dados fornecidos usando interpolação linear.

Usando os dados acima, vamos criar uma função de interpolação e desenhar um novo gráfico interpolado.

```
f1 = interp1d(x, y,kind = 'linear')

f2 = interp1d(x, y, kind = 'cubic')
```

Usando a função interp1d, criamos duas funções f1 e f2. Essas funções, para uma determinada entrada x, retornam y. A terceira variável kind representa o tipo da técnica de interpolação. 'Linear', 'Mais próximo', 'Zero', 'Slinear', 'Quadrático', 'Cúbico' são algumas técnicas de interpolação.

Agora, vamos criar uma nova entrada de maior comprimento para ver a clara diferença de interpolação. Usaremos a mesma função dos dados antigos nos novos dados.

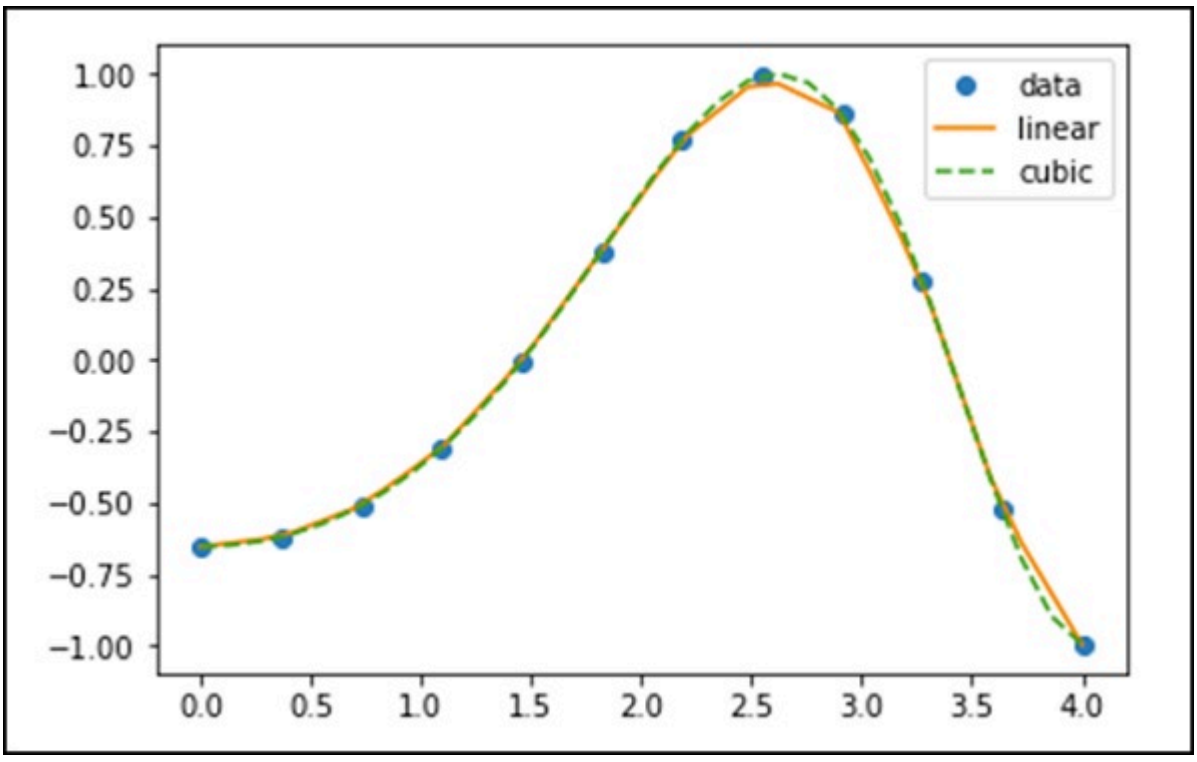
```
xnew = np.linspace(0, 4,30)

plt.plot(x, y, 'o', xnew, f(xnew), '-', xnew, f2(xnew), '--')

plt.legend(['data', 'linear', 'cubic','nearest'], loc = 'best')

plt.show()
```

O programa acima irá gerar a seguinte saída.



Splines

Para desenhar curvas suaves através de pontos de dados, os desenhistas costumavam usar tiras finas e flexíveis de madeira, borracha dura, metal ou plástico, chamadas estrias mecânicas. Para usar um spline mecânico, os pinos foram colocados em uma seleção criteriosa de pontos ao longo de uma curva em um projeto e, em seguida, o spline foi dobrado, de modo que tocasse cada um desses pinos.

Claramente, com esta construção, o spline interpola a curva nestes pinos. Pode ser utilizado para reproduzir a curva em outros desenhos. Os pontos onde os pinos estão localizados são chamados de nós. Podemos alterar a forma da curva definida pelo spline ajustando a localização dos nós.

Spline Univariada

O spline de suavização unidimensional ajusta-se a um determinado conjunto de pontos de dados. A classe UnivariateSpline em scipy.interpolate é um método conveniente para criar uma função, com base na classe de pontos de dados fixos – scipy.interpolate.UnivariateSpline(x, y, w = None, bbox = [None, None], k = 3, s = Nenhum, ramal = 0, check_finite = Falso).

Parameters - A seguir estão os parâmetros de um Spline Univariado.

- Isso ajusta uma spline y = spl(x) de grau k aos dados x, y fornecidos.
- 'w' - Especifica os pesos para ajuste do spline. Deve ser positivo. Se nenhum (padrão), os pesos serão todos iguais.
- 's' - Especifica o número de nós especificando uma condição de suavização.
- 'k' - Grau do spline de suavização. Deve ser <= 5. O padrão é k = 3, um spline cúbico.
- Ext - Controla o modo de extrapolação para elementos que não estão no intervalo definido pela sequência de nós.

  - se ext = 0 ou 'extrapolar', retorna o valor extrapolado.
  - se ext = 1 ou 'zero', retorna 0
  - se ext = 2 ou 'raise', gera um ValueError
  - se ext = 3 de 'const', retorna o valor limite.

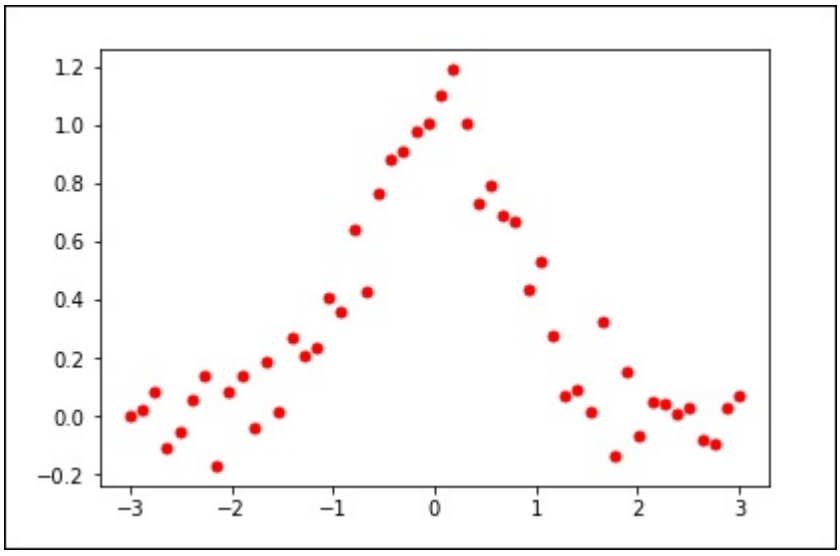
- check_finite – Se deve verificar se as matrizes de entrada contêm apenas números finitos.

Consideremos o seguinte exemplo.

```
import matplotlib.pyplot as plt
from scipy.interpolate import UnivariateSpline
x = np.linspace(-3, 3, 50)
y = np.exp(-x**2) + 0.1 * np.random.randn(50)
```

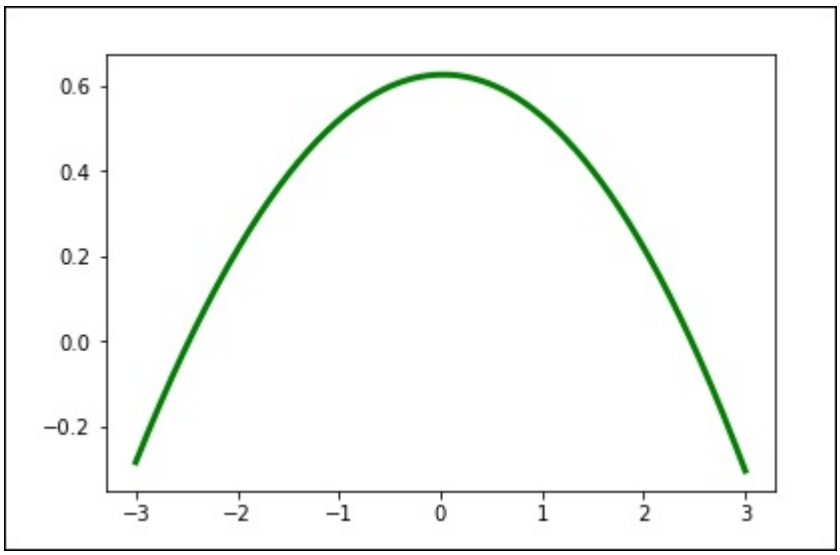
```
plt.plot(x, y, 'ro', ms = 5)
plt.show()
```

Use o valor padrão para o parâmetro de suavização.

