

Scipy

1. Scipy 的核心计算部分是一些 Fortran 数值计算库：

- 线性代数使用 LAPACK 库
- 快速傅立叶变换使用 FFTPACK 库
- 常微分方程求解使用 ODEPACK 库
- 非线性方程组求解以及最小值求解使用 MINPACK 库

一、常数和特殊函数

1. constants 模块

1. scipy 的 constants 模块包含了众多的物理常数：

- constants.c : 真空中的光速
- constants.h : 普朗克常数
- constants.g : 重力加速度

- o `constants.m_e` : 电子质量

```
In [1]: from scipy import constants as C
In [2]: ?C
In [3]: C.c, C.h, C.g, C.m_e
Out[3]: (299792458.0, 6.62607004e-34, 9.80665, 9.10938356e-31)
```

```
String form. __module__ scipy.constants from e:\software\python3_5\lib\site-packages\scipy\constants.py
File:      e:\software\python3_5\lib\site-packages\scipy\constants\__init__.py
Docstring:
=====
Constants (:mod:`scipy.constants`)
=====

.. currentmodule:: scipy.constants

Physical and mathematical constants and units.

Mathematical constants
=====

=====
``pi``          Pi
``golden``     Golden ratio
``golden_ratio`` Golden ratio
=====

Physical constants
=====

=====
``c``           speed of light in vacuum
``speed of light`` speed of light in vacuum
```

2. 在字典 `constants.physical_constants` 中，以物理常量名为键，对应的值是一个含有三元素的元组，分别为：常量值、单位、误差。

```
In [1]: from scipy import constants as C
In [6]: C.physical_constants["electron mass"], C.physical_constants.keys()
Out[6]: ((9.10938356e-31, 'kg', 1.1e-38),
          dict_keys(['shielded proton magn. moment to nuclear magneton ratio', 'proton magn. moment to nuclear mass constant energy equivalent', 'tau-electron mass ratio', 'proton mass energy equivalent to Bohr magneton ratio', 'atomic mass constant', 'helion mass density', 'Loschmidt constant (273.15 K, 100 kPa)', 'Sackur-Tetrode constant (1 g yromagn. ratio)', 'Bohr magneton in K/T', 'natural unit of energy in MeV', 'Rydberg constant energy equivalent in MeV', 'molar Planck constant', 'electron to alpha particle conversion ratio over 2 pi', 'muon Compton wavelength', 'hertz-kelvin relationship'])
```

3. `constants` 模块还包含了许多单位信息，它们是 1 单位的量转换成标准单位时的数值：

- `C.mile` : 一英里对应的米
- `C.inch` : 一英寸对应的米
- `C.gram` : 一克等于多少千克
- `C.pound` : 一磅对应多少千克

```
In [1]: from scipy import constants as C
In [7]: C.mile, C.inch, C.gram, C.pound
Out[7]: (1609.343999999998, 0.0254, 0.001, 0.4535923699999997)
```

2. special 模块

1. `scipy` 的 `special` 模块是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。这些特殊函数都是 `ufunc` 函数，支持数组的广播运算。

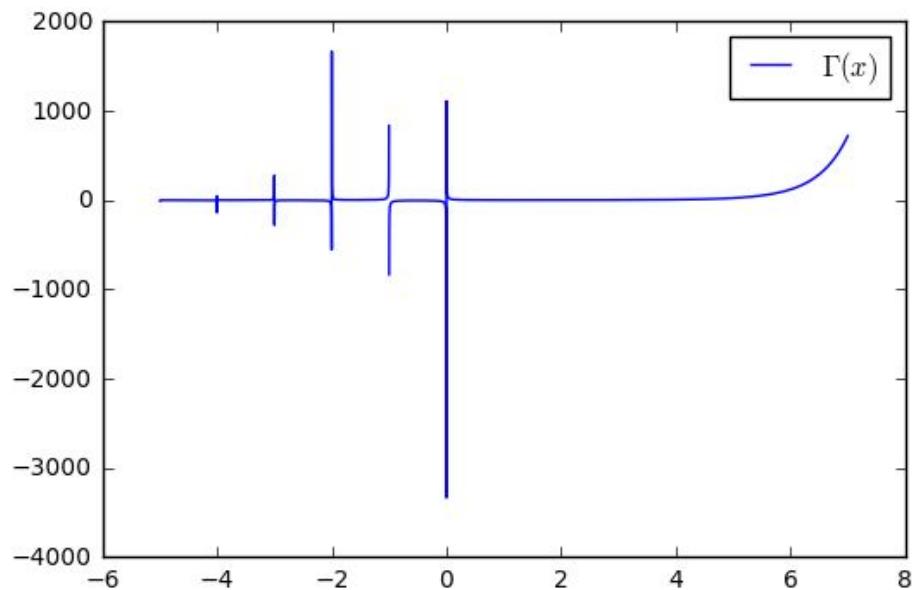
2. `gamma` 函数：`special.gamma(x)`。其数学表达式为：

