SciPy

—数值计算库

目录

- 常数和特殊函数
- 线性代数-linalg
- 优化—optimize
  - 非线性方程组求解
  - 最小二乘拟合
  - 函数最小值
- 插值—interpolate
  - B样条曲线插值
  - 外推和Spline拟合
  - 二维插值

目录

- 数值积分—integrate
  - 球的体积
  - 解常微分方程组
- 统计—stats
  - 连续和离散概率分布
  - 二项、泊松、伽玛分布
- 稀疏矩阵—sparse

SciPy在NumPy的基础上增加了众多的数学、科学以及工程计算中常用的模块，例如线性代数、常微分方程数值求解、优化、统计、信号处理、图像处理、稀疏矩阵，等等。在本节将通过实例介绍SciPy中常用的一些模块。在实例程序中会使用matplotlib绘制二维和三维图表，在后续章节中这些绘图库进行详细介绍。

```
>>> import scipy

>>> scipy.__version__
```

常数和特殊函数

SciPy的constants模块包含了众多的物理常数：

```
>>> from scipy import constants as C
>>> C.c #真空中的光速
299792458.0
>>> C.h #普朗克常数
6.62606930800080626-34
```

在字典physical_constants中，以物理常量名为键，对应的值是一个含有三个元素的元组， 分别为常数值、单位及误差，例如下面的程序可以查看电子质量：

```
>>> C.physical_constants["electron mass"]
 (9.109382599999998e-31,'kg', 1.5999999999999999e-37)
>>> C.physical_constants
```

常数和特殊函数

除了物理常数之外，constants模块中还包括许多单位信息，例如：

```
>>> C.mile # 1英里等于多少米
1609.3439999999998
>>>C.inch      # 1英寸等于多少米
0.025399999999999999
>>> C.gram    # 1克等于多少千克
0.001
>>> C.pound    # 1磅等于多少千克
0.45359236999999997
```

常数和特殊函数

SciPy的special模块是一个非常完整的函数库，其中包含了基本数学函数、特殊数学函数以及NumPy中出现的所有函数。由于函数数量众多，本节仅对其进行简要介绍。至于具体包含的函数列表，请参考SciPy的帮助文档。

伽玛函数Γ是概率统计学中经常出现的一个特殊函数，它的计算公式如下：

$$\Gamma(z)=\int_0^{\infty} t^{z-1}e^{-t}dt$$

常数和特殊函数

显然，通过此公式计算 Γ 函数的值是比较麻烦的，可以用special模块中的gamma()进行计算：

```
>>> import scipy.special as S
>>> S.gamma(4)
6.0
>>> S.gamma(0.5)
1.772458509055159
>>> S.gamma(1+1j)  # gamma 函数支持复数
(0.4980156681183562 9-0.15494982836181106j )
>>> S.gamma(1000)
inf
```

常数和特殊函数

Γ 函数是阶乘函数在实数和复数范围上的扩展，它的增长速度非常快，因为1000的阶乘已经超过了双精度浮点数的表示范围，因此结果是无穷大。为了计算更大的范围，可以使用S.gammaln()：

```
>>>S.gammaln(1000)
5905.2204232091817
```

S.gammaln(x)计算ln(|Γ(x)|)的值，它使用特殊的算法，直接计算 Γ 函数的对数值，因此可以表示更大的范围。

常数和特殊函数

special模块中的某些函数并不是数学意义上的特殊函数，例如log1p(x)计算log(1+x)的值。这是由于浮点数的精度有限，无法很精确地表示十分接近1的实数。例如无法用浮点数表示”1 + 1e-20”的值，因此“log(1+1e-20)”的值为0,而当使用log1p()时，则可以很精确地计算：

```
>>> 1 + 1e-20
1.0
>>> log(1+1e-20)
0.0
>>> S.log1p(1e-20)
9.999999999999995e-21
```

实际上当x非常小时，log1p(x)约等于x 。

线性代数-linalg

NumPy和SciPy都提供了线性代数函数库linalg，SciPy的线性代数库比NumPy更加全面。

numpy.linalg.solve(A, b)和scipy.linalg.solve(A, b)可以用来解线性方程组Ax=b，也就是计算$x=A^{-1}b$。这里A为(M, M)的方形矩阵，x和b为长为M的向量。有时候A是固定的，需要对多组b进行求解，因此第二个参数也可以是(M, N)的矩阵B。这样计算出来的X也为(M, N)的矩阵。它相当于计算$A^{-1}B$。

线性代数-linalg

在一些矩阵公式中经常会出现类似于$A^{-1}B$的运算，它们用solve(A,B)计算，这要比直接计算逆矩阵然后做矩阵乘法更快捷一些：

```
import numpy as np
from scipy import linalg
M, N = 500, 50
A = np.random.rand(M, M)
B = np.random.rand(M, N)
X1 = linalg.solve(A, B)
X2 = np.dot(linalg.inv(A), B)
np.allclose(X1, X2)
```

```
True
```

```
%timeit linalg.solve(A, B)
%timeit np.dot(linalg.inv(A), B)
```

```
linalg.(tab)
```

线性代数-linalg

若需要对多组B进行求解，但是又不好将它们合并成一个矩阵，例如某些矩阵公式中可能会有$A^{-1}B$, $A^{-1}C$, $A^{-1}D$ 等乘法，而B,C,D是通过某种方式逐次计算的。这时可以采用lu_factor()和lu_solve()。先调用lu_factor(A)对矩阵A进行LU分解，得到一个元组：(LU矩阵, 排序数组)，将这个元组传递给lu_solve()，即可对不同的B进行求解。由于已经对A进行了LU分解，lu_solve()能够很快得出结果。

```
luf = linalg.lu_factor(A)
X3 = linalg.lu_solve(luf, B)
np.allclose(X1, X3)
```

```
True
```

线性代数-linalg

除了使用lu_factor()和lu_solve()之外，可以先通过inv()计算逆矩阵，然后通过dot()计算矩阵乘积。下面比较二者的速度。

```
M, N = 1000, 100
np.random.seed(0)
A = np.random.rand(M, M)
B = np.random.rand(M, N)
Ai = linalg.inv(A)
luf = linalg.lu_factor(A)
```

```
%timeit linalg.inv(A)
%timeit np.dot(Ai, B)
%timeit linalg.lu_factor(A)
%timeit linalg.lu_solve(luf, B)
```

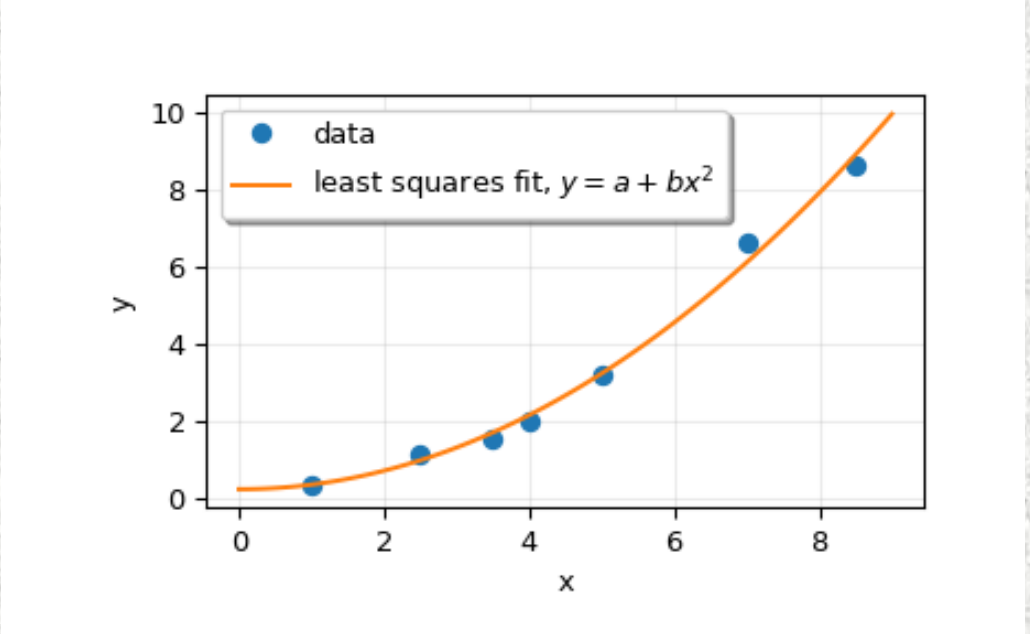
线性代数-linalg

lstsq()比solve()更一般化，它不要求矩阵是正方形的，也就是说方程的个数可以少于、等于或者多于未知数的个数。它找到一组解，使得方程误差的平方和为最小。

```
import numpy as np
from scipy.linalg import lstsq
import matplotlib.pyplot as plt

x = np.array([1, 2.5, 3.5, 4, 5, 7, 8.5])
y = np.array([0.3, 1.1, 1.5, 2.0, 3.2, 6.6, 8.6])

M = x[:, np.newaxis]**[0, 2]    # y = a + bx^2
p, res, rnk, s = lstsq(M, y)

plt.plot(x, y, 'o', label='data')
xx = np.linspace(0, 9, 101)
```

线性代数-linalg

```
yy = p[0] + p[1]*xx**2
plt.plot(xx, yy, label='least squares fit, $y = a + bx^2$')
plt.xlabel('x')
plt.ylabel('y')
plt.legend(framealpha=1, shadow=True)
plt.grid(alpha=0.25)
plt.show()
```



线性代数-linalg

- 小结：
  - 求逆矩阵：linalg.inv()
  - 求行列式的值：linalg.det（）
  - 求模：linalg.norm（）
  - 求解线性方程组：linalg.solve（a,b）
  - 求超定方程的最小二乘解：
    x,resid,rank,sigma =linalg.lstsq（a,b） #x为解
  - 广义逆：linalg.pinv()or linalg.pinv2()
    inv函数只接受方阵作为输入矩阵，而pinv函数则没有这个限制。

线性代数-linalg

- 求特征值和特征向量：

linalg.eig(A)　返回矩阵的特征值与特征向量

linalg.eigvals(A)返回矩阵的特征值

linalg.eig(A, B)求解 Av=λBv 的问题

- Decompositions分解

LU decomposition　lu()

Cholesky decomposition　cholesky()

QR decomposition　qr()

Schur decomposition　schur()

奇异值分解Singular value decomposition

linalg.svd()

优化—optimize

- 非线性方程组求解

optimize模块中的fsolve()可以对非线性方程组进行求解，它的基本调用形式如下：

x=fsolve(func, x0)

func是计算方程组误差的函数，它的参数x是一个数组，其值为方程组的一组可能的解。func返回将x代入方程组之后得到的每个方程的误差，x0为未知数的一组初始值。假设要对下面的方程组进行求解.

$$f1(u1,u2,u3)=0,\ f2(u1,u2,u3)=0,$$
$$f3(u1,u2,u3)=0$$

优化—optimize

那么func可以如下定义：

```
def func(x):
    u1,u2,u3 = x
    return [f1(u1,u2,u3), f2(u1,u2,u3), f3(u1,u2,u3)]
```

使用fsolve求非线性方程组的解，方程如下：(scipy_fsolve.py)

$$5x_1 + 3 = 0, \quad 4x_0^2 - 2\sin(x_1 x_2) = 0, x_1 x_2 - 1.5 = 0$$

优化—optimize

```
from scipy.optimize import fsolve
from math import sin

def f(x):
        x0, x1, x2 = x.tolist()
        return [
                5*x1+3,
                4*x0*x0 - 2*sin(x1*x2),
                x1*x2 - 1.5
        ]

# f计算方程组的误差，[1,1,1]是未知数的初始值
result = fsolve(f, [1,1,1])
print result
print f(result)
```

优化—optimize

输出为：

```
[-0.70622057 -0.6 -2.5 ]
[0.0, -9.126033262418786e-14, 5.329070518200751e-15]
```

由于fsolve函数在调用函数f时，传递的参数为数组，因此如果直接使用数组中的元素计算的话，计算速度将会有所降低，因此这里先用tolist()将数组中的元素转换为Python中的标准浮点数，然后调用标准math库中的函数进行运算。

优化—optimize

在对方程组进行求解时，fsolve会自动计算方程组的雅可比矩阵，如果方程组中的未知数很多，而与每个方程有关的未知数较少时，即雅可比矩阵比较稀疏时，传递一个计算雅可比矩阵的函数将能大幅度提高运算速度。在一个模拟计算的程序中需要大量求解近有50个未知数的非线性方程组的解。每个方程平均与6个未知数相关，通过传递雅可比矩阵的计算函数使计算速度提高了4倍。

优化—optimize

雅可比矩阵

雅可比矩阵是一阶偏导数以一定方式排列的矩阵，它给出了可微分方程与给定点的最优线性逼近，因此类似于多元函数的导数。例如前面的函数f1,f2,f3和未知数u1,u2,u3的雅可比矩阵如下：

$$\begin{bmatrix} \frac{\partial f1}{\partial u1} & \frac{\partial f1}{\partial u2} & \frac{\partial f1}{\partial u3} \\ \frac{\partial f2}{\partial u1} & \frac{\partial f2}{\partial u2} & \frac{\partial f2}{\partial u3} \\ \frac{\partial f3}{\partial u1} & \frac{\partial f3}{\partial u2} & \frac{\partial f3}{\partial u3} \end{bmatrix}$$

优化—optimize

使用雅可比行列式求非线性方程组的解
:(scipy_fsolve_jacobian.py)

```
from scipy.optimize import fsolve
from math import sin,cos

def j(x):
        x0, x1, x2 = x.tolist()
        return [
                [0, 5, 0],
                [8*x0, -2*x2*cos(x1*x2), -2*x1*cos(x1*x2)],
                [0, x2, x1]
        ]
result = fsolve(f, [1,1,1], fprime=j)

print result
print f(result)
```