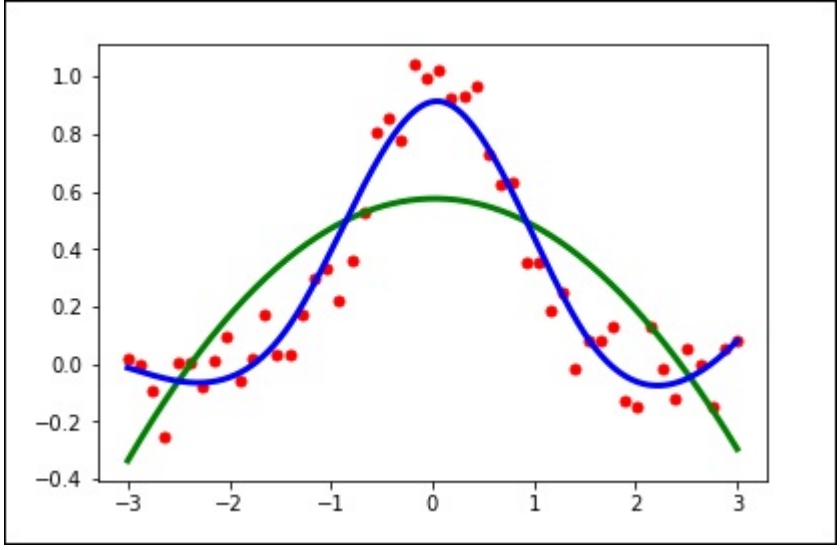



```
spl = UnivariateSpline(x, y)
xs = np.linspace(-3, 3, 1000)
plt.plot(xs, spl(xs), 'g', lw = 3)
plt.show()
```

Altere manualmente a quantidade de suavização.




```
spl.set_smoothing_factor(0.5)
plt.plot(xs, spl(xs), 'b', lw = 3)
plt.show()
```



SciPy - Entrada e Saída

O pacote Scipy.io (Input and Output) oferece uma ampla gama de funções para contornar diferentes formatos de arquivos. Alguns desses formatos são -

- Matlab
- IDL
- Mercado Matricial
- Aceno
- Arff
- Netcdf, etc.

Vamos discutir em detalhes os formatos de arquivo mais comumente usados -

MATLAB

A seguir estão as funções usadas para carregar e salvar um arquivo .mat.

Sr. Não.	Descrição da função
1	**tapete de carga** Carrega um arquivo MATLAB
2	**salvarmat** Salva um arquivo MATLAB
3	**quem é** Lista variáveis dentro de um arquivo MATLAB

Consideremos o seguinte exemplo.

```
import scipy.io as sio
import numpy as np

#Save a mat file
vect = np.arange(10)
sio.savemat('array.mat', {'vect':vect})

#Now Load the File
mat_file_content = sio.loadmat(‘array.mat’)
Print mat_file_content
```

O programa acima irá gerar a seguinte saída.

```
{
  'vect': array([[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]]), '__version__': '1.0',
  '__header__': 'MATLAB 5.0 MAT-file Platform: posix, Created on: Sat Sep 30
  09:49:32 2017', '__globals__': []
}
```

Podemos ver o array junto com as informações Meta. Se quisermos inspecionar o conteúdo de um arquivo MATLAB sem ler os dados na memória, use o **comando whosmat** conforme mostrado abaixo.

```
import scipy.io as sio
mat_file_content = sio.whosmat(‘array.mat’)
print mat_file_content
```

O programa acima irá gerar a seguinte saída.

```
[('vect', (1, 10), 'int64')]
```

SciPy - Linalg

SciPy é construído usando as bibliotecas otimizadas **ATLAS LAPACK** e **BLAS** . Possui recursos de álgebra linear muito rápidos. Todas essas rotinas de álgebra linear esperam um objeto que possa ser convertido em uma matriz bidimensional. A saída dessas rotinas também é um array bidimensional.

SciPy.linalg vs NumPy.linalg

Um scipy.linalg contém todas as funções que estão em numpy.linalg. Além disso, scipy.linalg também possui algumas outras funções avançadas que não estão em numpy.linalg. Outra vantagem de usar scipy.linalg sobre numpy.linalg é que ele é sempre compilado com suporte

BLAS/LAPACK, enquanto para NumPy isso é opcional. Portanto, a versão do SciPy pode ser mais rápida dependendo de como o NumPy foi instalado.

Equações lineares

O recurso **scipy.linalg.solve** resolve a equação linear a * x + b * y = Z, para os valores desconhecidos de x, y.

Como exemplo, suponha que se queira resolver as seguintes equações simultâneas.

x + 3y + 5z = 10

2x + 5y + z = 8

2x + 3y + 8z = 3

Para resolver a equação acima para os valores x, y, z, podemos encontrar o vetor solução usando uma matriz inversa conforme mostrado abaixo.

$$\begin{bmatrix} x\\ y\\ z \end{bmatrix} = \begin{bmatrix} 1 e 3 e 5\\ 2 e 5 e 1\\ 2 e 3 e 8 \end{bmatrix}^ {-1} \begin{bmatriz} 10\\ 8\\ 3 \end{bmatriz} = \frac{1}{25} \begin{bmatriz} -232\\ 129\\ 19 \end{bmatriz} = \ começar{bmatriz} -9,28\\ 5,16\\ 0,76 \end{bmatriz}.$$

No entanto, é melhor usar o comando **linalg.solve** , que pode ser mais rápido e numericamente mais estável.

A função resolver recebe duas entradas 'a' e 'b' nas quais 'a' representa os coeficientes e 'b' representa o respectivo valor do lado direito e retorna a matriz de solução.

Consideremos o seguinte exemplo.

```
#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
```

O programa acima irá gerar a seguinte saída.

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

Encontrando um Determinante

O determinante de uma matriz quadrada A é frequentemente denotado como |A| e é uma quantidade frequentemente usada em álgebra linear. No SciPy, isso é calculado usando a função **det()** . Ele pega uma matriz como entrada e retorna um valor escalar.

Consideremos o seguinte exemplo.

```
#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
```

O programa acima irá gerar a seguinte saída.

```
-2.0
```

Autovalores e autovetores

O problema de autovalor-autovetor é uma das operações de álgebra linear mais comumente empregadas. Podemos encontrar os valores Eigen (λ) e os vetores Eigen correspondentes (v) de uma matriz quadrada (A) considerando a seguinte relação -

$$\mathbf{Av = \lambda v}$$

scipy.linalg.eig calcula os autovalores de um problema de autovalor comum ou generalizado. Esta função retorna os valores Eigen e os vetores Eigen.

Consideremos o seguinte exemplo.

```
#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
```

O programa acima irá gerar a seguinte saída.

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

Decomposição de valor singular

Uma decomposição de valores singulares (SVD) pode ser pensada como uma extensão do problema de autovalores para matrizes que não são quadradas.

O **scipy.linalg.svd** fatora a matriz 'a' em duas matrizes unitárias 'U' e 'Vh' e uma matriz 1-D 's' de valores singulares (reais, não negativos) de modo que a == U*S *Vh, onde 'S' é uma matriz de zeros de formato adequado com a diagonal principal 's'.

Consideremos o seguinte exemplo.

```
#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
```

O programa acima irá gerar a seguinte saída.

```
(
  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]
  ])
)
```

SciPy - Ndimage

O submódulo SciPy ndimage é dedicado ao processamento de imagens. Aqui, ndimage significa uma imagem n-dimensional.

Algumas das tarefas mais comuns no processamento de imagens são as seguintes &miuns;

- Entrada/Saída, exibindo imagens
- Manipulações básicas - Cortar, inverter, girar, etc.
- Filtragem de imagem - Eliminação de ruído, nitidez, etc.
- Segmentação de imagem - Rotulagem de pixels correspondentes a diferentes objetos
- Classificação
- Extração de recursos
- Cadastro

Vamos discutir como alguns deles podem ser alcançados usando o SciPy.

Abrindo e gravando em arquivos de imagem

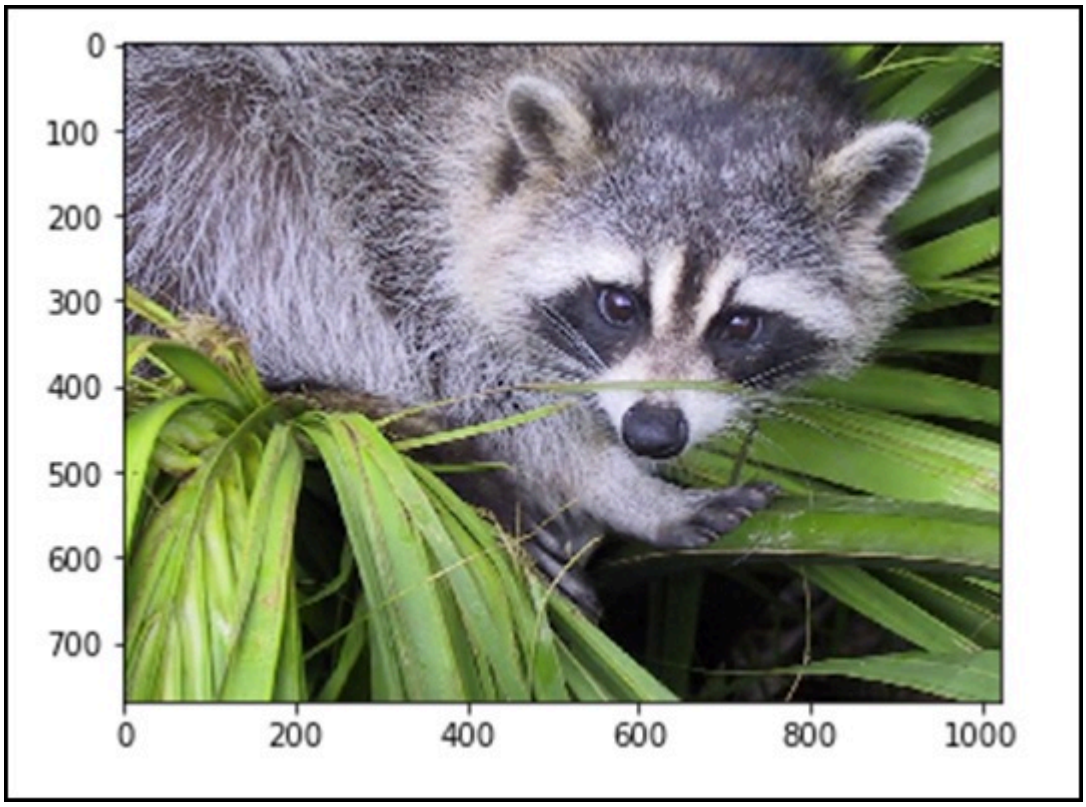
O **pacote misc** do SciPy vem com algumas imagens. Usamos essas imagens para aprender as manipulações de imagens. Consideremos o seguinte exemplo.

```
from scipy import misc
f = misc.face()
```

```
misc.imsave('face.png', f) # uses the Image module (PIL)

import matplotlib.pyplot as plt
plt.imshow(f)
plt.show()
```

O programa acima irá gerar a seguinte saída.