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

```
In [9]: %matplotlib inline
from scipy import special as S
import numpy as np
import matplotlib.pyplot as plt
```

```
In [19]: x=np.linspace(-5, 7, num=10000)
gamma=S.gamma(x)
fig=plt.figure()
ax=fig.add_subplot(1, 1, 1)
ax.plot(x, gamma, label=r"\Gamma(x)")
ax.legend(loc="best")
fig.show()
```

```
e:\software\python3_5\lib\site-packages\matplotlib\figure.py:397: UserWarning: matplotlib
  show the figure
    "matplotlib is currently using a non-GUI backend, "
```

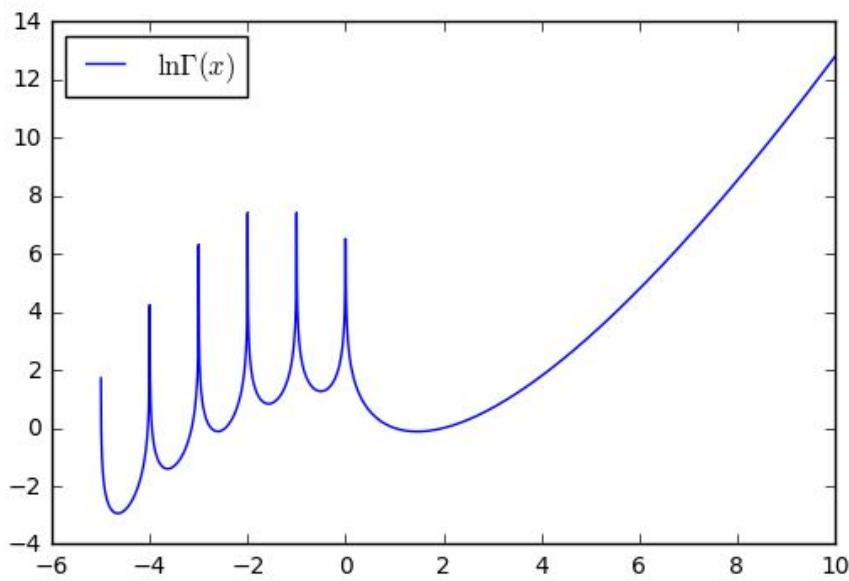


3. `gamma` 函数是阶乘函数在实数和复数系上的扩展，增长的非常迅速。`1000` 的阶乘已经超过了双精度浮点数的表示范围。为了计算更大范围，可以使用 `gammaln` 函数来计算 $\ln(|\Gamma(z)|)$ 的值：`special.gammaln(x)`

```
In [9]: %matplotlib inline
from scipy import special as S
import numpy as np
import matplotlib.pyplot as plt
```

```
In [21]: x=np.linspace(-5, 10, num=10000)
gammaln=S.gammaln(x)
fig=plt.figure()
ax=fig.add_subplot(1, 1, 1)
ax.plot(x, gammaln, label=r"$\ln|\Gamma(x)|$")
ax.legend(loc="best")
fig.show()
```

```
e:\software\python3_5\lib\site-packages\matplotlib\figure.py:397: UserWarning: matplotlib is currently using a non-GUI backend, "
```