优化—optimize

计算雅可比矩阵的函数j()和f()一样，其x参数是未知数的一组值，它计算非线性方程组在x处的雅可比矩阵。通过fprime参数将j()传递给fsolve()。由于本例中的未知数很少， 因此计算雅可比矩阵并不能显著地提高计算速度.。

优化—optimize

- 最小二乘拟合

  假设有一组实验数据(x_i，y_i)，事先知道它们之间应该满足某函数关系$y_i=f(x_i)$，通过这些已知信息，需要确定函数f的一些参数。例如，如果函数f是线性函数 $f(x)=kx+b$,那么参数 k和b就是需要确定的值。

优化—optimize

如果用p表示函数中需要确定的参数，那么目标就是找到一组p,使得下面的函数S的值最小：

$$S(p)=\sum_{i=1}^{M}[y_i-f(x_i,p)]^2$$

这种算法被称为最小二乘拟合(Least-square fitting)。在optimize模块中，可以使用leastsq()对数据进行最小二乘拟合计算。leastsq() 的用法很简单，只需要将计算误差的函数和待确定参数的初始值传递给它即可。下面是用leastsq()对线性函数进行拟合的程序。

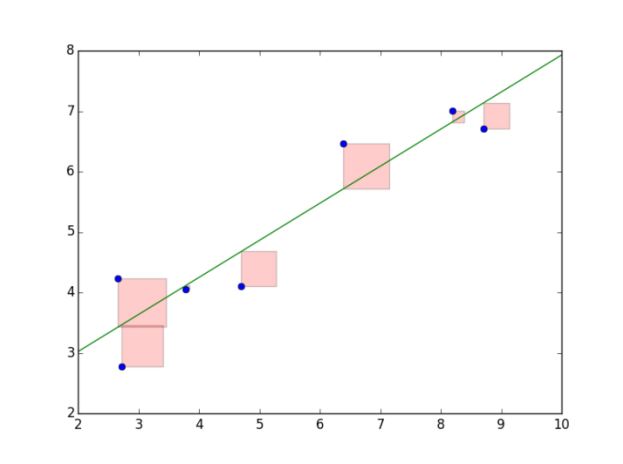
优化—optimize

用最小二乘法拟合直线，并显示误差曲面
(scipy_least_square_line.py)

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

X = np.array([ 8.19, 2.72, 6.39, 8.71, 4.7 , 2.66, 3.78])
Y = np.array([ 7.01, 2.78, 6.47, 6.71, 4.1 , 4.23, 4.05])

def residuals(p):
    "计算以p为参数的直线和原始数据之间的误差"
    k, b = p
    return Y - (k*X + b)

# leastsq使得residuals()的输出数组的平方和最小，参数的初始值为[1,0]
r = leastsq(residuals, [1, 0])
k, b = r[0]
print "k =",k, "b =",b
```

优化—optimize

```
#下面是绘图部分
import pylab as pl
from matplotlib.patches import Rectangle

pl.plot(X, Y, "o")
X0 = np.linspace(2, 10, 3)
Y0 = k*X0 + b
pl.plot(X0, Y0)

for x, y in zip(X, Y):
    y2 = k*x+b
    rect = Rectangle((x,y), abs(y-y2), y2-y, facecolor="red“,alpha=0.2)
    pl.gca().add_patch(rect)

pl.gca().set_aspect(“equal”)
#ax.set_aspect("equal")使axes per unit length相等
pl.show()
```



优化—optimize

leastsq()函数传入误差计算函数和初始值[1,0]，该初始值将作为误差计算函数的第一个参数传入；计算的结果r是一个包含两个元素的元组，第一个元素是一个数组，表示拟合后的参数k、b；第二个元素如果等于1、2、3、4中的其中一个整数，1则拟合成功，否则将会返回mesg .　　程序的输出为：

```
k = 0.613495349193 b = 1.79409254326
```

residuals()的参数p是拟合直线的参数，函数返回的是原始数据和拟合直线之间的误差。

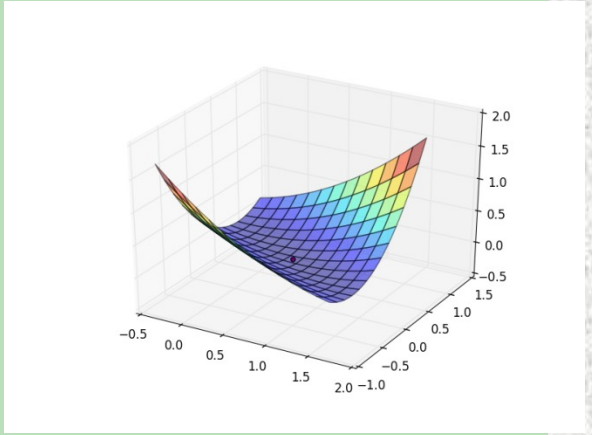
优化—optimize

对于直线拟合来说，误差的平方和是直线参数和的二次多项式函数，因此可以用下图所示的曲面直观地显示误差平方和与两个参数之间的关系。图中用灰色圆球表示曲面的最小点，它的X-Y轴的坐标就是leastsq()的拟合结果。下面的函数S()用来计算误差曲面，其参数k和b均为二维数组。

```
####### 误差曲面 #######
scale_k = 1.0
scale_b = 10.0
scale_error = 1000.0
```

优化—optimize

```
def S(k, b):
    "计算直线y=k*x+b和原始数据X、Y的误差的平方和"
    error = np.zeros(k.shape)
    for x, y in zip(X, Y):
        error += (y - (k*x + b))**2
    return error

ks, bs = np.mgrid[k-scale_k:k+scale_k:40j, b-
scale_b:b+scale_b:40j]
error = S(ks, bs)/scale_error

from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(ks, bs/scale_b, error, rstride=3, cstride=3,
cmap="jet", alpha=0.5)
ax.scatter([k],[b/scale_b],[S(k,b)/scale_error], c="r", s=20)
plt.show()
```



优化—optimize

再看一个对正弦波数据进行拟合的例子:

使用最小二乘法对带噪声的正玄波数据进行拟合(scipy_least_square_sin2.py).

```
"""
使用leastsq()对带噪声的正弦波数据进行拟合。拟合所得到的参数虽然和实际的参数完全不同，但是由于正弦函数具有周期性，如图所示，实际上拟合的结果和实际的函数是一致的。
"""
import numpy as np
from scipy.optimize import leastsq

def func(x, p):
    """
    数据拟合所用的函数: A*sin(2*pi*k*x + theta)
    """
    A, k, theta = p
    return A*np.sin(2*np.pi*k*x+theta)
```

优化—optimize

```
def residuals(p, y, x):
    """
    实验数据x, y和拟合函数之间的差，p为拟合需要找到的系数
    """
    return y - func(x, p)

x = np.linspace(0, 2*np.pi, 100)
A, k, theta = 10, 0.34, np.pi/6 # 真实数据的函数参数
y0 = func(x, [A, k, theta]) # 真实数据
# 加入噪声之后的实验数据
np.random.seed(0)
y1 = y0 + 2 * np.random.randn(len(x))

p0 = [7, 0.40, 0] # 第一次猜测的函数拟合参数
```

优化—optimize

```
# 调用leastsq进行数据拟合
# residuals为计算误差的函数
# p0为拟合参数的初始值
# args为需要拟合的实验数据
plsq = leastsq(residuals, p0, args=(y1, x))

print u"真实参数:", [A, k, theta]
print u“拟合参数”, plsq[0] # 实验数据拟合后的参数

import pylab as pl
pl.plot(x, y0, label=u"真实数据")
pl.plot(x, y1,"o", label=u"带噪声的实验数据")
pl.plot(x, func(x, plsq[0]), label=u"拟合数据")
pl.legend()
pl.show()
```

优化—optimize

程序中，要拟合的func()是一个正弦函数，它的参数p是一个数组，包含决定正弦波的三个参数：A、k、theta,分别对应正弦函数的振幅、频率和相角。一组包含噪声的数据:(x,y1),其中数组y1在标准正弦波数据y0之上添加了随机噪声。

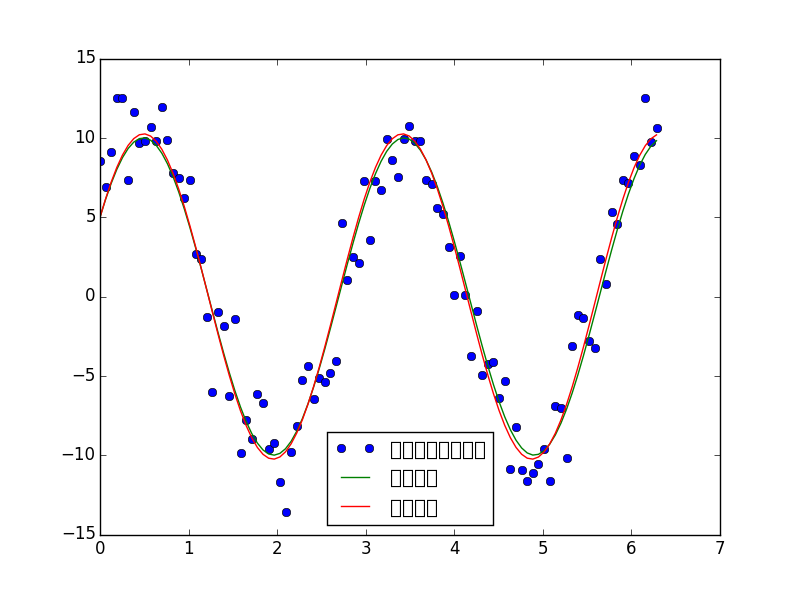
用leastsq()对带噪声的实验数据(x,y1)进行数据拟合，它可以找到数组x和真实数据y0之间的正弦关系，即确定A、k、theta等参数。这里将(y1, x)传递给args参数。Leastsq()会将这两个额外的参数传递给residuals()。因此residuals()有三个参数，p是正弦函数的参数，y和x是表示实验数据的数组。

优化—optimize

下面是程序的输出：

```
真实参数: [10, 0.34, 0.5235987755982988]
拟合参数 [ 10.25218748   0.3423992    0.50817423]
```

有时看到拟合参数虽然和真实参数完全不同，但是由于正弦函数具有周期性，实际上拟合参数得到的函数和真实参数对应的函数是一致的。



优化—optimize

对于这种一维曲线拟合,optimize库还提供了一个curve_fit()函数(scipy_least_square_sin.py)，下面使用此函数对正弦波数据进行拟合。它的目标函数与leastsq()稍有不同，各个待优化参数将直接作为函数的参数传入。

```
def func2(x, A, k, theta):
    return A*np.sin(2*np.pi*k*x+theta)

popt, _ = optimize.curve_fit(func2, x, y1, p0=p0)
print popt
```

```
[ 10.6087528    0.34063063   0.47625263]
```

优化—optimize

如果频率的初值和真实值的差别较大，拟合结果中的频率参数可能不能收敛于实际的频率。在下面的例子中，由于频率初值的选择不当，导致curve_fit()未能收敛为真实的参数。这时可以通过其它方法先估算一个频率的近似值，或者使用全局优化算法。

```
popt,pcov= optimize.curve_fit(func2, x, y1, p0=[10, 1, 0])
print u"真实参数:", [A, k, theta]
print u"拟合参数", popt
```

```
真实参数: [10, 0.34, 0.5235987755982988]
拟合参数 [ 1.05736284  1.03207993 -0.36284522]
```

优化—optimize

- 函数局域最小值

optimize模块还提供了许多求函数最小值的算法：fmin、fmin_powell、 fmin_cg、fmin_bfgs 等。下面用一个实例观察这些“fmin*()”是如何找到函数的最小值的。在本例中，要计算最小值的函数f(x,y)为：

$$f(x, y) = (1-x)^2+100(y-x^2)^2$$

为了提高运算速度和精度，有些“fmin()” 带有一个fprime参数，它是计算目标函数f对各个自变量的偏导数的函数。

优化—optimize

f(x,y)对变量x和y的偏导函数为：

$$\frac{\partial f}{\partial x} = -2 + 2x - 400x(y - x^2), \quad \frac{\partial f}{\partial y} = 200y - 200x^2$$

这个函数叫做Rosenbrock(罗森布罗克)函数，它经常用来测试最小化算法的收敛速度。它有一个十分平坦的山谷区域，收敛到此山谷区域比较容易，但是在山谷区域搜索到最小点则比较困难。根据函数的计算公式不难看出此函数的最小值是0,在(1,1)处。

下面的程序计算f(x,y)的最小值，并且绘制出f所表示的曲面和寻找最小值时的搜索路径

优化—optimize

观察fmin*函数计算最小值时的路径 (scipy_fmin_demo.py)

```
"""
使用fmin()计算函数最小值，并用matplotlib绘制搜索最小值的路径。
"""
import scipy.optimize as opt
import numpy as np
import sys

points = []
def f(p):
        x, y = p
        z = (1-x)**2 + 100*(y-x**2)**2
        points.append((x,y,z))
        return z
```

优化—optimize

```
def fprime(p):
        x, y = p
        dx = -2 + 2*x - 400*x*(y - x**2)
        dy = 200*y - 200*x**2
        return np.array([dx, dy])

init_point =(-2,-2)

try:
        method = sys.argv[1]
except:
        method = " fmin”
```

优化—optimize

```
fmin_func = opt.__dict__[method] #>>>opt.fmin_l_bfgs_b?