Quaisquer imagens em seu formato bruto são a combinação de cores representadas pelos números no formato matricial. Uma máquina entende e manipula as imagens com base apenas nesses números. RGB é uma forma popular de representação.

Vejamos as informações estatísticas da imagem acima.

```
from scipy import misc
face = misc.face(gray = False)
print face.mean(), face.max(), face.min()
```

O programa acima irá gerar a seguinte saída.

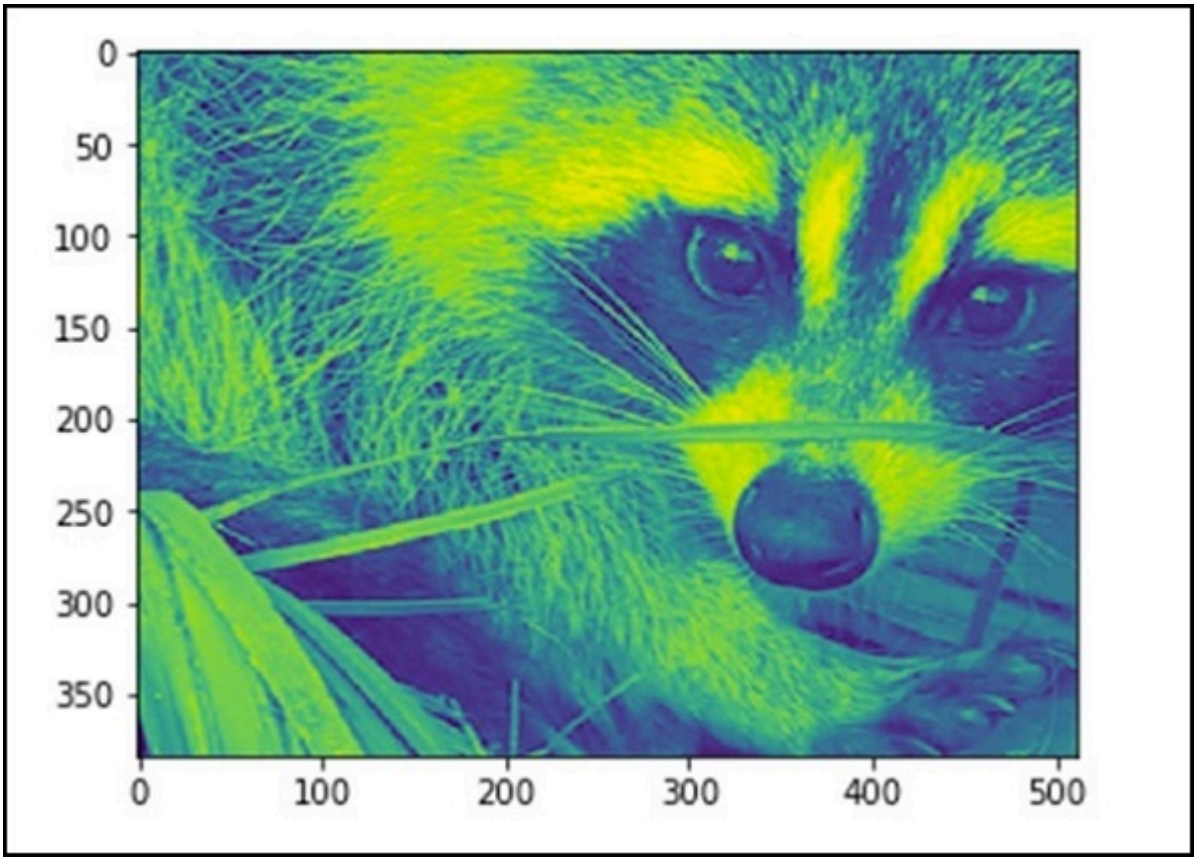
```
110.16274388631184, 255, 0
```

Agora sabemos que a imagem é feita de números, portanto qualquer alteração no valor do número altera a imagem original. Vamos realizar algumas transformações geométricas na imagem. A operação geométrica básica é cortar

```
from scipy import misc
face = misc.face(gray = True)
```

```
lx, ly = face.shape
# Cropping
crop_face = face[lx / 4: - lx / 4, ly / 4: - ly / 4]
import matplotlib.pyplot as plt
plt.imshow(crop_face)
plt.show()
```

O programa acima irá gerar a seguinte saída.

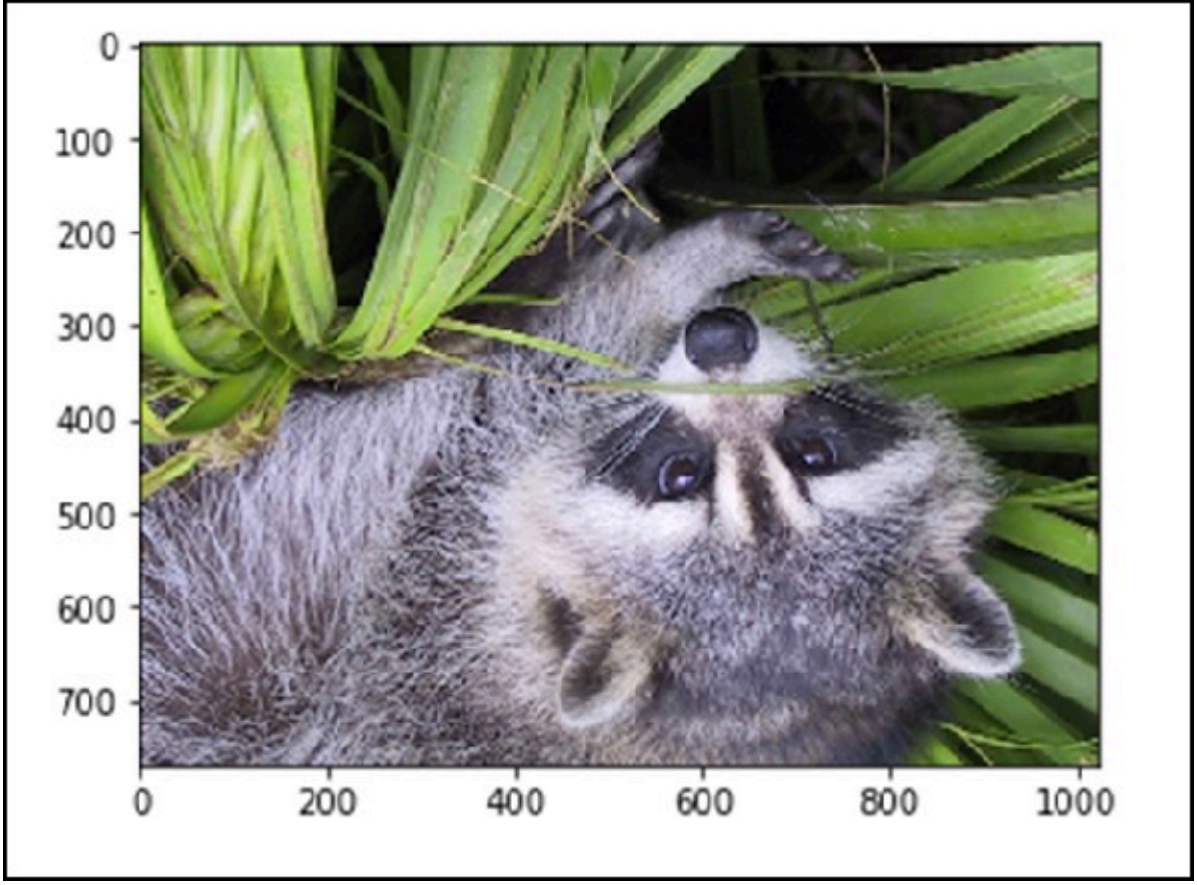


Também podemos realizar algumas operações básicas, como virar a imagem de cabeça para baixo, conforme descrito abaixo.

```
# up <-> down flip
from scipy import misc
face = misc.face()
flip_ud_face = np.flipud(face)

import matplotlib.pyplot as plt
plt.imshow(flip_ud_face)
plt.show()
```

O programa acima irá gerar a seguinte saída.

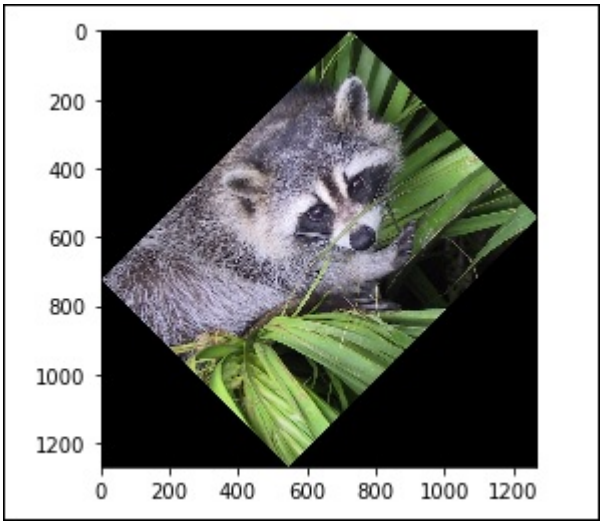


Além disso, temos a **função rotate()** , que gira a imagem em um ângulo especificado.

```
# rotation
from scipy import misc,ndimage
face = misc.face()
rotate_face = ndimage.rotate(face, 45)

import matplotlib.pyplot as plt
plt.imshow(rotate_face)
plt.show()
```

O programa acima irá gerar a seguinte saída.



Filtros

Vamos discutir como os filtros ajudam no processamento de imagens.

O que é filtragem no processamento de imagens?