4. 计算雅可比椭圆函数：`sn, cn, dn, phi=special.ellipj(u, m)`，其中：

- `sn` = $\sin(\phi)$
- `cn` = $\cos(\phi)$
- `dn` = $\sqrt{1 - m \sin^2(\phi)}$
- `phi` = ϕ
- `u` = $\int_0^\phi \frac{1}{\sqrt{1-m \sin^2(\phi)}} d\theta$

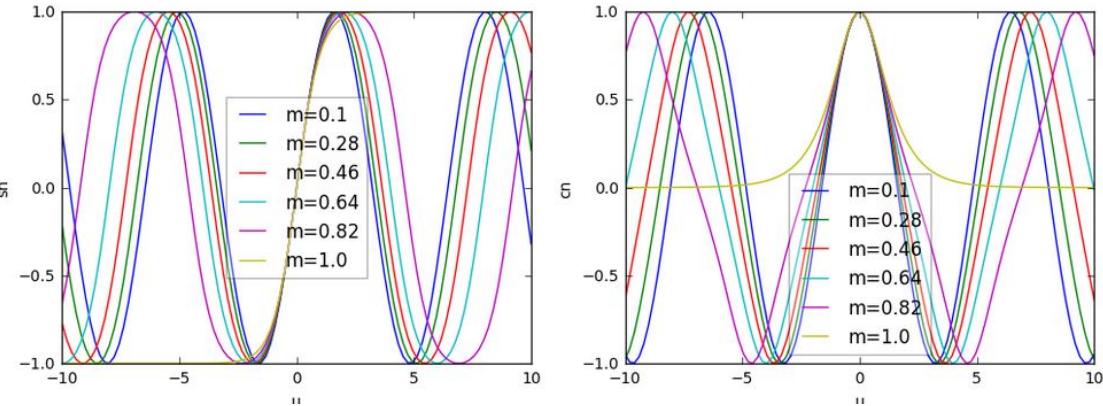
```
In [1]: import numpy as np
import scipy.special as S
import matplotlib.pyplot as plt
%matplotlib inline

In [3]: m=np.linspace(0.1,1,num=6)
u=np.linspace(-10,10,num=100)
sn,cn,dn,phi=S.ellipj(u[:,None],m[None,:])

In [13]: fig=plt.figure(figsize=(12,9))
ylabels=["sn","cn","dn","phi"]
for i,data in enumerate([sn,cn,dn,phi]):
    ax=fig.add_subplot(2,2,i+1)
    for line_index in range(6):
        ax.plot(u,data[:,line_index],label="m=%s"%m[line_index])

    ax.set_xlabel("u")
    ax.set_ylabel(ylabels[i])
    ax.legend(loc="best",framealpha=0.3)
fig.show()

e:\software\python3_5\lib\site-packages\matplotlib\figure.py:397: UserWarning: matplotlib is currently using a non-GUI backend, so cannot show the figure
"matplotlib is currently using a non-GUI backend,"
```



5. `special` 模块的某些函数并不是数学意义上的特殊函数。如 `log1p(x)` 计算的是 $\log(1 + x)$ 。这是因为浮点数精度限制，无法精确地表示非常接近 1 的实数。因此 $\log(1 + 10^{-20})$ 的值为 0。但是 $\log1p(1 + 10^{-20})$ 的值可以计算。

```
In [1]: import numpy as np
import scipy.special as S
import matplotlib.pyplot as plt
%matplotlib inline
```

```
In [15]: print(1+1e-20)
print(np.log(1+1e-20))
print(S.log1p(1e-20))
```

```
1.0
0.0
1e-20
```

二、拟合与优化

- `scipy` 的 `optimize` 模块提供了许多数值优化算法。
- 求解非线性方程组：

```
scipy.optimize.fsolve(func, x0, args=(), fprime=None, full_output=0, col_deriv=0,
xtol=1.49012e-08, maxfev=0, band=None, epsfcn=None, factor=100, diag=None)
```