if method in ["fmin", "fmin_powell"]:
        result = fmin_func(f, init_point) #参数为目标函数和初始值
elif method in ["fmin_cg", "fmin_bfgs", "fmin_l_bfgs_b",
"fmin_tnc"]:
       result = fmin_func(f, init_point, fprime) #参数为目标函数、
初始值和导函数
elif method in ["fmin_cobyla"]:
        result = fmin_func(f, init_point, [])
else:
        print "fmin function not found"
        sys.exit(0)
```

优化—optimize

f()计算f(x,y)的函数值，为了记录下最小化过程中的计算轨迹，在f()中将每个计算过的点都添加进全局列表points中。fprime()计算f(x,y)对两个自变量在p处的偏导函数的值。

最小化的初值设置为(-2,-2)，此程序从optimize模块的__dict_字典中获得由命令行参数指定的最小值函数。不同的“fmin*()”参数有所不同，例如有些算法不需要fprime()。

优化—optimize

下面的程序通过求解卷积的逆运算演示fmin的功能,比较fmin, fmin_powell, fmin_cg, fmin_bfgs。

对于一个离散的线性时不变系统h, 如果它的输入是x，那么其输出y可以用x和h的卷积表示：$y = x\ (*)\ h$

已知系统的输入x和输出y，计算传递函数h；或已知系统的传递函数h和输出y，计算系统的输入x。这种运算称为反卷积运算。

fmin计算反卷积，这种方法只能用在很小规模的数列之上，不过用来评价fmin函数的性能还是不错的。（ scipy_fmin.py)

优化—optimize

```
import scipy.optimize as opt
import numpy as np

def test_fmin_convolve(fminfunc, x, h, y, yn, h0):
        #x (*) h = y, (*)表示卷积,yn为在y的基础上添加一些干扰噪声的结果,h0为求解h的初始值
        def convolve_func(h):
        #计算yn - x (*) h 的power,fmin将通过计算使得此power最小
                return np.sum((yn - np.convolve(x, h))**2)

        # 调用fmin函数，以h0为初始值
        hn = fminfunc(convolve_func, h0)

        print fminfunc.__name__
        print "--------------------"
```

优化—optimize

```
        # 输出x (*) hn 和y 之间的相对误差
        print "error of y:", np.sum((np.convolve(x, hn)-
        y)**2)/np.sum(y**2)
        # 输出hn 和h 之间的相对误差
        print "error of h:", np.sum((hn-h)**2)/np.sum(h**2)


def test_n(m, n, nscale):
"""
随机产生x, h, y, yn, h0等数列，调用各种fmin函数求解hn
m为x的长度, n为h的长度, nscale为干扰的强度
"""
        x = np.random.rand(m)
        h = np.random.rand(n)
        y = np.convolve(x, h)
        yn = y + np.random.rand(len(y)) * nscale
        h0 = np.random.rand(n)
```

优化—optimize

```
        test_fmin_convolve(opt.fmin, x, h, y, yn, h0)
        test_fmin_convolve(opt.fmin_powell, x, h, y, yn, h0)
        test_fmin_convolve(opt.fmin_cg, x, h, y, yn, h0)
        test_fmin_convolve(opt.fmin_bfgs, x, h, y, yn, h0)

if __name__ == "__main__":
test_n(200, 20, 0.1)
```

下面是程序的输出：

```
fmin
---------------------
error of y: 0.00121459580857
error of h: 0.05334854789
```

优化—optimize

fmin_powell
---
error of y: 0.000148253799473
error of h: 0.000363859459049

fmin_cg
---
error of y: 0.000147878823512
error of h: 0.000354937912181

fmin_bfgs
---
error of y: 0.000147878843851
error of h: 0.000354940012281

优化—optimize

- 计算全域最小值

前面介绍的几种最小值优化算法都只能计算局域的最小值，optimize库还提供了几种能进行全局优化的算法，以前面的正弦波拟合为例。在使用leastsq()对正弦波进行拟合时，误差函数residuals()返回一个数组，表示各个取样点的误差。而函数全域最小值算法则只能对一个标量值进行最小化，因此这里的最小化的目标函数func_error()返回所有取样点的误差的平方和。

```
def func_error(p, y, x):
        return np.sum((y - func(x, p))**2)
```

优化—optimize

使用optimize.basinhopping()全域优化函数找出正弦波的三个参数。它的前两个参数和其它求最小值的函数一样：目标函数和初始值。由于它是全局优化函数，因此初始值的选择并不是太重要。niter参数是全域优化算法的迭代次数，迭代的次数越多，就越有可能找到全域最优解。（ scipy_fmin_global.py ）

```
result = optimize.basinhopping(func_error, (1, 1, 1),
    niter = 10,
    minimizer_kwargs={"method":"L-BFGS-B",
                      "args":(y1, x)})
print result.x
```

```
[ 10.25218681  -0.34239909   2.63341581]
```

优化—optimize

```
result = optimize.basinhopping(func_error, (1, 1, 1),
    niter = 10,
    minimizer_kwargs={"method":"L-BFGS-B",
                      "args":(y1, x)})
print result.x
```

在basinhopping()内部需要调用局域最小值函数，其minimizer_kwargs参数决定了所采用的局域最小值算法以及传递给此函数的参数，下面的程序指定使用L-BFGS-B算法搜索局域最小值，并且将两个对象y1和x传递给该局域最小值求解函数的args参数，而该函数会将这两个参数传递给func_error()。

优化—optimize

虽然频率和相位和原始数据的不同，但是由于正弦函数的周期性，其拟合曲线是和原始数据重合的：

```
[ 10.25218681  -0.34239909   2.63341581]
```

```
plt.plot(x, y1, "o", label=u"带噪声的实验数据")
plt.plot(x, y0, label=u"真实数据")
plt.plot(x, func(x, result.x), label=u"拟合数据")
plt.legend(loc="best");
plt.show()
```

优化—optimize

- optimize包中函数的概览：(optimize.)
  - **非线性最优化**

    fmin -- 简单Nelder-Mead算法

    fmin_powell -- 改进型Powell法

    fmin_bfgs -- 拟Newton法

    fmin_cg -- 非线性共轭梯度法

    fmin_ncg -- 线性搜索Newton共轭梯度

    leastsq -- 最小二乘
  - **有约束的多元函数问题**

    fmin_l_bfgs_b ---使用L-BFGS-B算法

    fmin_tnc ---梯度信息

    fmin_cobyla ---线性逼近

    fmin_slsqp ---序列最小二乘法

    nnls ---解|| Ax - b ||_2 for x>=0

优化—optimize

- **全局优化**

  basinhopping

  anneal ---模拟退火算法

  brute --强力法

- **标量函数**

  fminbound

  brent

  golden

  bracket

- **拟合**

  curve_fit-- 使用非线性最小二乘法拟合

- **标量函数求根**

  brentq ---classic Brent (1973)

  brenth ---A variation on the classic Brent（1980）

  ridder ---Ridder是提出这个算法的人名

优化—optimize

bisect ---二分法

newton ---牛顿法

fixed_point

- **多维函数求根**

fsolve ---通用

broyden1 ---Broyden’s first Jacobian approximation.

broyden2 ---Broyden’s second Jacobian approximation

newton_krylov ---Krylov approximation for inverse Jacobian

anderson ---extended Anderson mixing

excitingmixing ---tuned diagonal Jacobian approximation

linearmixing ---scalar Jacobian approximation

diagbroyden ---diagonal Broyden Jacobian approximation

- **实用函数**

line_search ---找到满足强Wolfe的alpha值

check_grad ---通过和前向有限差分逼近比较检查梯度函数的正确性

插值—interpolate

插值是一种通过已知的离散数据来求未知数据的方法。与拟合不同的是，它要求曲线通过所有的已知数据。SciPy的interpolate模块提供了许多对数据进行插值运算的函数.

- 一维插值

一维数据的插值运算可以通过interp1d()完成。其调用形式如下：

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

其中：参数kind是插值类型，给出了插值的曲线的阶数，可以有如下候选值：

插值—interpolate

‘zero’、‘nearest’:阶梯插值，相当于0阶曲线。

‘slinear’、‘linear’:线性插值，用一条直线连接所有的取样点，相当于1阶曲线，‘slinear’使用扩展库中的相关函数进行计算，而‘linear’则直接使用Python编写的函数进行运算，它们的结果一样。

‘quadratic’、‘cubic’： 2阶和3阶曲线，更高阶的曲线可以直接使用整数值指定。

interp1d对象可以计算x的取值范围之内任意点的函数值。它可以像函数一样直接调用，像NumPy的ufunc函数一样能对数组中的每个元素进行计算，并返回一个新的数组。

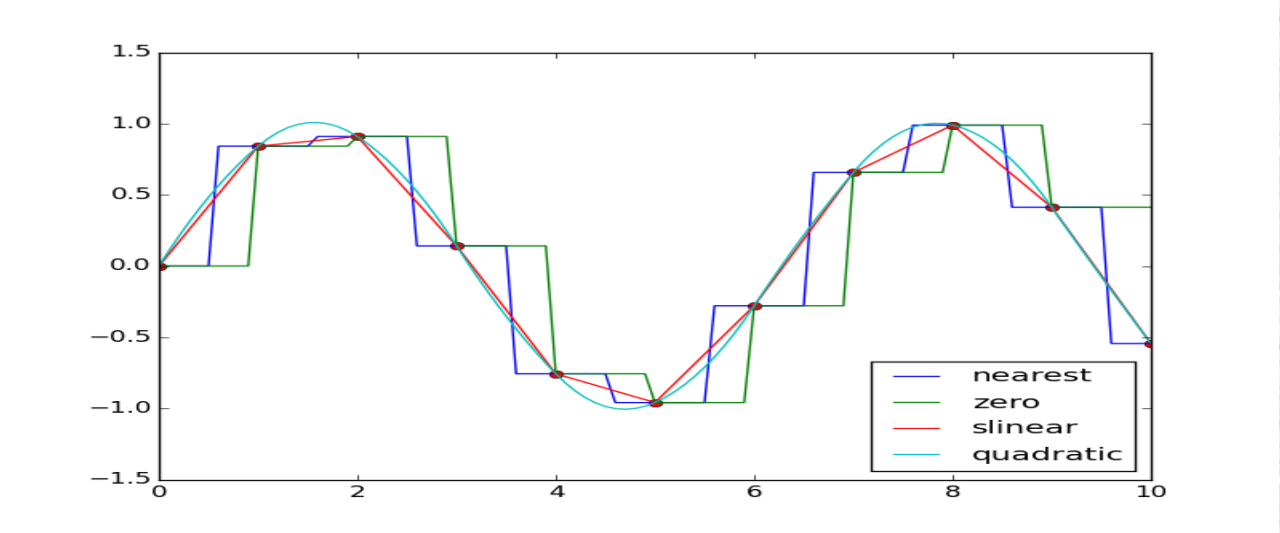
插值—interpolate

使用interp1d对数据进行各阶插值。(scipy_interp1d.py)

```
import numpy as np
from scipy import interpolate
import pylab as pl

x = np.linspace(0, 10, 11)
y = np.sin(x)

xnew = np.linspace(0, 10, 101)
pl.plot(x,y,'ro')
for kind in ['nearest', 'zero', 'slinear', 'quadratic']:
        f = interpolate.interp1d(x,y,kind=kind)
        ynew = f(xnew)
        pl.plot(xnew, ynew, label=str(kind))
```

插值—interpolate

```
pl.legend(loc='lower right')
pl.show()
```



程序中使用循环对相同的数据进行4种不同阶数的插值运算。首先使用数据点创建一个interp1d对象f,通过kind参数指定其阶数。调用f()计算出一系列的插值结果。

插值—interpolate

- 样条插值：使用上有两个基本步骤：

（1）首先要使用splrep()计算欲插值曲线的样条系数（对于N-维空间使用splprep）；

（2）在给定的点上用splev()计算样条插值结果。

tck=scipy.interpolate.splrep(*x*, *y*, *w=None*, *xb=None*, *xe=None*, *k=3*, *task=0*, *s=None*, *t=None*, *full_output=0*, *per=0*, *quiet=1*)

参数s用来确定平滑点数，通常是m-SQRT(2m),m是曲线点数。如果在插值中不需要平滑应该设定s=0，s=0时曲线经过每一个点。

插值—interpolate

splrep()输出的是一个3元素的元胞数组（t,c,k）,其中t是曲线点，c是计算出来的系数，k是样条阶数，通常是3阶，但可以通过k改变。

scipy.interpolate.splev(*x*, *tck*, *der=0*)
其中的der是进行样条计算是需要实际计算到的阶数，必须满足条件der<=k。

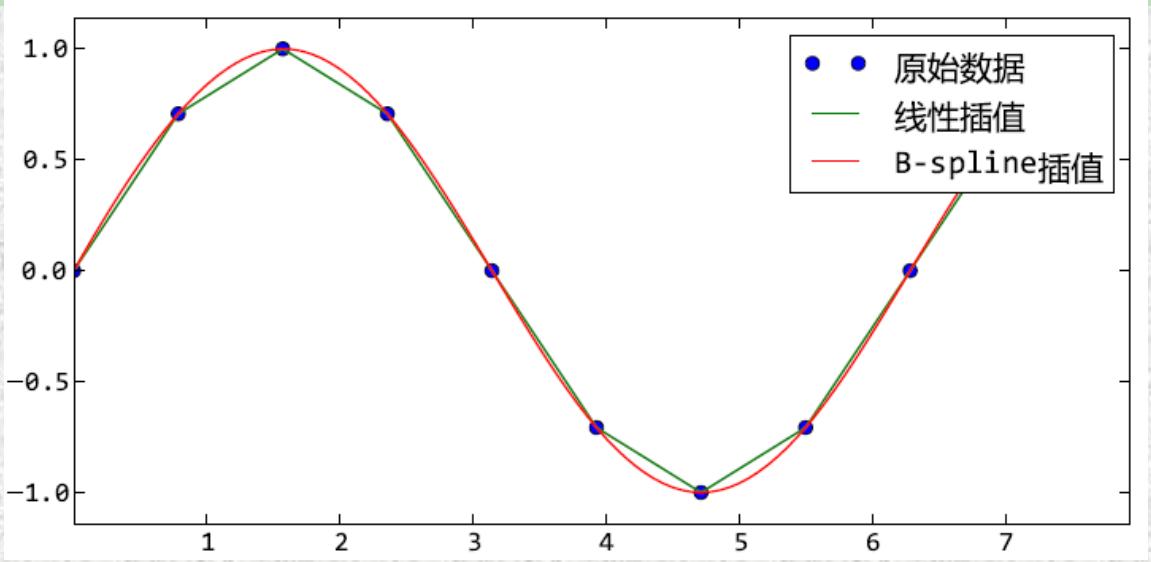
插值—interpolate

下面是使用直线和B-Spline对正弦波上的点进行插值的例子。(scipy_B_Spline)