A filtragem é uma técnica para modificar ou aprimorar uma imagem. Por exemplo, você pode filtrar uma imagem para enfatizar determinados recursos ou remover outros recursos. As operações de processamento de imagem implementadas com filtragem incluem Suavização, Nitidez e Melhoramento de Borda.

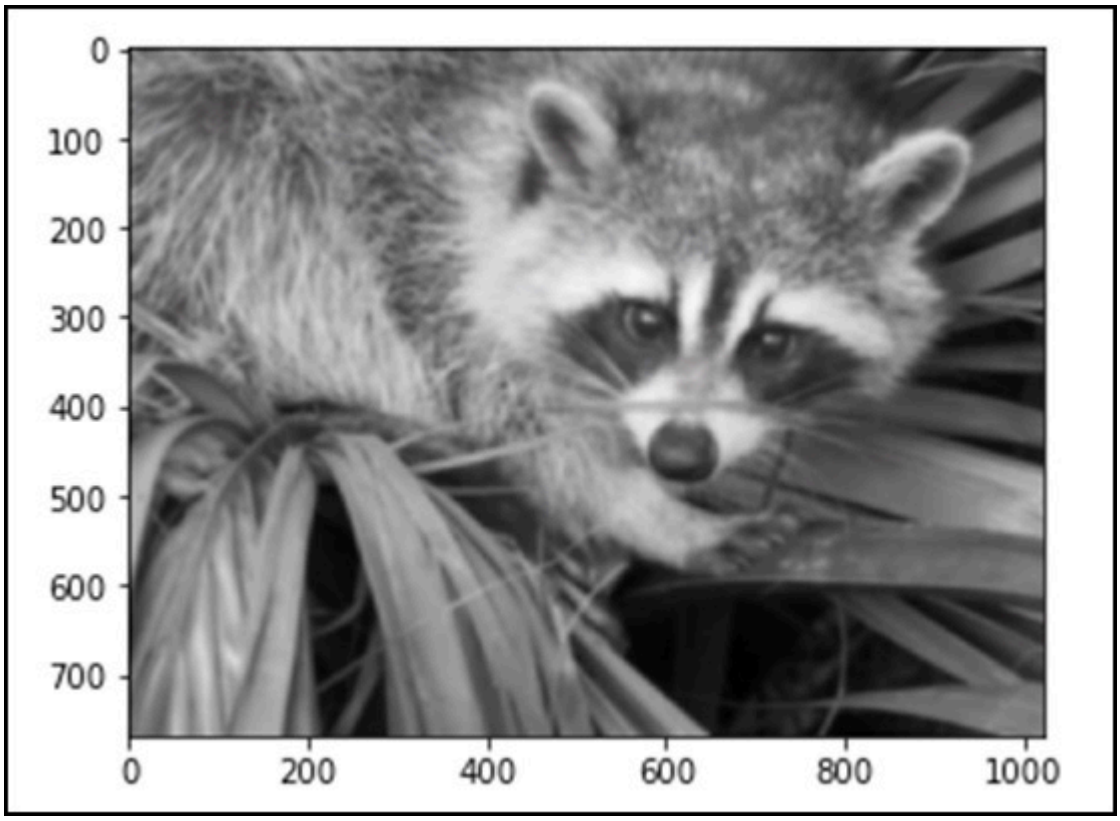
A filtragem é uma operação de vizinhança, na qual o valor de qualquer pixel na imagem de saída é determinado pela aplicação de algum algoritmo aos valores dos pixels na vizinhança do pixel de entrada correspondente. Vamos agora realizar algumas operações usando SciPy ndimage.

Desfoque

O desfoque é amplamente utilizado para reduzir o ruído na imagem. Podemos realizar uma operação de filtro e ver a mudança na imagem. Consideremos o seguinte exemplo.

```
from scipy import misc
face = misc.face()
blurred_face = ndimage.gaussian_filter(face, sigma=3)
import matplotlib.pyplot as plt
plt.imshow(blurred_face)
plt.show()
```

O programa acima irá gerar a seguinte saída.



O valor sigma indica o nível de desfoque em uma escala de cinco. Podemos ver a mudança na qualidade da imagem ajustando o valor sigma. Para obter mais detalhes sobre desfoque, clique em → Tutorial DIP (Digital Image Processing).

Detecção de borda

Vamos discutir como a detecção de bordas ajuda no processamento de imagens.

O que é detecção de borda?

A detecção de bordas é uma técnica de processamento de imagem para encontrar os limites de objetos nas imagens. Funciona detectando descontinuidades no brilho. A detecção de bordas é usada para segmentação de imagens e extração de dados em áreas como processamento de imagens, visão computacional e visão de máquina.

Os algoritmos de detecção de borda mais comumente usados incluem

- Sobel
- Astuto
- Prewitt
- Roberto
- Métodos de lógica difusa

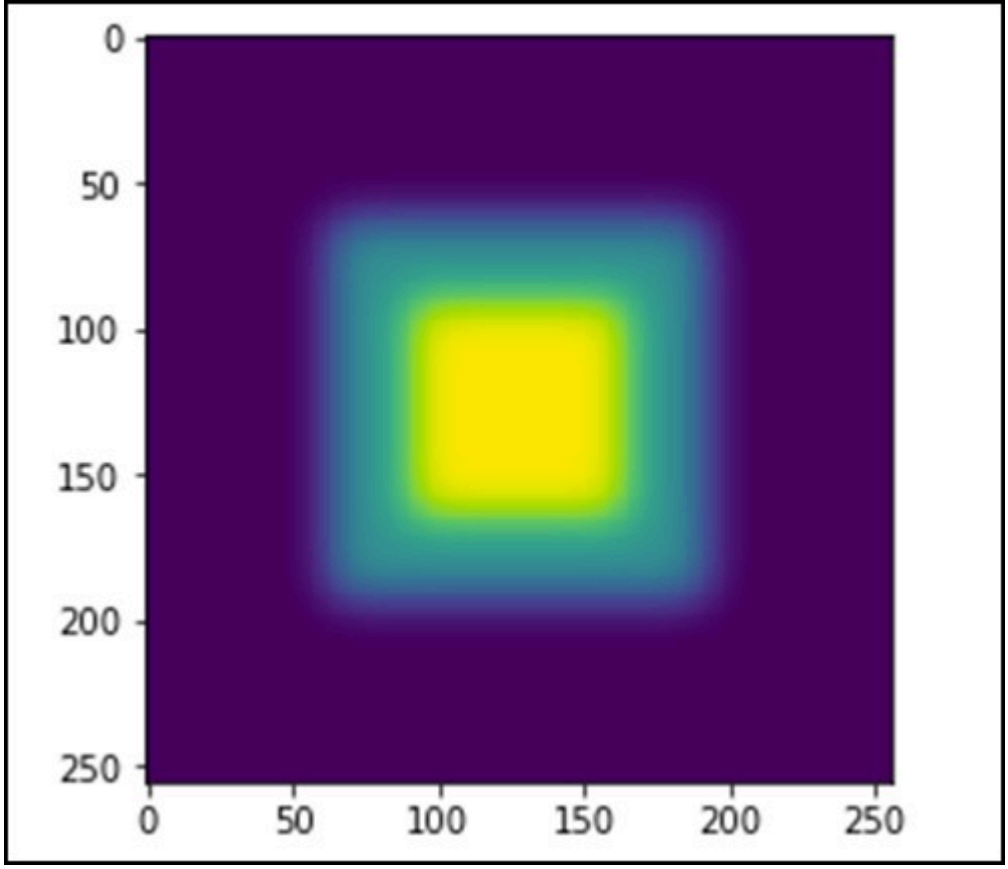
Consideremos o seguinte exemplo.

```
import scipy.ndimage as nd
import numpy as np

im = np.zeros((256, 256))
im[64:-64, 64:-64] = 1
im[90:-90,90:-90] = 2
im = ndimage.gaussian_filter(im, 8)

import matplotlib.pyplot as plt
plt.imshow(im)
plt.show()
```

O programa acima irá gerar a seguinte saída.



A imagem parece um bloco quadrado de cores. Agora detectaremos as bordas desses blocos coloridos. Aqui, ndimage fornece uma função chamada **Sobel** para realizar esta operação. Considerando que NumPy fornece a função **Hypot** para combinar as duas matrizes resultantes em uma.

Consideremos o seguinte exemplo.