- `func` : 是个可调用对象，它代表了非线性方程组。给他传入方程组的各个参数，它返回各个方程残差（残差为零则表示找到了根）
- `x0` : 预设的方程的根的初始值
- `args` : 一个元组，用于给 `func` 提供额外的参数。
- `fprime` : 用于计算 `func` 的雅可比矩阵（按行排列）。默认情况下，算法自动推算
- `full_output` : 如果为 `True`，则给出更详细的输出
- `col_deriv` : 如果为 `True`，则计算雅可比矩阵更快（按列求导）
- `xtol` : 指定算法收敛的阈值。当误差小于该阈值时，算法停止迭代
- `maxfev` : 设定算法迭代的最大次数。如果为零，则为 `100*(N+1)`，`N` 为 `x0` 的长度
- `band` : If set to a two-sequence containing the number of sub- and super-diagonals within the band of the Jacobi matrix, the Jacobi matrix is considered banded (only for `fprime=None`)
- `epsfcn` : 采用前向差分算法求解雅可比矩阵时的步长。
- `factor` : 它决定了初始的步长
- `diag` : 它给出了每个变量的缩放因子

返回值：

- `x` : 方程组的根组成的数组
- `infodict` : 给出了可选的输出。它是个字典，其中的键有：
 - `nfev` : `func` 调用次数
 - `njev` : 雅可比函数调用的次数
 - `fvec` : 最终的 `func` 输出
 - `fjac` : the orthogonal matrix, q, produced by the QR factorization of the final approximate Jacobian matrix, stored column wise
 - `r` : upper triangular matrix produced by QR factorization of the same matrix
- `ier` : 一个整数标记。如果为 1，则表示根求解成功
- `mesg` : 一个字符串。如果根未找到，则它给出详细信息

假设待求解的方程组为：

$$\begin{aligned} f_1(x_1, x_2, x_3) &= 0 \\ f_2(x_1, x_2, x_3) &= 0 \\ f_3(x_1, x_2, x_3) &= 0 \end{aligned}$$

那么我们的 `func` 函数为：

```
def func(x):
    x1, x2, x3 = x.tolist() # x 为向量，形状为 (3,)
    return np.array([f1(x1, x2, x3), f2(x1, x2, x3), f3(x1, x2, x3)])
```

数组的 `.tolist()` 方法能获得标准的 `python` 列表

而雅可比矩阵为：

$$J = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \frac{\partial f_1}{\partial x_3} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \frac{\partial f_2}{\partial x_3} \\ \frac{\partial f_3}{\partial x_1} & \frac{\partial f_3}{\partial x_2} & \frac{\partial f_3}{\partial x_3} \end{bmatrix}$$

```
def fprime(x):
    x1,x2,x3=x.tolist() # x 为向量，形状为 (3,)
    return np.array([[df1/dx1,df1/dx2,df1/dx3],
                    [df2/dx1,df2/dx2,df2/dx3],
                    [df3/dx1,df3/dx2,df3/dx3]])
```

```
In [1]: import numpy as np
from scipy.optimize import fsolve
```

求解非线性方程组：

$$\begin{aligned}x_1^2 - 5 &= 0 \\x_0^2 - \sin(x_1 x_2) &= 0 \\x_0 x_1 - 3 &= 0\end{aligned}$$

雅可比矩阵为：

$$\begin{bmatrix} 0 & 2x_1 & 0 \\ 2x_0 & -\cos(x_1 x_2)x_2 & -\cos(x_1 x_2)x_1 \\ x_1 & x_0 & 0 \end{bmatrix}$$

```
In [6]: from math import sin,cos
def func(x):
    x0,x1,x2=x.tolist()
    return [x1**2-5,x0**2-sin(x1*x2),x0*x1-3]
def jac(x):
    x0,x1,x2=x.tolist()
    return [[0, 2*x1, 0], [2*x0, -cos(x1*x2)*x2, -cos(x1*x2)*x1], [x1, x0, 0]]
```

```
In [11]: result=fsolve(func, x0=[1, 1, 1], fprime=jac, full_output=True)
print(result)
print(func(result[0]))
```

```
(array([ 1.16529848,  2.25726855,  0.69705758]), {'r': array([-3.24497185e+00, -8.11716618e-01,  2.14567947e-03,
       4.59142283e+00,   3.77546141e-04, -2.04395965e-03]), 'fvec': array([ 0.09526132,  0.35792407, -0.36960838]), 'nfev': 28, 'fjac': array([[ 2.82634231e-08, -7.18391835e-01, -6.95638679e-01],
       [ 9.83280206e-01, -1.26675070e-01,  1.30818436e-01],
       [ 1.82098974e-01,   6.84007748e-01, -7.06380468e-01]], 'qtf': array([-1.58391905e-05, -2.30771906e-05,  5.23253962e-01]), 'njev': 7}, 5, 'The iteration is not making good progress, as measured by the \n improvement from the last ten iterations.')
[0.09526131850515096, 0.35792406616163586, -0.3696083790605411]
```