```
import numpy as np
import pylab as pl
from scipy import interpolate

x = np.linspace(0, 2*np.pi+np.pi/4, 10)
y = np.sin(x)

x_new = np.linspace(0, 2*np.pi+np.pi/4, 100)
f_linear = interpolate.interp1d(x, y)
tck = interpolate.splrep(x, y)
y_bspline = interpolate.splev(x_new, tck)

pl.plot(x, y, "o", label=u"原始数据")
pl.plot(x_new, f_linear(x_new), label=u"线性插值")
```

插值—interpolate

```
pl.plot(x_new, y_bspline, label=u"B-spline插值")
pl.legend()
pl.show()
```



B-Spline插值运算需要先使用splrep函数计算出B-Spline曲线的参数，然后将参数传递给splev函数计算出各个取样点的插值结果。

插值—interpolate

- 外推和 Spline 拟合

前面介绍的interp1d类要求其参数x是一个递增序列,并且只能在x的取值范围之内进行内插计算，不能用它进行外推运算，即无法计算x的取值范围之外的数据点。UnivariateSpline类的插值运算比interp1d更高级，它支持外推运算，其调用形式如下：

```
UnivariateSpline(x, y, w=None, bboxs=[None, None], k=3, s=None)
```

插值—interpolate

```
UnivariateSpline(x, y, w=None, bboxs=[None, None], k=3, s=None)
```

x、y是保存数据点的X-Y坐标的数组，其中x必须是递增序列。

w是为每个数据点指定的权重值。

bboxs序列指定的近似区间的边界.

k为样条曲线的阶数。

s是平滑参数，它使得最终生成的样条曲线满足条件$\sum(w\cdot(y-spline(x)))^2\leq s$,即当 s>0 时，样条曲线并不一定通过各个数据点.为了让曲线通过所有数据点，必须将s参数设置为0.

插值—interpolate

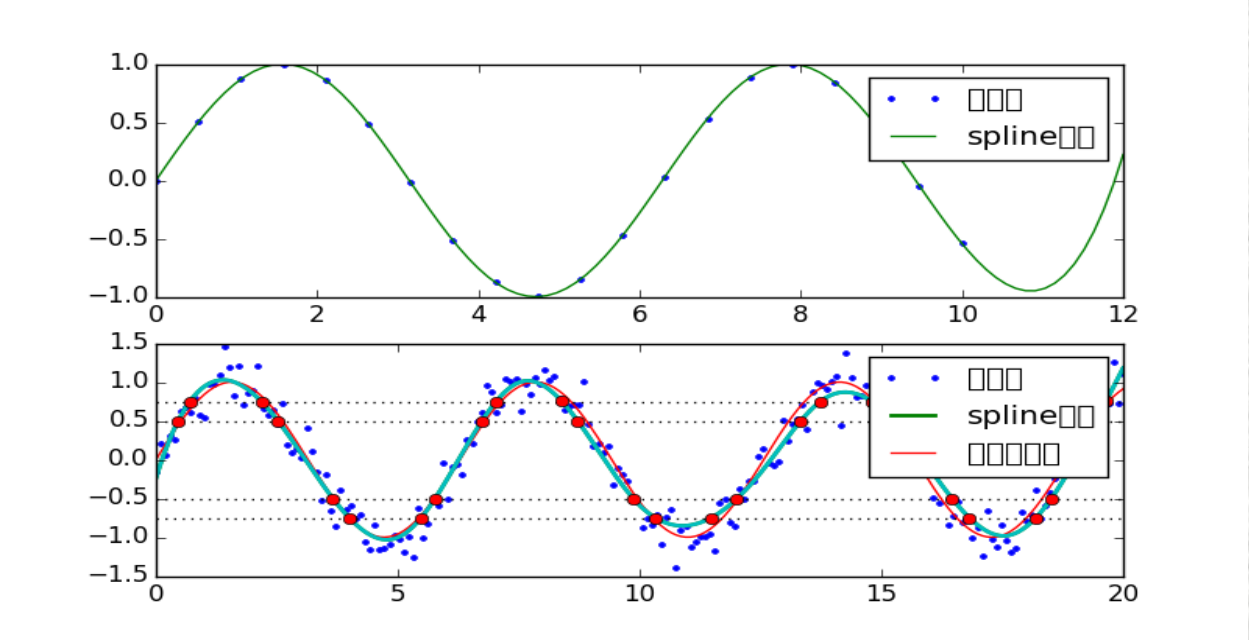
使用UnivariateSpline进行插值运算:(scipy_uspline.py)

```
import numpy as np
import pylab as pl
from scipy import interpolate

x1 = np.linspace(0, 10, 20)
y1 = np.sin(x1)
sx1 = np.linspace(0, 12, 100)
sy1 = interpolate.UnivariateSpline(x1, y1, s=0)(sx1)

x2 = np.linspace(0, 20, 200)
y2 = np.sin(x2) +
np.random.standard_normal(len(x2))*0.2
sx2 = np.linspace(0, 20, 2000)
sy2 = interpolate.UnivariateSpline(x2, y2, s=8)(sx2)
```

插值—interpolate

```
pl.figure(figsize=(8, 4))
pl.subplot(211)
pl.plot(x1, y1, ".", label=u"数据点")
pl.plot(sx1, sy1, label=u"spline曲线")
pl.legend()


pl.subplot(212)
pl.plot(x2, y2, ".", label=u"数据点")
pl.plot(sx2, sy2, linewidth=2, label=u"spline曲线")
pl.plot(x2, np.sin(x2), label=u"无噪声曲线")
pl.legend()
pl.show()
```

插值—interpolate



在x轴在大于10的样条曲线仍然呈现出正弦波类似的形状。对于带噪声的输入数据选择合适的s参数能够使得样条曲线接近无噪声时的波形，可以把它看作使用样条曲线对数据进行拟合运算。

插值—interpolate

- 参数插值

前面所介绍的插值函数都需要X轴的数据是按照递增顺序排列的，就像一般的函数曲线一样。数学上还有一种参数曲线，它使用一个参数和两个函数，定义二维平面上的一条曲线，例如圆形、心形等曲线都是参数曲线。参数曲线的插值可以通过splprep()和splev()实现，这组函数支持高维空间的曲线的插值，这里以二维曲线为例，介绍其用法。

x和y的数值由参数t控制。x=f(t),y=g(t)

（scipy_interpolate_spl.py）

插值—interpolate

```
import numpy as np
from matplotlib import pyplot as plt
from scipy import interpolate

x = np.random.rand(10)
y = np.random.rand(10)
np.random.shuffle(y)

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

for s in (0, 1e-4):
        tck, t = interpolate.splprep([x, y], s=s)
```

#首先调用splprep()，其第一个参数为一组一维数组，每个数组是各点在对应轴上的坐标,s为平滑系数。splprep()返回两个对象，其中tck是一个元组，它包含了插值曲线的所有信息，t是自动计算出参数曲线的参数数组。

插值—interpolate

```
        xi,yi=interpolate.splev(np.linspace(t[0],t[-
1],200),tck)

#调用splev()进行插值运算，其第一个参数为一个新的参数数组，这里将t的取值范围等分200份，第二个参数为splprep()返回的第一个对象。实际上，参数数组t是正规化之后的各个线段长度的累计，因此t的范围位0到1。

        if s == 0:
                plt.plot(xi, yi, lw=1, label=u"s=%g" %s)
        else:
                plt.plot(xi, yi, lw=3, alpha=0.4,
label=u"s=%g" %s)
plt.legend()
plt.show()
```

插值—interpolate

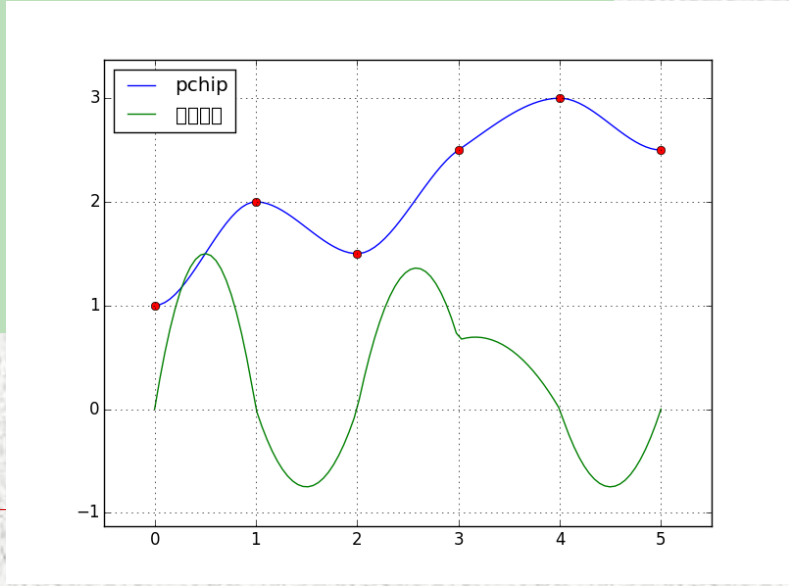
- 单调插值pchip():

  前面介绍的几种插值方法，不能保证数据点的单调性，即曲线的最值可能出现在数据点之外的地方。PchipInterpolator类(别名pchip)使用单调三次插值，能够保证曲线的所有最值都出现在数据点之上。下面的程序用pchip()对数据点进行插值，并绘制其一阶导数曲线，由下图的导数曲线可知，所有最值数据点出的导数都为0。

```
import numpy as np
from matplotlib import pyplot as plt
from scipy import interpolate
```

插值—interpolate

```
x = [0, 1, 2, 3, 4, 5]
y = [1, 2, 1.5, 2.5, 3, 2.5]
xs = np.linspace(x[0], x[-1], 100)
curve = interpolate.pchip(x, y)
ys = curve(xs)
dcurve = curve.derivative()
dys = dcurve(xs)
plt.plot(xs, ys, label=u"pchip")
plt.plot(xs, dys, label=u"一阶导数")
plt.plot(x, y, "o")
plt.legend(loc="best")
plt.grid()
plt.margins(0.1, 0.1)
plt.show()
```



插值—interpolate

- 二维插值

使用interp2d()可以进行二维插值运算，它的调用形式如下：

```
interp2d(x, y, z, kind='linear',...)
```

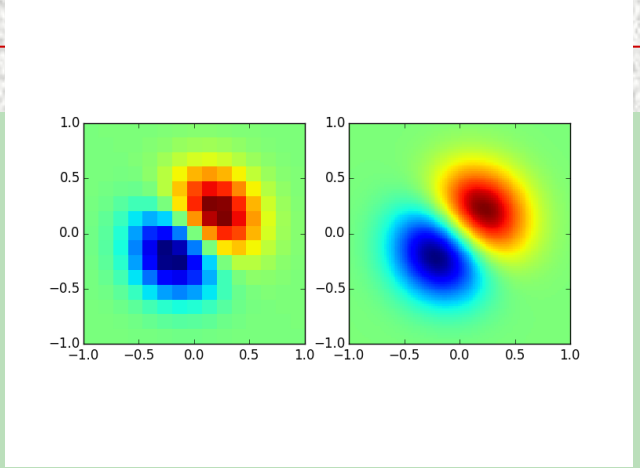
其中：kind 参数指定插值运算的阶数，可以为'linear'、'cubic'或'quintic'。

使用interp2d函数进行二维插值.(scipy_interp2d.py)

插值—interpolate

```
"""
演示二维插值。
"\"
import numpy as np
from scipy import interpolate
import pylab as pl

def func(x, y):
        return (x+y)*np.exp(-5.0*(x**2 + y**2))

# X-Y轴分为15*15的网格
y, x = np.mgrid[-1:1:15j, -1:1:15j]
fvals = func(x,y) # 计算每个网格点上的函数值

# 二维插值
newfunc = interpolate.interp2d(x, y, fvals, kind='cubic')
```

插值—interpolate




```
# 计算100*100的网格上的插值
xnew = np.linspace(-1,1,100)
ynew = np.linspace(-1,1,100)
fnew = newfunc(xnew, ynew)