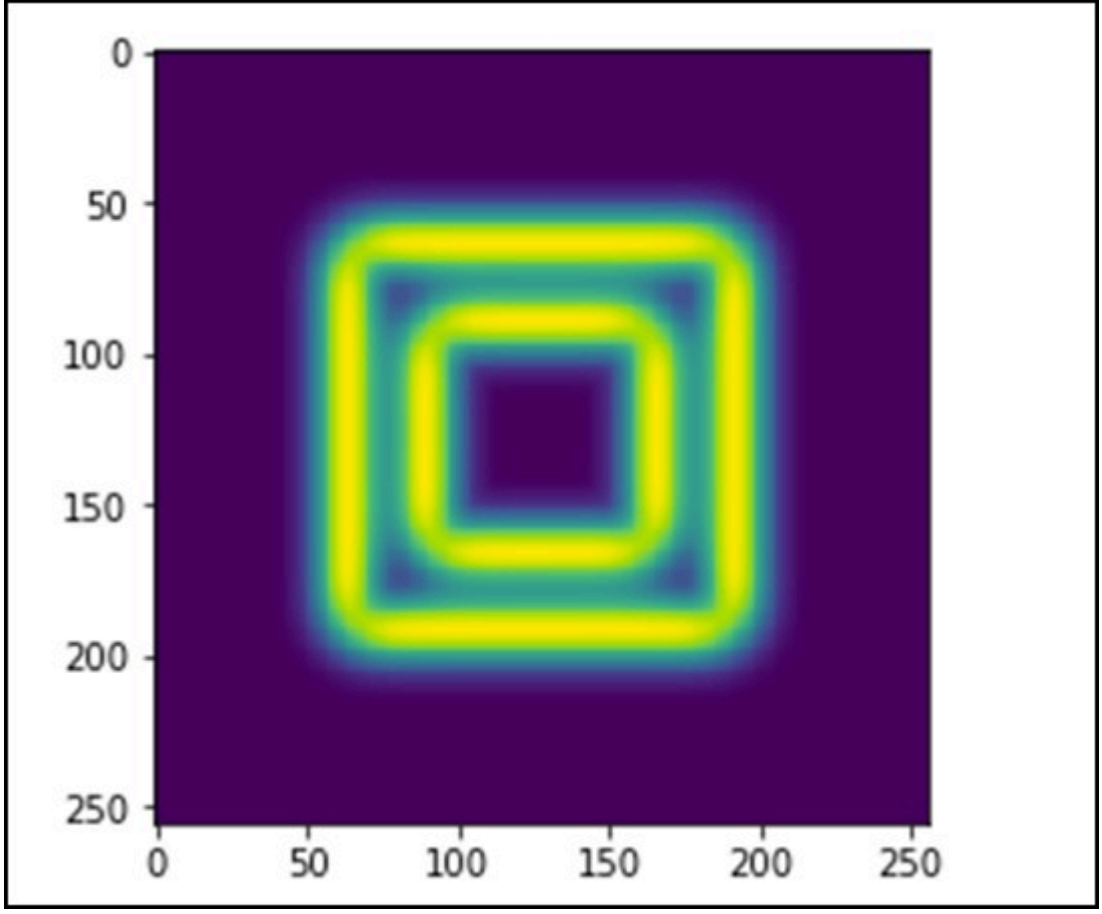
```
import scipy.ndimage as nd
import matplotlib.pyplot as plt

im = np.zeros((256, 256))
im[64:-64, 64:-64] = 1
im[90:-90,90:-90] = 2
im = ndimage.gaussian_filter(im, 8)

sx = ndimage.sobel(im, axis = 0, mode = 'constant')
sy = ndimage.sobel(im, axis = 1, mode = 'constant')
sob = np.hypot(sx, sy)

plt.imshow(sob)
plt.show()
```

O programa acima irá gerar a seguinte saída.



SciPy - Otimizar

O **pacote scipy.optimize** fornece vários algoritmos de otimização comumente usados. Este módulo contém os seguintes aspectos -

- Minimização irrestrita e restrita de funções escalares multivariadas (minimize()) usando uma variedade de algoritmos (por exemplo, BFGS, Nelder-Mead simplex, Newton Conjugate Gradient, COBYLA ou SLSQP)
- Rotinas de otimização globais (força bruta) (por exemplo, anneal(), bashopping())
- Algoritmos de minimização de mínimos quadrados (leastsq()) e ajuste de curva (curve_fit())
- Scalar univariate functions minimizers (minimize_scalar()) and root finders (newton())
- Multivariate equation system solvers (root()) using a variety of algorithms (e.g. hybrid Powell, Levenberg-Marquardt or large-scale methods such as Newton-Krylov)

Unconstrained & Constrained minimization of multivariate scalar functions

The **minimize() function** provides a common interface to unconstrained and constrained minimization algorithms for multivariate scalar functions in **scipy.optimize**. To demonstrate the minimization function, consider the problem of minimizing the Rosenbrock function of the NN variables −

$$f(x) = \sum_{i = 1}^{N-1} \:100(x_i - x_{i-1}^{2})$$

The minimum value of this function is 0, which is achieved when xi = 1.

Nelder–Mead Simplex Algorithm

In the following example, the minimize() routine is used with the **Nelder-Mead simplex algorithm (method = 'Nelder-Mead')** (selected through the method parameter). Let us consider the following example.

```
import numpy as np
from scipy.optimize import minimize

def rosen(x):

x0 = np.array([1.3, 0.7, 0.8, 1.9, 1.2])
res = minimize(rosen, x0, method='nelder-mead')

print(res.x)
```

The above program will generate the following output.

```
[7.93700741e+54 -5.41692163e+53 6.28769150e+53 1.38050484e+55 -4.14751333e+54]
```

The simplex algorithm is probably the simplest way to minimize a fairly well-behaved function. It requires only function evaluations and is a good choice for simple minimization problems. However, because it does not use any gradient evaluations, it may take longer to find the minimum.

Another optimization algorithm that needs only function calls to find the minimum is the **Powell's method**, which is available by setting method = 'powell' in the minimize() function.

Least Squares

Solve a nonlinear least-squares problem with bounds on the variables. Given the residuals f(x) (an m-dimensional real function of n real variables) and the loss function rho(s) (a scalar function), least_squares find a local minimum of the cost function F(x). Let us consider the following example.

In this example, we find a minimum of the Rosenbrock function without bounds on the independent variables.

```
#Rosenbrock Function
def fun_rosenbrock(x):
   return np.array([10 * (x[1] - x[0]**2), (1 - x[0])])

from scipy.optimize import least_squares
input = np.array([2, 2])
res = least_squares(fun_rosenbrock, input)
```

```
print res
```

Notice that, we only provide the vector of the residuals. The algorithm constructs the cost function as a sum of squares of the residuals, which gives the Rosenbrock function. The exact minimum is at x = [1.0,1.0].

The above program will generate the following output.

```
active_mask: array([ 0., 0.])
     cost: 9.8669242910846867e-30
     fun: array([ 4.44089210e-15, 1.11022302e-16])
     grad: array([ -8.89288649e-14, 4.44089210e-14])
     jac: array([[-20.00000015,10.],[ -1.,0.]])
  message: '`gtol` termination condition is satisfied.'
     nfev: 3
     njev: 3
  optimality: 8.8928864934219529e-14
     status: 1
     success: True
        x: array([ 1., 1.])
```

Root finding

Let us understand how root finding helps in SciPy.

Scalar functions

If one has a single-variable equation, there are four different root-finding algorithms, which can be tried. Each of these algorithms require the endpoints of an interval in which a root is expected (because the function changes signs). In general, **brentq** is the best choice, but the other methods may be useful in certain circumstances or for academic purposes.

Fixed-point solving

Um problema intimamente relacionado com encontrar os zeros de uma função é o problema de encontrar um ponto fixo de uma função. Um ponto fixo de uma função é o ponto no qual a avaliação da função retorna o ponto: g(x) = x. Claramente o ponto fixo de **gg** é a raiz de f(x) = g(x)−x. Equivalentemente, a raiz de **ff** é o ponto fixo de g(x) = f(x)+x. A rotina fixo_ponto fornece um método iterativo simples usando a **aceleração da sequência de Aitkens** para estimar o ponto fixo de **gg** , se um ponto de partida for fornecido.

Conjuntos de equações

Encontrar uma raiz de um conjunto de equações não lineares pode ser alcançado usando a **função root()** . Vários métodos estão disponíveis, entre os quais **hybr** (o padrão) e lm, usam respectivamente o **método híbrido de Powell** e o **método Levenberg-Marquardt** do MINPACK.

O exemplo a seguir considera a equação transcendental de variável única.

$$x2 + 2\cos(x) = 0$$

Uma raiz da qual pode ser encontrada da seguinte forma -

```
import numpy as np
from scipy.optimize import root
def func(x):
   return x*2 + 2 * np.cos(x)
sol = root(func, 0.3)
print sol
```

O programa acima irá gerar a seguinte saída.

```
fjac: array([[-1.]])
fun: array([ 2.22044605e-16])
message: 'The solution converged.'
  nfev: 10
  qtf: array([ -2.77644574e-12])
    r: array([-3.34722409])
  status: 1
  success: True
    x: array([-0.73908513])
```

SciPy - Estatísticas

Todas as funções estatísticas estão localizadas no subpacote **scipy.stats** e uma lista bastante completa dessas funções pode ser obtida usando a função **info(stats)** . Uma lista de variáveis aleatórias disponíveis também pode ser obtida na **documentação** do subpacote stats. Este módulo contém um grande número de distribuições de probabilidade, bem como uma biblioteca crescente de funções estatísticas.

Cada distribuição univariada tem sua própria subclasse conforme descrito na tabela a seguir -

Sr. Não.	Classe e descrição
1	**rv_contínuo** Uma classe genérica de variável aleatória contínua destinada à subclasse

2	**rv_discreto** Uma classe genérica de variável aleatória discreta destinada à subclasse
3	**rv_histograma** Gera uma distribuição dada por um histograma

Variável aleatória contínua normal

Uma distribuição de probabilidade na qual a variável aleatória X pode assumir qualquer valor é uma variável aleatória contínua. A palavra-chave location (loc) especifica a média. A palavra-chave escala (escala) especifica o desvio padrão.

Como uma instância da classe **rv_continuous** , o objeto **norm** herda dela uma coleção de métodos genéricos e os completa com detalhes específicos para esta distribuição específica.

Para calcular o CDF em vários pontos, podemos passar uma lista ou um array NumPy. Consideremos o seguinte exemplo.

```
from scipy.stats import norm
import numpy as np
print norm.cdf(np.array([1,-1., 0, 1, 3, 4, -2, 6]))
```

O programa acima irá gerar a seguinte saída.

```
array([ 0.84134475, 0.15865525, 0.5 , 0.84134475, 0.9986501 ,
0.99996833, 0.02275013, 1. ])
```

Para encontrar a mediana de uma distribuição, podemos usar a Função de Ponto Percentual (PPF), que é o inverso do CDF. Vamos entender usando o exemplo a seguir.

```
from scipy.stats import norm
print norm.ppf(0.5)
```

O programa acima irá gerar a seguinte saída.

```
0.0
```

Para gerar uma sequência de variáveis aleatórias, devemos usar o argumento da palavra-chave size, que é mostrado no exemplo a seguir.

```
from scipy.stats import norm
print norm.rvs(size = 5)
```

O programa acima irá gerar a seguinte saída.

```
array([ 0.20929928, -1.91049255, 0.41264672, -0.7135557 , -0.03833048])
```

A saída acima não é reproduzível. Para gerar os mesmos números aleatórios, use a função seed.