3. 最小二乘法拟合数据：

```
scipy.optimize.leastsq(func, x0, args=(), Dfun=None, full_output=0, col_deriv=0,
ftol=1.49012e-08, xtol=1.49012e-08, gtol=0.0, maxfev=0, epsfcn=None,
factor=100, diag=None)
```

- `func`：是个可调用对象，给出每次拟合的残差。最开始的参数是待优化参数；后面的参数由 `args` 给出
- `x0`：初始迭代值
- `args`：一个元组，用于给 `func` 提供额外的参数。
- `Dfun`：用于计算 `func` 的雅可比矩阵（按行排列）。默认情况下，算法自动推算。它给出残差的梯度。最开始的参数是待优化参数；后面的参数由 `args` 给出
- `full_output`：如果非零，则给出更详细的输出
- `col_deriv`：如果非零，则计算雅可比矩阵更快（按列求导）
- `ftol`：指定相对的均方误差的阈值
- `xtol`：指定参数解收敛的阈值

- `gtol` : Orthogonality desired between the function vector and the columns of the Jacobian
- `maxfev` : 设定算法迭代的最大次数。如果为零：如果为提供了 `Dfun`，则为 `100*(N+1)`，`N` 为 `x0` 的长度；如果未提供 `Dfun`，则为 `200*(N+1)`
- `epsfcn` : 采用前向差分算法求解雅可比矩阵时的步长。
- `factor` : 它决定了初始的步长
- `diag` : 它给出了每个变量的缩放因子

返回值：

- `x` : 拟合解组成的数组
- `cov_x` : Uses the `fjac` and `ipvt` optional outputs to construct an estimate of the jacobian around the solution
- `infodict` : 给出了可选的输出。它是个字典，其中的键有：
 - `nfev` : `func` 调用次数
 - `fvec` : 最终的 `func` 输出
 - `fjac` : A permutation of the R matrix of a QR factorization of the final approximate Jacobian matrix, stored column wise.
 - `ipvt` : An integer array of length `N` which defines a permutation matrix, `p`, such that `fjacp = qr`, where `r` is upper triangular with diagonal elements of nonincreasing magnitude
- `ier` : 一个整数标记。如果为 `1/2/3/4`，则表示拟合成功
- `mesg` : 一个字符串。如果解未找到，则它给出详细信息

假设我们拟合的函数是 $f(x, y; a, b, c) = 0$ ，其中 a, b, c 为参数。假设数据点的横坐标为 X ，纵坐标为 Y ，那么我们可以给出 `func` 为：

```
def func(p,x,y):
    a,b,c=p.tolist() # 这里p 为数组，形状为 (3,)； x,y 也是数组，形状都是 (N,)
    return f(x,y;a,b,c)
```

其中 `args=(X,Y)`

而雅可比矩阵为 $\left[\frac{\partial f}{\partial a}, \frac{\partial f}{\partial b}, \frac{\partial f}{\partial c}\right]$ ，给出 `Dfun` 为：

```
def func(p,x,y):
    a,b,c=p.tolist()
    return np.c_[df/da,df/db,df/dc]# 这里p为数组，形状为 (3,)；x,y 也是数组，形状都是 (N,)
```

其中 `args=(X,Y)`

```
In [22]: x=np.linspace(-10, 10, num=100)
y=np.sin(x)+np.random.randint(-1, 1, 100)
```