# 绘图
# 为了更明显地比较插值前后的区别，使用关键字参数
#interpolation='nearest'
# 关闭imshow()内置的插值运算。
pl.subplot(121)
pl.imshow(fvals, extent=[-1,1,-1,1], cmap=pl.cm.jet,
interpolation='nearest', origin="lower")
pl.subplot(122)
pl.imshow(fnew, extent=[-1,1,-1,1], cmap=pl.cm.jet,
interpolation='nearest', origin="lower")
pl.show()
```

插值—interpolate

func是计算曲面上各点高度的函数。所得到的二维数组fvals的第0轴与Y轴对应，第一轴与X轴对应 。interp2d 对象可以像函数一样调用，用它计算插值曲面在一个更密的网格中的高度值。 fnew这里计算的参数是两个一维数组，分别指定网格的X-Y轴坐标，而不需要通过mgrid创建网格坐标数组。

interp2d只能对网格形状的取样值进行插值运算，如果需要对随机散列的取样点进行插值，就需要使用griddata()和径向基函数(Radial Basis Function,简称RBF)插值算法。RBF支持多维散列点的插值运算。

插值—interpolate

使用griddata()对随机取样点进行二维插值。

```
import numpy as np
from scipy.interpolate import griddata

ripple = lambda x,y:np.sqrt(x**2 + y**2) + np.sin(x**2 + y**2)

grid_x, grid_y = np.mgrid[0:5:1000j, 0:5:1000j]

xy = np.random.rand(100, 2)

sample = ripple(xy[:,0] * 5 , xy[:,1] * 5)

grid_z0 = griddata(xy * 5, sample, (grid_x, grid_y), method='cubic')
```

插值—interpolate

使用RBF对随机取样点进行二维插值。(scipy_rbf.py)

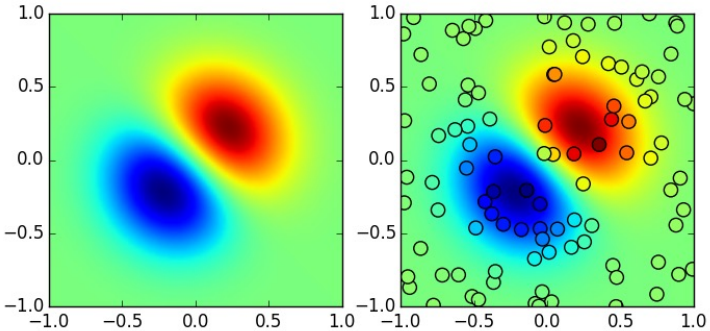
```
import numpy as np
from scipy import interpolate
import pylab as pl

def func(x,y):
        return (x+y)*np.exp(-5.0*(x**2 + y**2))

# 计算曲面函数上100个随机分布的点
x = np.random.uniform(-1.0, 1.0, size=100)
y = np.random.uniform(-1.0, 1.0, size=100)
fvals = func(x,y)
```

插值—interpolate

```
# 使用Rbf进行插值运算
newfunc = interpolate.Rbf(x, y, fvals,
function='multiquadric')
ynew, xnew = np.mgrid[-1:1:100j, -1:1:100j] # 插值结果的网格
fnew = newfunc(xnew, ynew)
truevals = func(xnew, ynew) # 函数的真实值

pl.subplot(121)
pl.imshow(truevals,extent=[-1,1,-1,1], cmap=pl.cm.jet,
origin="lower")
pl.subplot(122)
pl.scatter(x,y,20,fvals,cmap=pl.cm.jet)
pl.imshow(fnew,extent=[-1,1,-1,1], cmap=pl.cm.jet,
origin="lower")
pl.show()
```

插值—interpolate

使用随机点创建一个RBF对象，并通过function参数指定所使用的径向基函数。RBF对象也可以像函数那样被调用，用它计算更密的网格上各点的值。它的两个参数是指定X-Y轴坐标的两个数组。和interp2d对象不同的是，它不会自动产生网格上的各点，因此为了使用等距的正交网格，使用mgird对象创建这两个数组。



插值—interpolate

- 二维插值的三维展示方法

```
# -*- coding: utf-8 -*-
"""
演示二维插值。
"""
# -*- coding: utf-8 -*-
import numpy as np
from mpl_toolkits.mplot3d import Axes3D
import matplotlib as mpl
from scipy import interpolate
import matplotlib.cm as cm
import matplotlib.pyplot as plt

def func(x, y):
        return (x+y)*np.exp(-5.0*(x**2 + y**2))
```

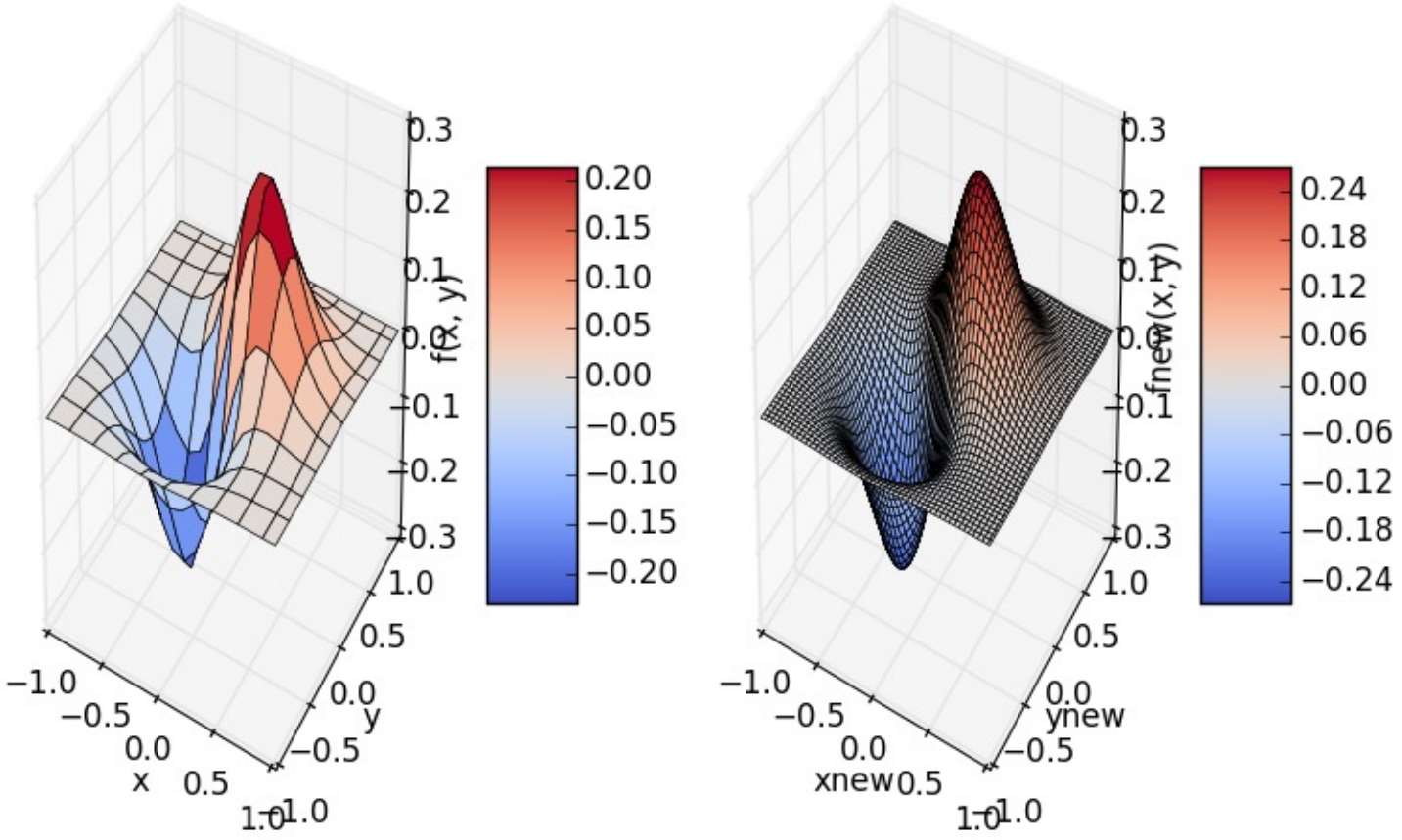
插值—interpolate

```
# X-Y轴分为20*20的网格
x = np.linspace(-1, 1, 20)
y = np.linspace(-1,1,20)
x, y = np.meshgrid(x, y)#20*20的网格数据
fvals = func(x,y) # 计算每个网格点上的函数值 20*20的值

fig = plt.figure(figsize=(9, 6))
#Draw sub-graph1
ax=plt.subplot(1, 2, 1,projection = '3d')
surf = ax.plot_surface(x, y, fvals, rstride=2, cstride=2,
cmap=cm.coolwarm,linewidth=0.5, antialiased=True)
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('f(x, y)')
plt.colorbar(surf, shrink=0.5, aspect=5)#标注
```

插值—interpolate

```
#二维插值
newfunc = interpolate.interp2d(x, y, fvals, kind='cubic')
#newfunc为一个函数
# 计算100*100的网格上的插值
xnew = np.linspace(-1,1,100)#x
ynew = np.linspace(-1,1,100)#y
fnew = newfunc(xnew, ynew)#np.shape(fnew) is 100*100
xnew, ynew = np.meshgrid(xnew, ynew)
ax2=plt.subplot(1, 2, 2,projection = '3d')
surf2 = ax2.plot_surface(xnew, ynew, fnew, rstride=2,
cstride=2, cmap=cm.coolwarm,linewidth=0.5,
antialiased=True)
ax2.set_xlabel('xnew')
ax2.set_ylabel('ynew')
ax2.set_zlabel('fnew(x, y)')
plt.colorbar(surf2, shrink=0.5, aspect=5)#标注
plt.show()
```

插值—interpolate



数值积分—integrate

SciPy的integrate模块提供了几种数值积分算法，其中包括对常微分方程组(ODE)的数值积分。本节以计算球体体积和洛伦茨吸引子轨迹为例介绍integrate模块的用法。

- 球的体积

数值积分是对定积分的数值求解，例如可以利用数值积分计算某个形状的面积。先考虑一下如何计算半径为1的半圆的面积。根据圆的面积公式，其面积应该等于π/2。单位半圆的曲线方程为 $y=\sqrt{1-x^2}$,可以通过下面的half_circle()进行计算.用数值积分求圆的面积和球的体积(scipy_integrate.py)

数值积分—integrate

```
def half_circle(x):
        return (1-x**2)**0.5
```

最简单的数值积分算法就是将要积分的面积分为许多小矩形，然后计算这些矩形的面积之和。下面使用这种方法，将X轴上-1到1的区间分为10000等份，然后计算面积和：

```
>>> N = 10000
>>> x = np.linspace(-1, 1, N)
>>> dx=x[1] - x[0]
>>> y = half_circle(x)
>>> 2 * dx *np.sum(y) # 面积的两倍
3.1415893269307378
```

数值积分—integrate

也可以用NumPy的trapz()计算半圆上由各点构成的多边形的面积：

```
>>> np.trapz(y, x) * 2   # 面积的两倍
3.1415893269316042
```

trapz()计算的是以(x,y)为顶点坐标的折线与X轴所夹的面积。如果使用SciPy的integrate模块中的数值积分函数quad(),将能得到非常精确的结果：

```
>>> from scipy import integrate
>>> pi_half, err =integrate.quad (half_circle,-1, 1)
>>> pi_half*2
3.1415926535897984
```

数值积分—integrate

计算多重定积分可以通过多次调用quad()来实现，为了调用方便，integrate模块提供了dblquad()以进行二重定积分，以及tplquad()用于进行三重定积分.下面以计算单位半球体积为例，说明dblquad()的用法。

单位半球面上的点(x,y,z)满足如下方程：$x^2+y^2+z^2 = 1$

因此下面的half_sphere()可以通过X-Y轴坐标计算球面上点的Z轴坐标值：