Distribuição uniforme

Uma distribuição uniforme pode ser gerada usando a função uniforme. Consideremos o seguinte exemplo.

```
from scipy.stats import uniform
print uniform.cdf([0, 1, 2, 3, 4, 5], loc = 1, scale = 4)
```

O programa acima irá gerar a seguinte saída.

```
array([ 0. , 0. , 0.25, 0.5 , 0.75, 1. ])
```

Construir distribuição discreta

Vamos gerar uma amostra aleatória e comparar as frequências observadas com as probabilidades.

Distribuição binomial

Como uma instância da **classe rv_discrete** , o **objeto binom** herda dela uma coleção de métodos genéricos e os completa com detalhes específicos para esta distribuição específica. Consideremos o seguinte exemplo.

```
from scipy.stats import uniform
print uniform.cdf([0, 1, 2, 3, 4, 5], loc = 1, scale = 4)
```

O programa acima irá gerar a seguinte saída.

```
array([ 0. , 0. , 0.25, 0.5 , 0.75, 1. ])
```

Estatísticas descritivas

As estatísticas básicas como Min, Max, Mean e Variance tomam o array NumPy como entrada e retornam os respectivos resultados. Algumas funções estatísticas básicas disponíveis no **pacote scipy.stats** são descritas na tabela a seguir.

Sr. Não.	Descrição da função
1	**descrever()** Calcula várias estatísticas descritivas do array passado
2	**média()** Calcula a média geométrica ao longo do eixo especificado
3	**hmean()** Calcula a média harmônica ao longo do eixo especificado
4	**curtose()** Calcula a curtose
5	**modo()** Retorna o valor modal
6	**inclinação()** Testa a distorção dos dados
7	**f_oneway()** Executa uma ANOVA unidirecional
8	**qqr()** Calcula o intervalo interquartil dos dados ao longo do eixo especificado
9	**pontuação z()** Calcula a pontuação z de cada valor na amostra, em relação à média amostral e ao desvio padrão
10	**sem()** Calcula o erro padrão da média (ou erro padrão de medição) dos valores na matriz de entrada

Várias dessas funções têm uma versão semelhante em **scipy.stats.mstats** , que funciona para arrays mascarados. Vamos entender isso com o exemplo dado abaixo.

```
from scipy import stats
import numpy as np
x = np.array([1,2,3,4,5,6,7,8,9])
print x.max(),x.min(),x.mean(),x.var()
```

O programa acima irá gerar a seguinte saída.

```
(9, 1, 5.0, 6.666666666666667)
```

Teste T

Vamos entender como o teste T é útil no SciPy.

ttest_1samp

Calcula o teste T para a média de UM grupo de pontuações. Este é um teste bilateral para a hipótese nula de que o valor esperado (média) de uma amostra de observações independentes 'a' é igual à média da população dada, **popmean** . Consideremos o seguinte exemplo.

```
from scipy import stats
rvs = stats.norm.rvs(loc = 5, scale = 10, size = (50,2))
print stats.ttest_1samp(rvs,5.0)
```

O programa acima irá gerar a seguinte saída.

```
Ttest_1sampResult(statistic = array([-1.40184894, 2.70158009]),
pvalue = array([ 0.16726344, 0.00945234]))
```

Comparando duas amostras

Nos exemplos a seguir, existem duas amostras, que podem vir da mesma distribuição ou de distribuições diferentes, e queremos testar se essas amostras têm as mesmas propriedades estatísticas.

ttest_ind - Calcula o teste T para as médias de duas amostras independentes de pontuações. Este é um teste bilateral para a hipótese nula de que duas amostras independentes têm valores médios (esperados) idênticos. Este teste assume que as populações têm variâncias idênticas por padrão.

Podemos usar este teste se observarmos duas amostras independentes da mesma população ou de uma população diferente. Consideremos o seguinte exemplo.

```
from scipy import stats
rvs1 = stats.norm.rvs(loc = 5,scale = 10,size = 500)
rvs2 = stats.norm.rvs(loc = 5,scale = 10,size = 500)
print stats.ttest_ind(rvs1,rvs2)
```

O programa acima irá gerar a seguinte saída.

```
Ttest_indResult(statistic = -0.67406312233650278, pvalue = 0.50042727502272966)
```

Você pode testar o mesmo com uma nova matriz do mesmo comprimento, mas com uma média variada. Use um valor diferente em **loc** e teste o mesmo.

SciPy-CSGraph

CSGraph significa **Compressed Sparse Graph** , que se concentra em algoritmos de gráficos rápidos baseados em representações de matrizes esparsas.

Representações gráficas

Para começar, vamos entender o que é um gráfico esparso e como ele ajuda nas representações gráficas.

O que exatamente é um gráfico esparso?

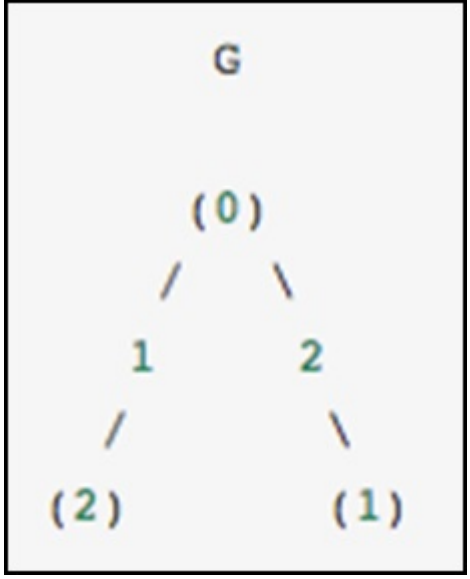
Um gráfico é apenas uma coleção de nós, que possuem links entre eles. Os gráficos podem representar quase tudo – conexões de redes sociais, onde cada nó é uma pessoa e está conectado a conhecidos; imagens, onde cada nó é um pixel e está conectado a pixels vizinhos; pontos em uma distribuição de alta dimensão, onde cada nó está conectado aos seus vizinhos mais próximos; e praticamente qualquer outra coisa que você possa imaginar.

Uma maneira muito eficiente de representar dados gráficos é em uma matriz esparsa: vamos chamá-la de G. A matriz G é de tamanho N x N, e G[i, j] fornece o valor da conexão entre o nó 'i' e o nó 'j'. Um gráfico esparso contém principalmente zeros - ou seja, a maioria dos nós possui apenas algumas conexões. Esta propriedade acaba sendo verdadeira na maioria dos casos de interesse.

A criação do submódulo de gráfico esparso foi motivada por vários algoritmos usados no scikit-learn que incluíam o seguinte -

- **Isomap** - Um algoritmo de aprendizado múltiplo, que requer encontrar os caminhos mais curtos em um gráfico.
- **Hierarchical clustering** - Um algoritmo de cluster baseado em uma árvore geradora mínima.
- **Spectral Decomposition** - Um algoritmo de projeção baseado em laplacianos de grafos esparsos.

Como exemplo concreto, imagine que gostaríamos de representar o seguinte gráfico não direcionado -



Este gráfico possui três nós, onde os nós 0 e 1 são conectados por uma aresta de peso 2, e os nós 0 e 2 são conectados por uma aresta de peso 1. Podemos construir as representações densa, mascarada e esparsa conforme mostrado no exemplo a seguir , tendo em mente que um gráfico não direcionado é representado por uma matriz simétrica.

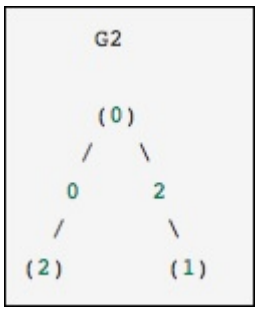
```
G_dense = np.array([ [0, 2, 1],
                     [2, 0, 0],
                     [1, 0, 0] ])

G_masked = np.ma.masked_values(G_dense, 0)
from scipy.sparse import csr_matrix

G_sparse = csr_matrix(G_dense)
print G_sparse.data
```

O programa acima irá gerar a seguinte saída.

```
array([2, 1, 2, 1])
```



Isso é idêntico ao gráfico anterior, exceto que os nós 0 e 2 estão conectados por uma aresta de peso zero. Neste caso, a representação densa acima leva a ambiguidades - como podem ser representadas não arestas, se zero é um valor significativo. Neste caso, uma representação mascarada ou esparsa deve ser usada para eliminar a ambiguidade.

Consideremos o seguinte exemplo.

```
from scipy.sparse.csgraph import csgraph_from_dense
G2_data = np.array
([
   [np.inf, 2, 0 ],
   [2, np.inf, np.inf],
   [0, np.inf, np.inf]
])
G2_sparse = csgraph_from_dense(G2_data, null_value=np.inf)
print G2_sparse.data
```

O programa acima irá gerar a seguinte saída.

```
array([ 2., 0., 2., 0.])
```

Escadas de palavras usando gráficos esparsos

Word ladders é um jogo inventado por Lewis Carroll, no qual as palavras são ligadas pela mudança de uma única letra a cada passo. Por exemplo -

APE → APT → AIT → BIT → BIG → BAG → MAG → HOMEM

Aqui, passamos de "APE" para "MAN" em sete etapas, mudando uma letra de cada vez. A questão é: podemos encontrar um caminho mais curto entre essas palavras usando a mesma regra? Este problema é naturalmente expresso como um problema de gráfico esparso. Os nós corresponderão a palavras individuais e criaremos conexões entre palavras que diferem no máximo por uma letra.