待拟合的函数为：

$$f(x, y) = ax^3 + bx^2 + cx + d - y = 0$$

雅可比矩阵为：

$$[x^3, x^2, x, 1]$$

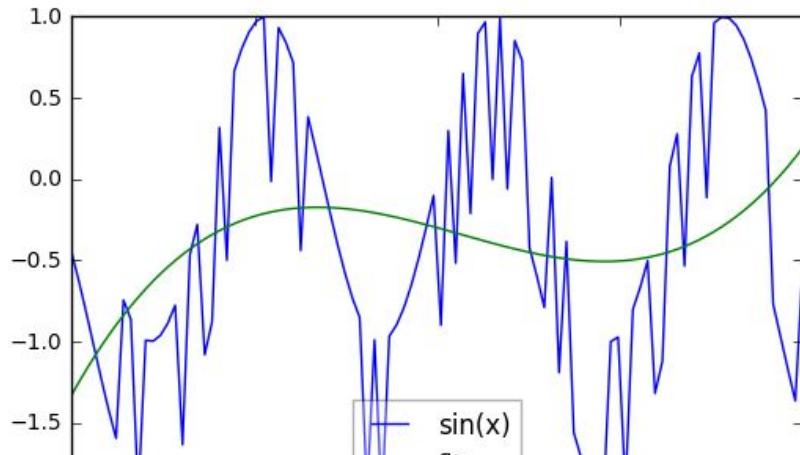
```
In [23]: def func(p, x, y):
    a, b, c, d=p.tolist()
    return a*x**3+b*x**2+c*x+d-y
def jac(p, x, y):
    a, b, c, d=p.tolist()
    return np.c_[x**3, x**2, x, np.ones_like(x)]
```

```
In [24]: result=leastsq(func, x0=np.array([1, 1, 1, 1]), args=(x, y), Dfun=jac)
result
```

```
Out[24]: (array([ 0.00137993, -0.00268245, -0.06202509, -0.29877851]), 3)
```

```
In [25]: fig=plt.figure()
ax=fig.add_subplot(1, 1, 1)
ax.plot(x, y, label="sin(x)")
ax.plot(x, y+func(result[0], x, y), label="fit")
ax.legend(loc="best", framealpha=0.3)
fig.show()
```

```
e:\software\python3_5\lib\site-packages\matplotlib\figure.py:397: UserWarning: matplotlib is currently
show the figure
    "matplotlib is currently using a non-GUI backend, "
```



4. `scipy` 提供了另一个函数来执行最小二乘法的曲线拟合：

```
scipy.optimize.curve_fit(f, xdata, ydata, p0=None, sigma=None, absolute_sigma=False,
check_finite=True, bounds=(-inf, inf), method=None, **kwargs)
```

- `f` : 可调用函数，它的优化参数被直接传入。其第一个参数一定是 `xdata`，后面的参数是待优化参数
- `xdata` : `x` 坐标
- `ydata` : `y` 坐标
- `p0` : 初始迭代值
- `sigma` : `y` 值的不确定性的度量
- `absolute_sigma` : If False, `sigma` denotes relative weights of the data points. The returned covariance matrix `pcov` is based on estimated errors in the data, and is not affected by the overall magnitude of the values in `sigma`. Only the relative magnitudes of the `sigma` values matter. If True,

sigma 描述了输入数据点的一次标准偏差。估计的协方差在 pcov 中基于这些值。

- `check_finite` : 如果为 `True` , 则检测输入中是否有 `nan` 或者 `inf`
- `bounds` : 指定变量的取值范围
- `method` : 指定求解算法。可以为 `'lm'/'trf'/'dogbox'`
- `kwargs` : 传递给 `leastsq/least_squares` 的关键字参数。

返回值 :

- `popt` : 最优化参数
- `pcov` : The estimated covariance of popt.