```
def half_sphere(x, y):
        return (1-x**2-y**2)**0.5
```

数值积分—integrate

X-Y轴平面与此球体的交线为一个单位圆，因此二重积分的计算区间为此单位圆。即对于X轴从-1到1进行积分，而对于Y轴则从-half_circle(x)到half_circle(x)进行积分。因此半球体积的二重积分公式为：

$$\int_{-1}^{1}\int_{-\sqrt{1-x^2}}^{\sqrt{1-x^2}}\sqrt{1-x^2-y^2}\,dydx$$

下面的程序使用dblquad()计算半球体积：

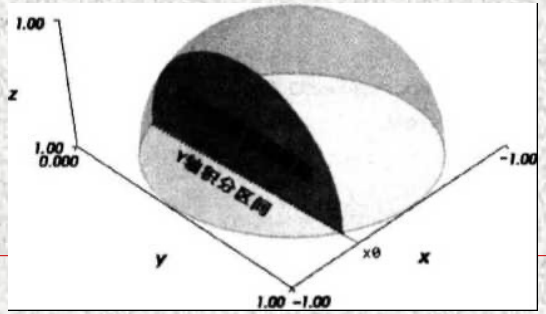
```
>>> integrate.dblquad( half_sphere, -1, 1, lambda x:-half_circle(x), lambda x:half_circle(x))
(2.094395102393198, 2.3252456653390915e-14)
>>> np.pi*4/3/2 #通过球体体积公式计算半球体积
2.094395102393195
```

数值积分—integrate

dblquad()的调用参数为：

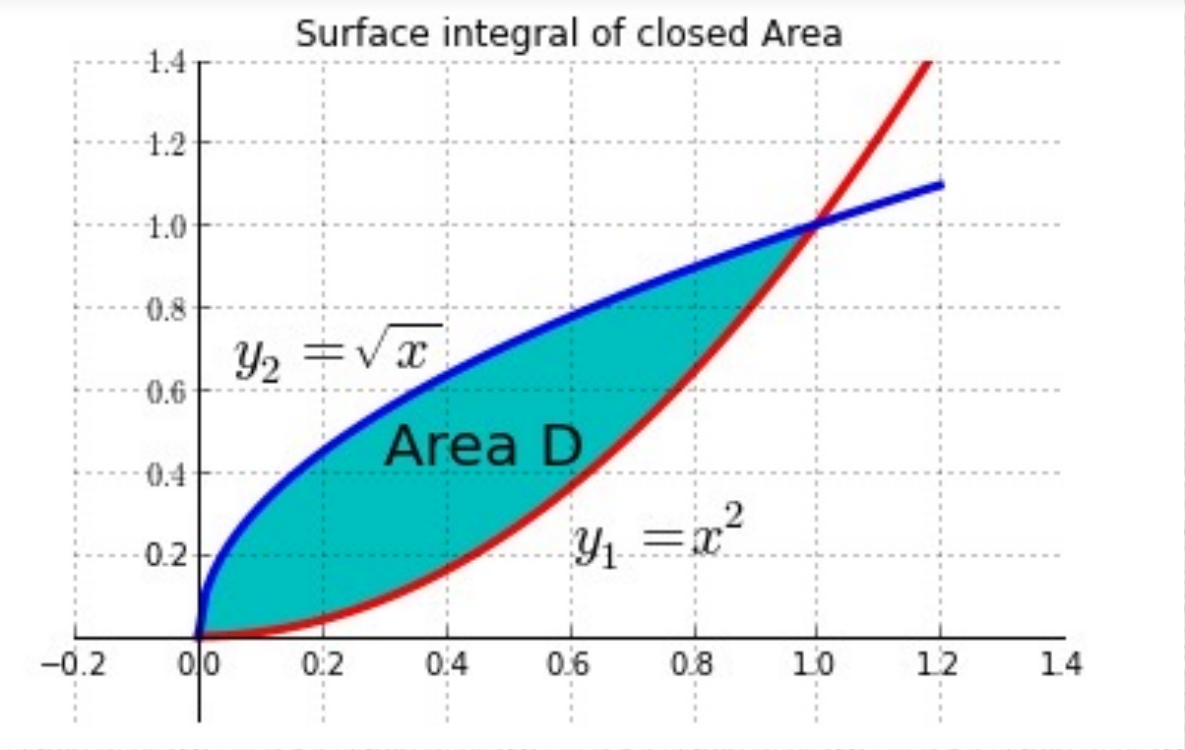
```
dblquad(func2d, a, b, gfun, hfun)
```

其中，func2d是需要进行二重积分的函数，它有两个参数，假设分别为x和y。a和b参数指定被积分函数的第一个变量(即x)的积分区间，而gfun和hfun参数指定第二个变量(即y)的积分区间。gfun和hfun是函数，它们通过变量x计算出变量y的积分区间，这样可以在X-Y平面上的任何区间对func2d进行积分。



.show()

数值积分—integrate



$$S_D = \int_0^1 dx \int_{x^2}^{\sqrt{x}} dy = \int_0^1 dy \int_{y^2}^{\sqrt{y}} dx = \int_0^1 \sqrt{x} - x^2 dx = \frac{1}{3}$$

数值积分—integrate

$$S_D = \int_0^1 dx \int_{x^2}^{\sqrt{x}} dy = \int_0^1 dy \int_{y^2}^{\sqrt{y}} dx = \int_0^1 \sqrt{x} - x^2 dx = \frac{1}{3}$$

```
s1,abser1 = integrate.quad(lambda x:sqrt(x)-x**2,0,1)
print("Area of D is %.10f" %s1)

Area of D is 0.3333333333
```

```
s2,abser2 = integrate.dblquad(lambda x,y: 1,0,1,lambda
x :x**2,lambda x:sqrt(x))
print("Area of D is %.10f" %s2)

Area of D is 0.3333333333
```

数值积分—integrate

- 解常微分方程组

integrate模块还提供了对常微分方程组进行积分的函数odeint()。下面看看如何用它计算洛伦茨吸引子的轨迹。洛伦茨吸引子由下面的三个微分方程定义：

$$\frac{dx}{dt}=\sigma\cdot(y-x),\qquad \frac{dy}{dt}=x\cdot(\rho-z)-y,\qquad \frac{dz}{dt}=xy-\beta z$$

这三个方程定义了三维空间中各个坐标点上的速度矢量。从某个坐标开始沿着速度矢量进行积分，就可以计算出无质量点在此空间中的运动轨迹。其中，σ、ρ、β为三个常数.不同的参数可以计算出不同的运动轨迹：x(t)、y(t)、z(t)。

数值积分—integrate

当参数为某些值时，轨迹出现混沌现象。即微小的初值差别也会显著地影响运动轨迹。下面是洛伦茨吸引子的轨迹计算和绘制程序：(scipy_odeint_lorenz)

```
from scipy.integrate import odeint
import numpy as np

def lorenz(w, t, p, r, b):
        # 给出位置矢量w（初值），和三个参数p, r, b计算出
        # dx/dt, dy/dt, dz/dt的值
        x, y, z = w.tolist()
        # 直接与lorenz的计算公式对应
        return p*(y-x), x*(r-z)-y, x*y-b*z
```

数值积分—integrate

```
t = np.arange(0, 30, 0.01) # 创建时间点
# 调用ode对lorenz进行求解, 用两个不同的初始值
track1 = odeint(lorenz, (0.0, 1.00, 0.0), t, args=(10.0,
28.0, 3.0))
track2 = odeint(lorenz, (0.0, 1.01, 0.0), t, args=(10.0,
28.0, 3.0))

# 绘图
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt

fig = plt.figure()
ax = Axes3D(fig)
ax.plot(track1[:,0], track1[:,1], track1[:,2])
ax.plot(track2[:,0], track2[:,1], track2[:,2])
plt.show()
```

数值积分—integrate

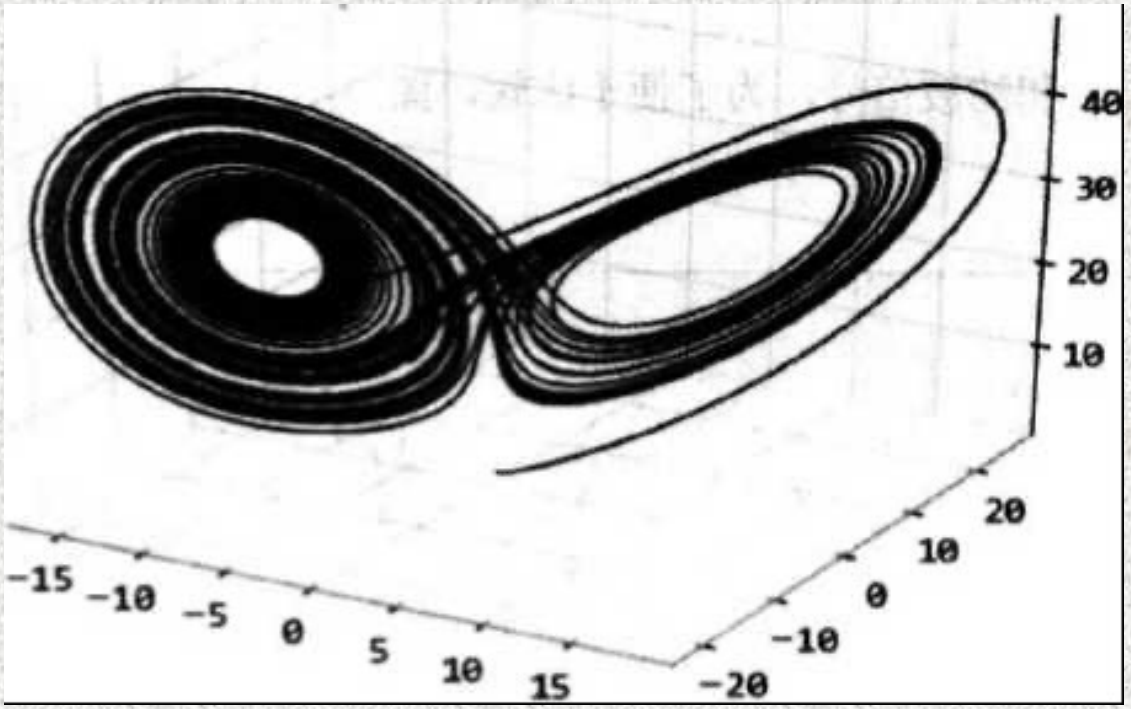
程序中首先定义一个函数lorenz(),它的任务是计算出某个坐标点在各个方向上的微分值，可以直接根据洛伦茨吸引子的公式得出。

使用不同的位移初始值两次调用odeint(),对微分方程求解。odeint()有许多参数分别为：

- lorenz:它是计算某个位置上各个方向的速度的函数。
- (0.0,1.0,0.0):位置初始值，它是计算常微分方程所需的各个变量的初始值。
- t:表示时间的数组，odeint()对此数组中的每个时间点进行求解，得出所有时间点的位置。
- args:这些参数直接传递给lorenz(),因此它们在整个积分过程中都是常量。

数值积分—integrate

最后通过matplotlib的三维绘图模块绘制出odeint()后得到的轨迹。即使初始值只相差0.01，两条运动轨迹也是完全不同的。



数值积分—integrate

- Examples

theta''(t) + b*theta'(t) + c*sin(theta(t)) = 0

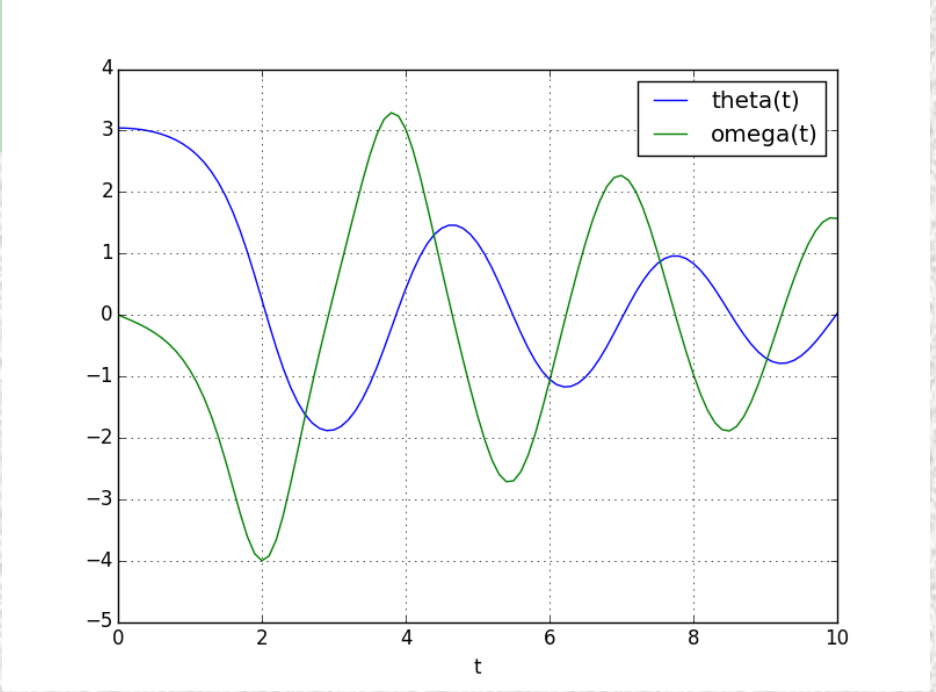
引入变量变换　theta'(t) = omega(t)

omega‘(t) = -b*omega(t) - c*sin(theta(t))

```
>>> def pend(y, t, b, c):
        theta, omega = y
        dydt = [omega, -b*omega - c*np.sin(theta)]
        return dydt
>>> b = 0.25   #参数
>>> c = 5.0
>>> y0 = [np.pi - 0.1, 0.0]  #初值
>>> t = np.linspace(0, 10, 101)
>>> from scipy.integrate import odeint
>>> sol = odeint(pend, y0, t, args=(b, c))
```

数值积分—integrate

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(t, sol[:, 0], 'b', label='theta(t)')
>>> plt.plot(t, sol[:, 1], 'g', label='omega(t)')
>>> plt.legend(loc='best')
>>> plt.xlabel('t')
>>> plt.grid()
>>> plt.show()
```



统计—stats

Scipy的stats模块包含了多种概率分布的随机变量，随机变量分为连续的和离散的两种。 所有的连续随机变量都是rv_continuous的派生类的对象，而所有的离散随机变量都是rv_discrete的派生类的对象。

- 连续和离散概率分布

可以使用下面的语句获得stats模块中所有的连续随机变量：

```
>>>from scipy import stats
>>> [k for k,v in stats.__dict__.items() if
isinstance(v,stats.rv_continuous)]
['genhalflogistic',' triang',' rayleigh','betaprime',...]
```

统计—stats

连续随机变量对象都有如下方法：

- rvs:对随机变量进行随机取值，可以通过size参数指定输出的数组大小。
- pdf:随机变量的概率密度函数。
- cdf:随机变量的累积分布函数，它是概率密度函数的积分。
- sf:随机变量的生存函数，它的值是1-cdf(t)。
- ppf:累积分布函数的反函数。
- stats:计算随机变量的期望值和方差。
- fit：对一组随机取样进行拟合，找出最适合取样数据的概率密度函数的系数。

统计—stats

下面以正态分布为例，简单介绍随机变量的用法。获得默认正态分布的随机变量的期望值和方差，默认情况下它是一个均值为0、方差为1的随机变量：

```
>>> stats.norm.stats()
(array(0.0), array(1.0))
```

norm可以像函数那样来调用，通过loc和scale参数可以指定随机变量的偏移和缩放参数。 对于正态分布的随机变量来说，这两个参数相当于指定其期望值和标准差：

```
>>> X =stats.norm(loc=1.0,scale=2.0)
>>> X.stats()
(array(1.0), array(4.0)
```