Obtendo uma lista de palavras

Primeiro, é claro, devemos obter uma lista de palavras válidas. Estou executando o Mac e o Mac possui um dicionário de palavras no local fornecido no bloco de código a seguir. Se você estiver em uma arquitetura diferente, talvez seja necessário pesquisar um pouco para encontrar o dicionário do sistema.

```
wordlist = open('/usr/share/dict/words').read().split()
print len(wordlist)
```

O programa acima irá gerar a seguinte saída.

```
235886
```

Agora queremos examinar palavras de comprimento 3, então vamos selecionar apenas as palavras de comprimento correto. Também eliminaremos palavras que comecem com letras maiúsculas (substantivos próprios) ou que contenham caracteres não alfanuméricos, como

apóstrofos e hífens. Por fim, garantiremos que tudo esteja em letras minúsculas para uma comparação posterior.

```
word_list = [word for word in word_list if len(word) == 3]
word_list = [word for word in word_list if word[0].islower()]
word_list = [word for word in word_list if word.isalpha()]
word_list = map(str.lower, word_list)
print len(word_list)
```

O programa acima irá gerar a seguinte saída.

```
1135
```

Agora, temos uma lista de 1.135 palavras válidas de três letras (o número exato pode mudar dependendo da lista específica usada). Cada uma dessas palavras se tornará um nó em nosso gráfico, e criaremos arestas conectando os nós associados a cada par de palavras, que difere em apenas uma letra.

```
import numpy as np
word_list = np.asarray(word_list)

word_list.dtype
word_list.sort()

word_bytes = np.ndarray((word_list.size, word_list.itemsize),
   dtype = 'int8',
   buffer = word_list.data)
print word_bytes.shape
```

O programa acima irá gerar a seguinte saída.

```
(1135, 3)
```

Usaremos a distância de Hamming entre cada ponto para determinar quais pares de palavras estão conectados. A distância de Hamming mede a fração de entradas entre dois vetores, que diferem: quaisquer duas palavras com distância de Hamming igual a 1/N1/N, onde NN é o número de letras conectadas na palavra escada.

```
from scipy.spatial.distance import pdist, squareform
from scipy.sparse import csr_matrix
hamming_dist = pdist(word_bytes, metric = 'hamming')
graph = csr_matrix(squareform(hamming_dist < 1.5 / word_list.itemsize))
```

Ao comparar as distâncias, não usamos igualdade porque pode ser instável para valores de ponto flutuante. A desigualdade produz o resultado desejado, desde que não haja duas entradas da lista de palavras idênticas. Agora que nosso gráfico está configurado, usaremos a busca pelo caminho mais curto para encontrar o caminho entre quaisquer duas palavras no gráfico.

```
i1 = word_list.searchsorted('ape')
i2 = word_list.searchsorted('man')
print word_list[i1],word_list[i2]
```

O programa acima irá gerar a seguinte saída.

```
ape, man
```

Precisamos verificar se elas correspondem, pois se as palavras não estiverem na lista haverá um erro na saída. Agora, tudo o que precisamos é encontrar o caminho mais curto entre esses dois índices no gráfico. Usaremos o algoritmo **de Dijkstra** , pois ele nos permite encontrar o caminho para apenas um nó.

```
from scipy.sparse.csgraph import dijkstra
distances, predecessors = dijkstra(graph, indices = i1, return_predecessors = True)
print distances[i2]
```

O programa acima irá gerar a seguinte saída.

```
5.0
```

Assim, vemos que o caminho mais curto entre o “macaco” e o “homem” contém apenas cinco passos. Podemos usar os predecessores retornados pelo algoritmo para reconstruir esse caminho.

```
path = []
i = i2

while i != i1:
   path.append(word_list[i])
   i = predecessors[i]

path.append(word_list[i1])
print path[::-1]i2]
```

O programa acima irá gerar a seguinte saída.

```
['ape', 'ope', 'opt', 'oat', 'mat', 'man']
```

SciPy - Espacial

O **pacote scipy.spatial** pode calcular triangulações, diagramas de Voronoi e cascos convexos de um conjunto de pontos, aproveitando a **biblioteca Qhull** . Além disso, contém **implementações KDTree** para consultas de pontos vizinhos mais próximos e utilitários para cálculos de distância em várias métricas.

Triangulações de Delaunay

Vamos entender o que são triangulações de Delaunay e como elas são usadas no SciPy.

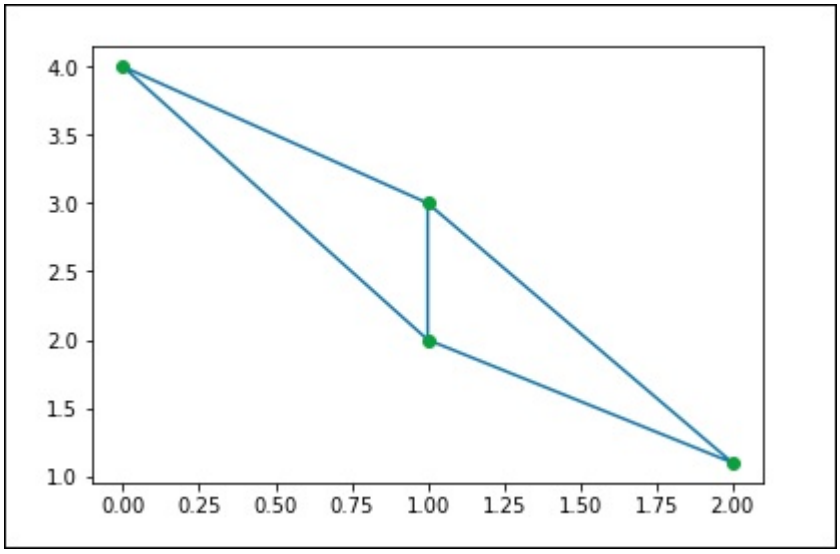
O que são triangulações de Delaunay?

Em matemática e geometria computacional, uma triangulação de Delaunay para um determinado conjunto **P** de pontos discretos em um plano é uma triangulação **DT(P)** tal que nenhum ponto em **P** está dentro do círculo circunscrito de qualquer triângulo em DT(P).

Podemos calcular o mesmo através do SciPy. Consideremos o seguinte exemplo.

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

O programa acima irá gerar a seguinte saída.



Pontos Coplanares

Vamos entender o que são Pontos Coplanares e como eles são usados no SciPy.

O que são pontos coplanares?

Pontos coplanares são três ou mais pontos que estão no mesmo plano. Lembre-se de que um plano é uma superfície plana que se estende sem fim em todas as direções. Geralmente é mostrado em livros de matemática como uma figura de quatro lados.

Vamos ver como podemos encontrar isso usando o SciPy. Consideremos o seguinte exemplo.

```
from scipy.spatial import Delaunay
points = np.array([[0, 0], [0, 1], [1, 0], [1, 1], [1, 1]])
tri = Delaunay(points)
print tri.coplanar
```

O programa acima irá gerar a seguinte saída.

```
array([[4, 0, 3]], dtype = int32)
```

Isso significa que o ponto 4 reside próximo ao triângulo 0 e ao vértice 3, mas não está incluído na triangulação.

Cascos convexos

Vamos entender o que são cascos convexos e como eles são usados no SciPy.

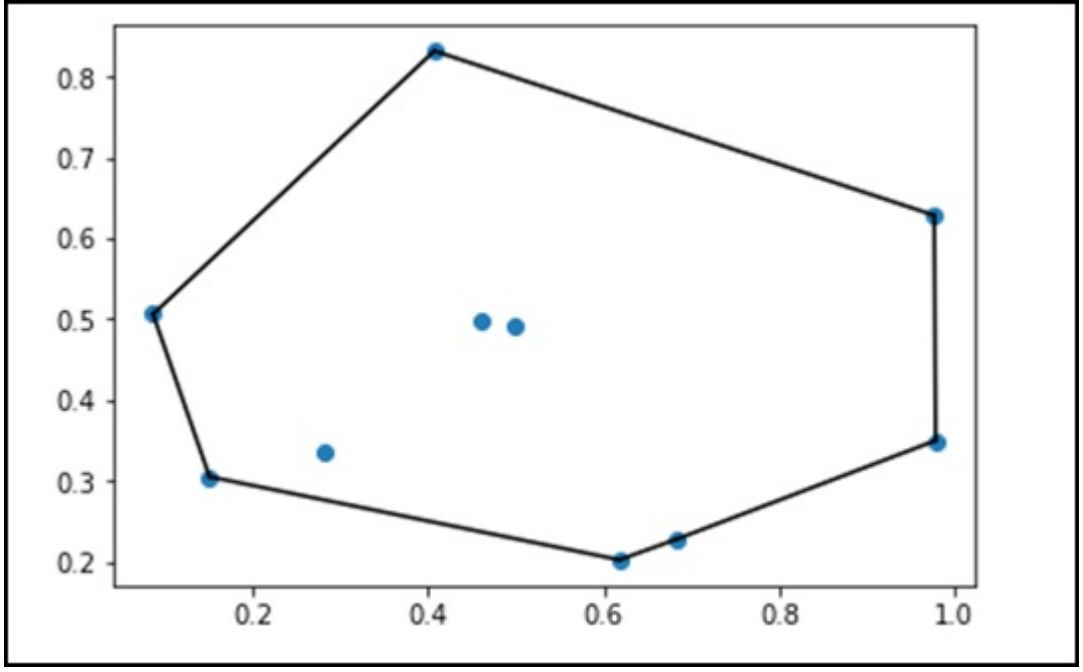
O que são cascos convexos?

Em matemática, a **casca convexa** ou **envelope convexo** de um conjunto de pontos X no plano euclidiano ou num espaço euclidiano (ou, mais geralmente, num espaço afim sobre os reais) é o menor **conjunto convexo** que contém X.

Vamos considerar o exemplo a seguir para entendê-lo em detalhes.

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

O programa acima irá gerar a seguinte saída.