假设我们拟合的函数是 $y = f(x; a, b, c)$, 其中 a, b, c 为参数。假设数据点的横坐标为 X , 纵坐标为 Y , 那么我们可以给出 `func` 为 :

```
def func(x,a,b,c):  
    return f(x;a,b,c)#x 为数组, 形状为 (N,)
```

```
y=np.sin(x)+np.random.randint(-1, 1, 100)
```

待拟合的函数为：

$$y = ax^3 + bx^2 + cx + d$$

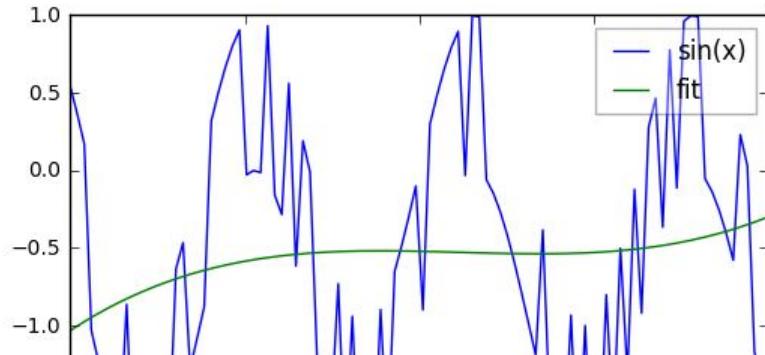
```
In [12]: def func(x, a, b, c, d):
    return a*x**3+b*x**2+c*x+d #x 为数组, 形状为 (N,)
```

```
In [13]: result=curve_fit(func, x, y, p0=np.array([1, 1, 1, 1]))
result
```

```
Out[13]: (array([ 4.13742255e-04, -1.44249786e-03, -4.67269318e-03,
   -5.20945359e-01]), array([[ 3.52240789e-07, -4.57132927e-14, -2.15585308e-05,
   -6.74752226e-18], [-4.57132927e-14, 9.23320935e-06, 2.61286902e-12,
   -3.13991292e-04], [-2.15585308e-05, 2.61286902e-12, 1.57058511e-03,
   7.76186961e-11], [-6.74752226e-18, -3.13991292e-04, 7.76186961e-11,
   1.92175103e-02]]))
```

```
In [15]: fig=plt.figure()
a, b, c, d=result[0].tolist()
ax=fig.add_subplot(1, 1, 1)
ax.plot(x, y, label="sin(x)")
ax.plot(x, func(x, a, b, c, d), label="fit")
ax.legend(loc="best", framealpha=0.3)
fig.show()
```

```
e:\software\python3_5\lib\site-packages\matplotlib\figure.py:397: UserWarning: matplotlib is currently using the figure
"matplotlib is currently using a non-GUI backend, "
```



5. 求函数最小值：

```
scipy.optimize.minimize(fun, x0, args=(), method=None, jac=None, hess=None,
hessp=None, bounds=None, constraints=(), tol=None, callback=None, options=None)
```

- `fun`：可调用对象，待优化的函数。最开始的参数是待优化的自变量；后面的参数由 `args` 给出
- `x0`：自变量的初始迭代值
- `args`：一个元组，提供给 `fun` 的额外的参数
- `method`：一个字符串，指定了最优化算法。可以为：`'Nelder-Mead'`、`'Powell'`、`'CG'`、`'BFGS'`、`'Newton-CG'`、`'L-BFGS-B'`、`'TNC'`、`'COBYLA'`、`'SLSQP'`、`'dogleg'`、`'trust-ncg'`