统计—stats

下面调用随机变量X的rvs()方法，得到包含一万次随机取样值的数组x:然后调用NumPy的mean()和var(),计算此数组的均值和方差，其结果符合随机变量x的特性：

```
>>> x =X.rvs(size=10000) # 对随机变量取 10000个值
>>> np.mean(x) # 期望值
1.0181259658732724
>>> np.var(x) # 方差
4.00188661646059
```

也可以使用fit()方法对随机取样序列x进行拟合，返回的是与随机取样值最吻合的随机变量的参数：

```
>>> stats.norm.fit(x) #得到随机序列的期望值和标准差
array([ 1.01810091, 2.00046946])
```

统计—stats

接下来比较随机变量X的概率密度函数和对数组x进行直方图统计的结果：

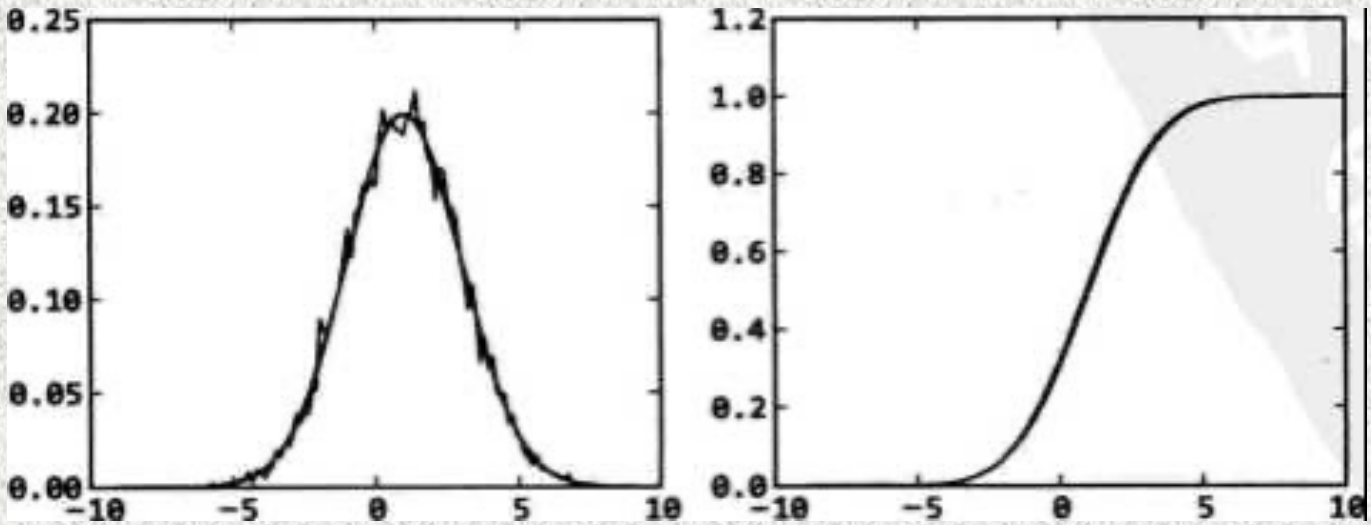
```
>>> t = np.arange(-10, 10, 0.01)
>>> pl.plot(t, X.pdf(t)) #绘制概率密度函数的理论值
>>> p, t2 =np.histogram(x, bins=100, normed=True)
>>> t2 = (t2[:-1] + t2[1:])/2
>>> pl.plot(t2, p) #绘制统计所得到的概率密度
```

其中，histogram()对数组x进行直方图统计。histogram()返回两个数组p和t2，其中p表示各个区间取样值出现的频数，由于normed参数为True,因此p的值是正规化之后的结果。t2表示区间，由于其中包括区间起点和终点，因此t2的长度为101。

统计—stats

下面的程序绘制随机变量X的累积分布函数和数组p的累加结果。

```
>>> pl.plot(t, X.cdf(t))
>>> pl.plot(t2, np.add.accumulate(p)*(t2[1]-t2[0]))
```



统计—stats

有些随机分布除了loc和scale参数之外，还需要额外的形状参数。例如伽玛分布可用于描述等待k个独立的随机事件发生所需的时间，k就是伽玛分布的形状参数。下面计算形状参数k为1和2时伽玛分布的期望值和方差：

```
>>> stats.gamma.stats(1.0)
(array(1.0), array(1.0))
>>> stats.gamma.stats(2.0)
(array(2.0) ,array(2.0))
```

伽玛分布的尺度参数θ和随机事件发生的频率相关，由scale参数指定：

```
>>> stats.gamma.stats(2.0,scale=2)
(array(4.0), array(8.0))
```

统计—stats

根据伽玛分布的数学定义可知其期望值为 $k\theta$,方差为 $k\theta^2$。上面的程序验证了这两个公式。 当随机分布有额外的形状参数时，它所对应的rvs()、pdf()等方法都会增加额外的参数以接收形状参数。例如下面的程序调用rvs()对k=2, $\theta = 2$ 的伽玛分布取4个随机值：

```
>>> x =stats.gamma.rvs(2, scale=2,size=4)
>>> x
array([ 2.20814048, 3.56652153, 4.30088176,
0.68262888])
```

统计—stats

接下来调用pdf(),查看上面4个随机值所对应的概率密度：

```
>>> stats.gamma.pdf(x, 2, scale=2)
array([ 0.18301012, 0.1498734 , 0.12519094,
0.12130919])
```

也可以先创建将形状参数和尺度参数固定的随机变量，然后再调用其pdf()计算概率密度：

```
>>> X = stats.gamma(2, scale=2)
>>> X.pdf(x)
array([ 0.18301012, 0.1498734 , 0.12519094,
0.12130919])
```

统计—stats

当分布函数的值域为离散时，称之为离散概率分布。例如投掷有6个面的骰子时，只能获得1到6的整数，因此得到的概率分布为离散的。对于离散随机分布，通常使用概率质量函数(PMF)描述其分布情况。

在stats库中所有描述离散分布的随机变量都从rv_discrete类继承。也可以直接用rv_discrete 类自定义离散概率分布。例如假设有一个不均匀的骰子，各点出现的概率不相等。可以用下面的数组x保存骰子的所有可能值，数组p保存每个值出现的概率：

统计—stats

```
>>> x = range(1,7)
>>> p = (0.4, 0.2, 0.1, 0.1, 0.1, 0.1)
```

于是，可以用下面的语句定义表示这个特殊骰子的随机变量，并调用其rvs()方法投掷此骰子20次，获得符合概率p的随机数：

```
>>> dice = stats.rv_discrete(values=(x,p))
>>> dice.rvs(size=20)
Array([2, 5, 1, 2, 1, 1, 2, 4, 1, 3, 1, 1, 4, 3, 1, 1, 1,
2, 6, 4])
```

统计—stats

- 二项、泊松、伽玛分布

本节用几个实例程序对概率论中的二项分布、泊松分布以及伽玛分布进行一些实验和讨论。 二项分布是最重要的离散概率分布之一。假设有一种只有两个结果的试验，其成功概率为 P,那么二项分布描述了进行n次这样的独立试验而成功k次的概率。二项分布的概率质量函数公式如下：

$$f(k;n,p)=\frac{n!}{k!(n-k)!}p^k(1-p)^{n-k}$$

统计—stats

可以通过二项分布的概率质量公式计算投掷5次骰子出现3次6点的概率。使用二项分布的概率质量函数pmf()可以很容易计算出现k次6点的概率。和概率密度函数pdf()类似，pmf()的第一个参数为随机变量的取值，后面的参数为描述随机分布所需的参数。对于二项分布来说，参数分别为n和P,而取值范围则为0到n之间的整数。下面的程序计算k为0到5所对应的概率：

```
>>> stats.binom.pmf(range(6), 5, 1/6.0)
array([0.401878, 0.401878, 0.166751, 0.032150,
0.003215, 0.000129])
```

由结果可知：出现0或1次6点的概率为40.2%,而出现3次6点的概率为3.215%。

统计—stats

在二项分布中，如果试验次数n很大，而每次试验成功的概率p很小，其乘积np比较适中，那么试验成功次数的概率可以用泊松分布近似描述。

在泊松分布中，使用 λ 描述单位时间(或单位面积)内随机事件的平均发生率。如果将二项分布中的试验次数n看作单位时间内所做的试验次数，那么它和事件出现概率P的乘积就是事件的平均发生率，即 $\lambda = np$ 。泊松分布的概率质量函数公式如下：

$$f(k;\lambda) = \frac{e^{-\lambda}\lambda^{k}}{k!}$$

统计—stats

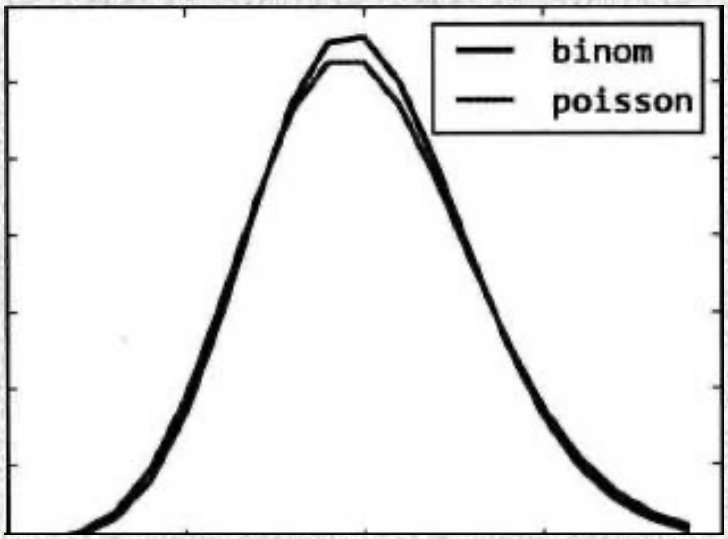
下面的程序分别计算二项分布和泊松分布的概率质量函数，当n足够大时，二者是十分接近的。(scipy_binom_poisson.py)

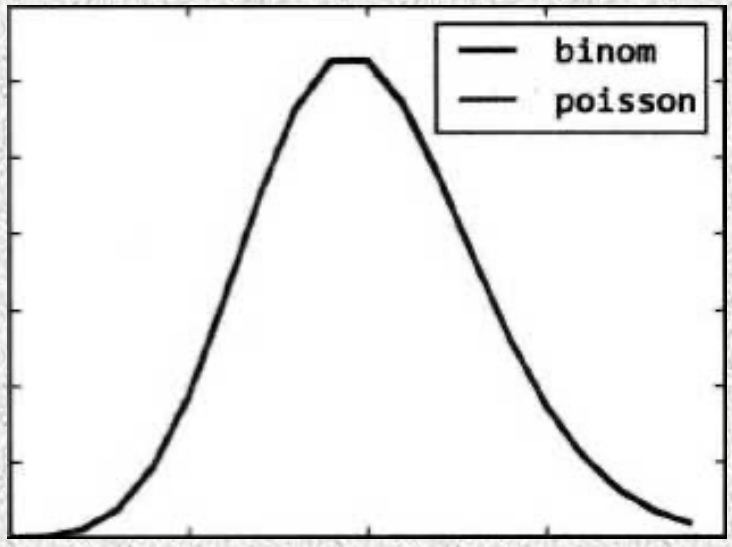
程序中事件平均发生率 λ 恒等于10。根据二项分布的试验次数计算每次事件出现的概率 $p=\lambda/n$ 。程序中的运算部分大致如下：

```
>>> _lambda = 10.0
>>> k = np.arange(20)
>>> possion = stats .poisson .pmf(k, _lambda) # 泊松分布
>>> binom100 = stats.binom.pmf(k, 100, _lambda/100) #二项式分布 100
>>> binom1000=stats.binom.pmf(k, 1000 , _lambda/1000) #二项式分布 1000
```

统计—stats

```
>>> np.max(np.abs(binom100-possion)) # 计算最大误差
 0.0067553111033533312
>>> np.max(np.abs(binom1000-possion))# n为 1000时，误差较小
0.00063017540509099912
```



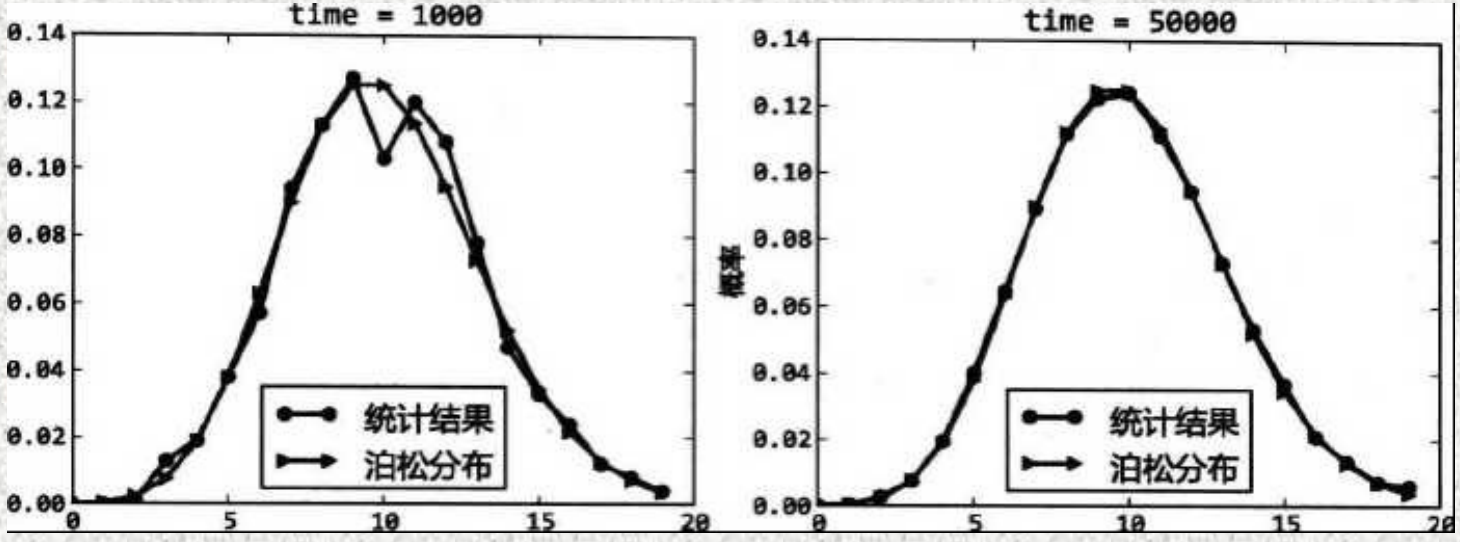


统计—stats

泊松分布适合描述单位时间内随机事件发生次数的分布情况。例如某设施在一定时间内的使用次数、机器出现故障的次数、自然灾害发生的次数等等。

下面使用随机数模拟泊松分布，并与其概率质量函数进行比较，事件每秒的平均发生次数为10,即 $\lambda = 10$ 。其中观察时间分别为1000秒，50000秒。可以看出：观察时间越长，事件每秒发生的次数就越符合泊松分布。(scipy_poisson_sim.py)