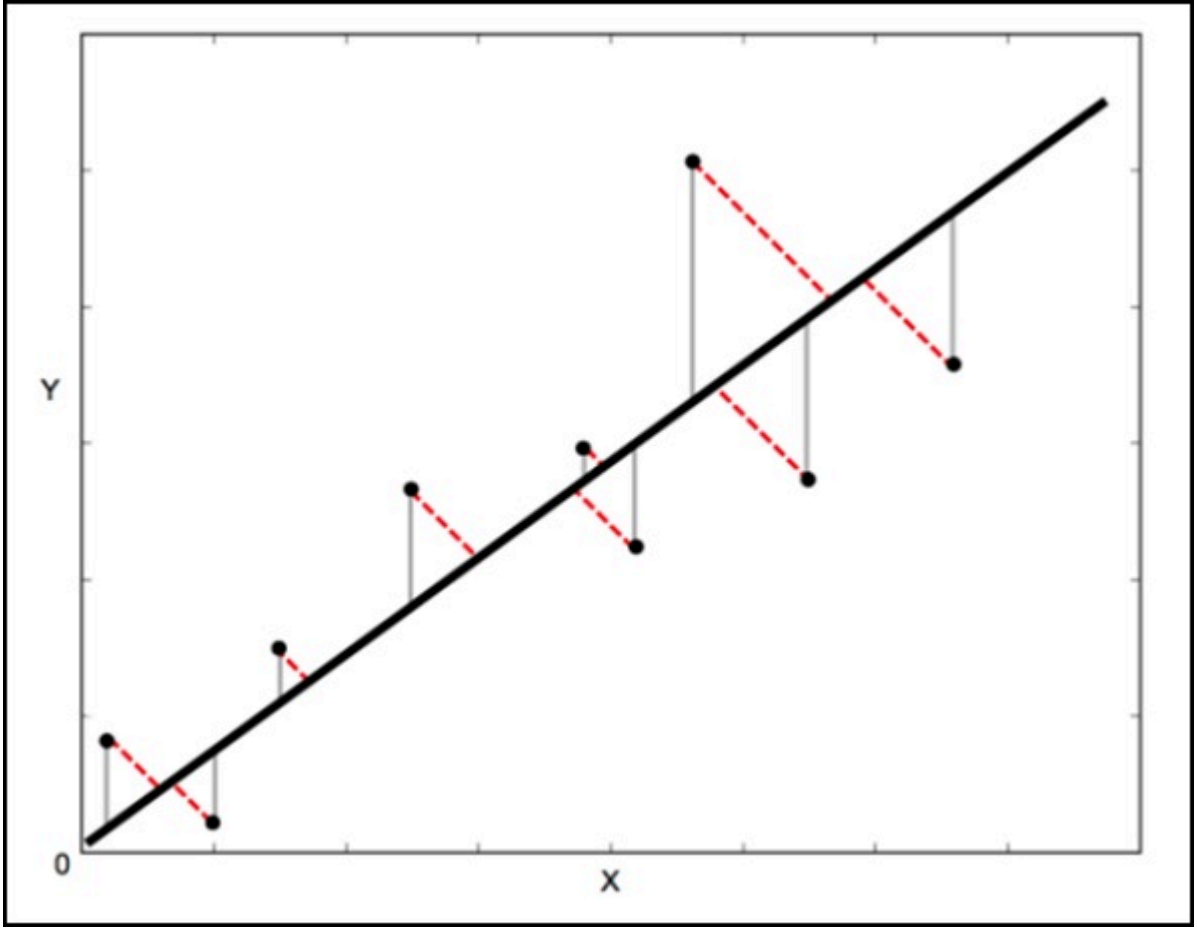
SciPy-ODR

ODR significa **Regressão de Distância Ortogonal** , que é usada nos estudos de regressão. A regressão linear básica é frequentemente usada para estimar a relação entre as duas variáveis **y** e **x** desenhando a linha de melhor ajuste no gráfico.

O método matemático utilizado para isso é conhecido como **Mínimos Quadrados** e visa minimizar a soma do erro quadrático de cada ponto. A questão principal aqui é como calcular o erro (também conhecido como resíduo) para cada ponto?

Em uma regressão linear padrão, o objetivo é prever o valor Y a partir do valor X – portanto, a coisa sensata a fazer é calcular o erro nos valores Y (mostrados como linhas cinzas na imagem a seguir). No entanto, às vezes é mais sensato levar em conta o erro tanto em X quanto em Y (conforme mostrado pelas linhas vermelhas pontilhadas na imagem a seguir).

Por exemplo - Quando você sabe que suas medições de X são incertas ou quando não deseja focar nos erros de uma variável em detrimento de outra.



A regressão de distância ortogonal (ODR) é um método que pode fazer isso (ortogonal neste contexto significa perpendicular – portanto, calcula erros perpendiculares à linha, em vez de apenas 'verticalmente').

Implementação scipy.odr para regressão univariada

O exemplo a seguir demonstra a implementação de scipy.odr para regressão univariada.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.odr import *
import random

# Initiate some data, giving some randomness using random.random().
x = np.array([0, 1, 2, 3, 4, 5])
y = np.array([i**2 + random.random() for i in x])

# Define a function (quadratic in our case) to fit the data with.
def linear_func(p, x):
   m, c = p
   return m*x + c

# Create a model for fitting.
linear_model = Model(linear_func)

# Create a RealData object using our initiated data from above.
```

```
data = RealData(x, y)

# Set up ODR with the model and data.
odr = ODR(data, linear_model, beta0=[0., 1.])

# Run the regression.
out = odr.run()

# Use the in-built pprint method to give us results.
out.pprint()
```

O programa acima irá gerar a seguinte saída.

```
Beta: [ 5.51846098 -4.25744878]
Beta Std Error: [ 0.7786442 2.33126407]

Beta Covariance: [
  [ 1.93150969 -4.82877433]
  [ -4.82877433 17.31417201
]]

Residual Variance: 0.313892697582
Inverse Condition #: 0.146618499389
Reason(s) for Halting:
  Sum of squares convergence
```

SciPy - Pacote Especial

As funções disponíveis no pacote especial são funções universais, que seguem a transmissão e o loop automático de array.

Vejamos algumas das funções especiais usadas com mais frequência -

- Função raiz cúbica
- Função exponencial
- Função Exponencial de Erro Relativo
- Função exponencial de soma de log
- Função Lambert
- Função de permutações e combinações
- Função Gama

Vamos agora entender resumidamente cada uma dessas funções.

Função raiz cúbica

A sintaxe desta função de raiz cúbica é – scipy.special.cbrt(x). Isso irá buscar a raiz cúbica elemento a elemento de **x** .

Consideremos o seguinte exemplo.

```
from scipy.special import cbrt
res = cbrt([10, 9, 0.1254, 234])
print res
```

O programa acima irá gerar a seguinte saída.

```
[ 2.15443469 2.08008382 0.50053277 6.16224015]
```

Função exponencial

A sintaxe da função exponencial é – scipy.special.exp10(x). Isso calculará 10 ** x elementos.

Consideremos o seguinte exemplo.

```
from scipy.special import exp10
res = exp10([2, 9])
print res
```

O programa acima irá gerar a seguinte saída.

```
[1.00000000e+02  1.00000000e+09]
```

Função Exponencial de Erro Relativo

A sintaxe para esta função é – scipy.special.exprel(x). Ele gera o erro relativo exponencial, (exp(x) - 1)/x.

Quando **x** está próximo de zero, exp(x) está próximo de 1, então o cálculo numérico de exp(x) - 1 pode sofrer uma perda catastrófica de precisão. Então exrel(x) é implementado para evitar a perda de precisão, que ocorre quando **x** está próximo de zero.

Consideremos o seguinte exemplo.

```
from scipy.special import exprel
res = exprel([-0.25, -0.1, 0, 0.1, 0.25])
print res
```

O programa acima irá gerar a seguinte saída.

```
[0.88479687 0.95162582 1.   1.05170918 1.13610167]
```

Função exponencial de soma de log

A sintaxe para esta função é – scipy.special.logsumexp(x). Ajuda a calcular o log da soma das exponenciais dos elementos de entrada.

Consideremos o seguinte exemplo.

```
from scipy.special import logsumexp
import numpy as np
a = np.arange(10)
res = logsumexp(a)
print res
```

O programa acima irá gerar a seguinte saída.

```
9.45862974443
```

Função Lambert

A sintaxe para esta função é – scipy.special.lambertw(x). Também é chamada de função Lambert W. A função Lambert W W(z) é definida como a função inversa de w * exp(w). Em outras palavras, o valor de W(z) é tal que z = W(z) * exp(W(z)) para qualquer número complexo z.

A função Lambert W é uma função de vários valores com infinitas ramificações. Cada ramo fornece uma solução separada da equação z = w exp(w). Aqui, as ramificações são indexadas pelo inteiro k.

Consideremos o seguinte exemplo. Aqui, a função Lambert W é o inverso de w exp(w).

```
from scipy.special import lambertw
w = lambertw(1)
print w
print w * np.exp(w)
```

O programa acima irá gerar a seguinte saída.

```
(0.56714329041+0j)
(1+0j)
```

Permutações e combinações

Vamos discutir permutações e combinações separadamente para entendê-las claramente.

Combinations - A sintaxe para a função de combinações é – scipy.special.comb(N,k). Vamos considerar o seguinte exemplo -

```
from scipy.special import comb
res = comb(10, 3, exact = False,repetition=True)
print res
```

O programa acima irá gerar a seguinte saída.

```
220.0
```

Note - Os argumentos da matriz são aceitos apenas para o caso exato = Falso. Se k > N, N < 0 ou k < 0, então um 0 será retornado.

Permutations - A sintaxe para função de combinações é – scipy.special.perm(N,k). Permutações de N coisas tomadas k de cada vez, ou seja, k-permutações de N. Isto também é conhecido como "permutações parciais".

Consideremos o seguinte exemplo.

```
from scipy.special import perm
res = perm(10, 3, exact = True)
print res
```

O programa acima irá gerar a seguinte saída.

```
720
```

Função Gama

A função gama é frequentemente referida como fatorial generalizado, pois z*gamma(z) = gama(z+1) e gama(n+1) = n!, para um número natural 'n'.

A sintaxe para a função de combinações é – scipy.special.gamma(x). Permutações de N coisas tomadas k de cada vez, ou seja, k-permutações de N. Isto também é conhecido como "permutações parciais".

A sintaxe para a função de combinações é – scipy.special.gamma(x). Permutações de N coisas tomadas k de cada vez, ou seja, k-permutações de N. Isto também é conhecido como "permutações parciais".

```
from scipy.special import gamma
res = gamma([0, 0.5, 1, 5])
print res
```

O programa acima irá gerar a seguinte saída.

```
[inf 1.77245385 1. 24.]
```