统计—stats



下面直接在解释器中介绍泊松分布的模拟过程。首先定义事件发生率 λ 和观察时间：

```
>>> _lambda = 10
>>> time = 1000
```

统计—stats

可以用NumPy的随机数生成函数rand()，产生平均分布于0到time之间的_lambda*time 个事件所发生的时刻。由于rand()产生的是0到1之间的平均分布的随机数，因此需要对其结果扩大time倍：

```
>>> t = np.random.rand(_lambda*time )*time
```

用histogram()可以统计数组t中每秒之内事件发生的次数count:

```
>>> count, time_edges = np.histogram(t, bins=time, range=(0,time))
>>> count
array([10, 9, 8, …, 11, 10, 18])
```

统计—stats

根据泊松分布的定义，count数组中数值的分布情况应该符合泊松分布。下面统计事件次数在0到20区间内的概率分布。当histogram()的normed参数为True并且每个统计区间的长度为1时，其结果和概率质量函数相等。

```
>>> dist, count_edges = np. histogram (count, bins=20, range= (0,20), normed=True)
>>> x = count_edges[:-1]
>>> poisson = stats .poisson.pmf(x, _lambda)
>>> np.max(np.abs(dist-poisson)) #最大误差很小,符合泊松分布
 0.008835624103707570 6
```

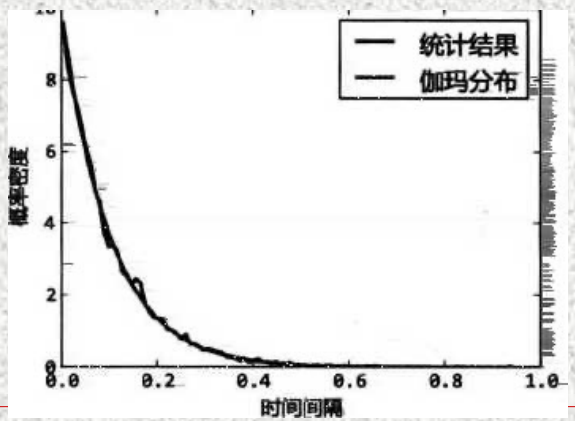
统计—stats

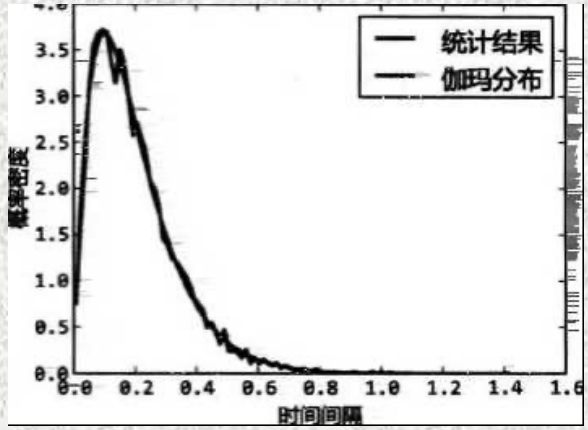
还可以换一个角度看随机事件的分布问题。可以观察相邻两个事件之间时间间隔的分布情况，或者隔k个事件的时间间隔的分布情况。根据概率论，事件之间的时间间隔应符合伽玛分布，由于时间间隔可以是任意数值，因此伽玛分布是一种连续概率分布。伽玛分布的概率密度函数公式如下，它描述第k个事件发生所需的等待时间的概率分布。$\Gamma(k)$ 是伽玛函数，当 k为整数时，它的值和k的阶乘k!相等。

$$f(X;k,\lambda)=\frac{X^{(k-1)}\lambda^{k}e^{(-\lambda X)}}{\Gamma(k)}$$

统计—stats

程序scipy_gamma_sim.py模拟了事件的时间间隔的伽玛分布，观察时间为10000秒，平均每秒产生10个事件。图中“k=1”，它表示相邻两个事件之间的时间间隔的分布，而“k=2”则表示相隔一个事件的两个事件之间的时间间隔的分布，可以看出它们都符合伽玛分布.





统计—stats

下面直接在解释器中模拟伽玛分布。首先在10000秒之内产生100000个随机事件发生的时刻.因此事件的平均发生次数为每秒10次：

```
>>> _lambda = 10
>>> time = 10000
>>> t = np.random.rand(_lambda*time)*time
```

为了计算事性前后的时间间隔，需要先对随机时刻进行排序：

```
>>> t.sort()
```

统计—stats

然后分别计算”k=1”和“k=2”时的时间间隔：

```
>>> s1 = t[1:] - t[:-1] #相邻两个事件之间的时间间隔
>>> s2 = t[2:] - t[:-2] #相隔一个事件的两个事件之间的时间间隔
```

对s1和s2分别调用histogram()进行概率统计，设置normed为True可以直接统计概率密度：

```
>>> dist1, x1= np.histogram(s1, bins=100, normed=True)
>>> dist2, x2 = np.histogram(s2 , bins=100, normed=True)
```

统计—stats

histogram()返回的第二个值为统计区间的边界，采用gamma.pdf()计算伽玛分布的概率密度时，使用各个区间的中值进行计算。Pdf()的第二个参数为k值，scale参数为1/λ（频率）：

```
>>> gamma1 = stats.gamma.pdf((x1[:-1]+x1[1:])/2, 1, scale=1.0/_lambda)
>>> gamma2 = stats.gamma.pdf((x2[:-1]+x2[1:])/2, 2, scale=1.0/_lambda)
>>> np.max(np.abs(gamma1 - dist1))
0.13557317865888141
>>> np.max(np.abs(gamma2 - dist2))
0.08737503086179465 6
```

统计—stats

由于概率密度函数的值本身比较大，因此上面的误差已经很小了：

```
>>> np.max(gamma1), np.max(gamma2)
(9.3483221580498537, 3.676795324101365 6)
```

gamma分布：scipy_gamma_dist.py

统计—stats

- t-检验

ttest_1samp, ttest_ind, ttest_rel均进行双侧检验

H0:μ=μ0

H1:μ≠μ0

单样本T检验-ttest_1samp

```
from scipy import stats
import numpy as np
np.random.seed(7654567) # 保证每次运行都会得到相同结果
rvs = stats.norm.rvs(loc=5, scale=10, size=(50,2))
stats.ttest_1samp(rvs, [1, 2]) #检验两列数的均值与1和2的差异是否显著
(array([ 2.0801775 ,  2.44893711]), array([ 0.04276084,  0.01795186]))
```

统计—stats

#分别显示两列数的t统计量和p值。由p值分别为0.042和0.018，当p值小于0.05时，认为差异显著，即第一列数的均值不等于1，第二列数的均值不等于2。

```
stats.ttest_1samp(rvs,[5.0,0.0])
(array([-0.68014479,  4.11038784]),
 array([  4.99613833e-01,   1.49986458e-04]))
```

#第一列数均值等于5（不拒绝原假设——均值等于5），第二列数均值不等于0（拒绝原假设——均值不等于5）

两独立样本t检验-ttest_ind

```
rvs1 = stats.norm.rvs(loc=5,scale=10,size=500)
rvs3 = stats.norm.rvs(loc=5, scale=20, size=500)
stats.ttest_ind(rvs1, rvs3, equal_var = False)
(0.38824573377478966, 0.69794492597976743)
```

当不确定两总体方差是否相等时，应先利用levene检验，检验两总体是否具有方差齐性，stats.levene(rvs1, rvs3)。如果两总体不具有方差齐性，需要将equal_val参数设定为“False”。

统计—stats

配对样本t检验

```
rvs3 = (stats.norm.rvs(loc=8,scale=10,size=500) +
stats.norm.rvs(scale=0.2,size=500))
stats.ttest_rel(rvs1,rvs3)
Ttest_relResult(statistic=-3.9664861949102703,
pvalue=8.362104305935217e-05)
```

（拒绝原假设，认为rvs1 与 rvs3所代表的总体均值不相等）

```
rvs2 = (stats.norm.rvs(loc=5,scale=10,size=500) +
stats.norm.rvs(scale=0.2,size=500))
stats.ttest_rel(rvs1,rvs2)

Ttest_relResult(statistic=0.9584313812005350 6,
pvalue=0.33830941004896109)
```

（不拒绝原假设，认为rvs1 与 rvs2 所代表的总体均值相等）

稀疏矩阵—sparse

- 直接将dense矩阵转换成稀疏矩阵（以coo_matrix为例：）

```
from scipy import sparse
A =sparse.coo_matrix([[1,2],[3,4]])
print(A)
```

- 按照相应存储形式的要求构建矩阵：

```
row  = array([0,0,0,0,1,3,1])
col  = array([0,0,0,2,1,3,1])
data = array([1,2,1,8,1,1,3])
matrix =sparse.coo_matrix((data, (row,col)), shape=(4,4))
print(matrix)
print(matrix.todense())
```

稀疏矩阵—sparse

- 稀疏矩阵大小及下标存取

```
csr = sparse.csr_matrix([[1, 5], [4, 0], [1, 3]])
print(csr.todense())       #csr.max()
print(csr.shape)           # print csr.transpose()
print(csr.shape[1])

print(csr)
print(csr[0])  # 'coo_matrix' object does not support indexing
print(csr[1,1])
```

- 将稀疏矩阵横向或者纵向合并

```
csr =sparse.csr_matrix([[1, 5, 5], [4, 0, 6], [1, 3, 7]])
csr2 =sparse.csr_matrix([[3, 0, 9]])
print(csr.todense())
print(csr2.todense())
print(sparse.vstack([csr, csr2]).todense())
```

稀疏矩阵—sparse

- sparse矩阵的读取

  可以像常规矩阵一样通过下标读取。也可以通过getrow(i)，getcol(i)读取特定的行或者特定的列，以及nonzero()读取非零元素的位置。

```
print(matrix.todense())
sub1 = matrix.getrow(0)
print(sub1)

sub2 = matrix.getcol(3) # sub2.shape；sub2.todense()
print(sub2)
sub3 = matrix.getcol(1)
print(sub3)          #sub3.nonzero()； sparse.find(sub3)
sub4 = sub2.getrow(3)
print(sub4)                    # sub4.shape ; sub4.ndim
```

稀疏矩阵—sparse

- 稀疏矩阵点积计算

```
A = sparse.csr_matrix([[1, 2, 0], [0, 0, 3]])
print(A.todense())

v = A.T
print(v.todense())

d = A.dot(v)
print(d)
d.shape
```

稀疏矩阵—sparse

- 生成随机数矩阵

```
x=sparse.rand(3,5,0.5)
x.toarray()
#第三个参数表示非零元素的密度

y=sparse.rand(3,5,0.3)
y.toarray()
z=x+y
z
z.toarray()

#稀疏矩阵其它计算
sparse.triu(x).toarray()
x.sqrt().toarray()
x.ceil().toarray()
x.diagonal()
x.shape
```

稀疏矩阵—sparse

- 稀疏矩阵求特征值（scipy_sparse1）

```
import numpy as np
from scipy.sparse.linalg import eigsh
from scipy.linalg import eigh
import scipy.sparse
import time

N = 3000
# 创建随机稀疏矩阵
m = scipy.sparse.rand(N, N)
# 创建包含相同数据的数组
a = m.toarray()
print('The numpy array data size: ' + str(a.nbytes) + '
bytes')
print('The sparse matrix data size: ' + str(m.data.nbytes)
+ ' bytes')
```

稀疏矩阵—sparse

```
# 数组求特征值
t0 = time.time()
res1 = eigh(a)  %res1 = eig(a)
dt = str(np.round(time.time() - t0, 3)) + ' seconds'
print('Non-sparse operation takes ' + dt)
# 稀疏长阵求特征值
t0 = time.time()
res2 = eigsh(m)
dt = str(np.round(time.time() - t0, 3)) + ' seconds'
print('Sparse operation takes ' + dt)
```

```
The numpy array data size: 72000000 bytes
The sparse matrix data size: 720000 bytes
Non-sparse operation takes 13.885 seconds
Sparse operation takes 0.085 seconds
```

稀疏矩阵—sparse

```
# 解方程：

from scipy.sparse import spdiags
from numpy import array
from scipy.sparse.linalg import spsolve


a = spdiags([[1, 2, 3, 4, 5], [6, 5, 8, 9, 10]], [0, 1], 5, 5)
b = array([1, 2, 3, 4, 5])


x = spsolve(a, b)
print(x)
print("Error: ", a*x-b)
```