- `jac` : 一个可调用对象 (最开始的参数是待优化的自变量 ; 后面的参数由 `args` 给出) , 雅可比矩阵。只在 `CG/BFGS/Newton-CG/L-BFGS-B/TNC/SLSQP/dogleg/trust-ncg` 算法中需要。如果 `jac` 是个布尔值且为 `True` , 则会假设 `fun` 会返回梯度 ; 如果是个布尔值且为 `False` , 则雅可比矩阵会被自动推断 (根据数值插值) 。
- `hess/hessp` : 可调用对象 (最开始的参数是待优化的自变量 ; 后面的参数由 `args` 给出) , 海森矩阵。只有 `Newton-CG/dogleg/trust-ncg` 算法中需要。二者只需要给出一个就可以 , 如果给出了 `hess` , 则忽略 `hessp` 。如果二者都未提供 , 则海森矩阵自动推断
- `bounds` : 一个元组的序列 , 给定了每个自变量的取值范围。如果某个方向不限 , 则指定为 `None` 。每个范围都是一个 `(min,max)` 元组。
- `constraints` : 一个字典或者字典的序列 , 给出了约束条件。只在 `COBYLA/SLSQP` 中使用。字典的键为 :
 - `type` : 给出了约束类型。如 `'eq'` 代表相等 ; `'ineq'` 代表不等
 - `fun` : 给出了约束函数
 - `jac` : 给出了约束函数的雅可比矩阵 (只用于 `SLSQP`)
 - `args` : 一个序列 , 给出了传递给 `fun` 和 `jac` 的额外的参数
- `tol` : 指定收敛阈值
- `options` : 一个字典 , 指定额外的条件。键为 :
 - `maxiter` : 一个整数 , 指定最大迭代次数
 - `disp` : 一个布尔值。如果为 `True` , 则打印收敛信息
- `callback` : 一个可调用对象 , 用于在每次迭代之后调用。调用参数为 `x_k` , 其中 `x_k` 为当前的参数向量

返回值 : 返回一个 `OptimizeResult` 对象。其重要属性为 :

- `x` : 最优解向量
- `success` : 一个布尔值 , 表示是否优化成功
- `message` : 描述了迭代终止的原因

假设我们要求解最小值的函数为 : $f(x,y) = (1-x)^2 + 100(y-x^2)^2$, 则雅可比矩阵为 :

$$\left[\frac{\partial f(x,y)}{\partial x}, \frac{\partial f(x,y)}{\partial y} \right]$$

则海森矩阵为 :

$$\begin{bmatrix} \frac{\partial^2 f(x,y)}{\partial x^2} & \frac{\partial^2 f(x,y)}{\partial x \partial y} \\ \frac{\partial^2 f(x,y)}{\partial y \partial x} & \frac{\partial^2 f(x,y)}{\partial y^2} \end{bmatrix}$$

于是有 :

```

def fun(p):
    x,y=p.tolist()#p 为数组，形状为 (2,)
    return f(x,y)
def jac(p):
    x,y=p.tolist()#p 为数组，形状为 (2,)
    return np.array([df/dx,df/dy])
def hess(p):
    x,y=p.tolist()#p 为数组，形状为 (2,)
    return np.array([[ddf/dxx,ddf/dxdy],[ddf/dydx,ddf/dyy]])

```

In [3]: `import numpy as np
from scipy.optimize import minimize`

计算函数

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

的最小值

雅可比矩阵为

$$\begin{bmatrix} 400x(x^2 - y) + 2(x - 1) & 200(y - x^2) \end{bmatrix}$$

海森矩阵为

$$\begin{bmatrix} 400(3x^2 - y) + 2 & -400x \\ -400x & 200 \end{bmatrix}$$

In [6]: `def func(p):
 x, y=p.tolist()
 return (1-x)**2+100*(y-x**2)**2
def jac(p):
 x, y=p.tolist()
 return np.array([2*(x-1)+400*x*(x**2-y), 200*(y-x**2)])
def hess(p):
 x, y=p.tolist()
 return np.array([[400*(3*x**2-y)+2, -400*x], [-400*x, 200]])`

In [7]: `result=minimize(func, x0=np.array([10, 10]), method='Newton-CG', jac=jac, hess=hess)
result`

Out[7]:

```

fun: 1.5507998929041444e-18
jac: array([-4.09654832e-07, -2.05662731e-07])
message: 'Optimization terminated successfully.'
nfev: 83
nhev: 51
nit: 51
njev: 133
status: 0
success: True
x: array([1., 1.])

```

6. 常规的最优化算法很容易陷入局部极值点。`basinhopping` 算法是一个寻找全局最优点的算法。

```

scipy.optimize.basinhopping(func, x0, niter=100, T=1.0, stepsize=0.5,
minimizer_kwarg=None, take_step=None, accept_test=None, callback=None,
interval=50, disp=False, niter_success=None)

```

- `func` : 可调用函数。为待优化的目标函数。最开始的参数是待优化的自变量；后面的参数由 `minimizer_kwarg`s 字典给出
- `x0` : 一个向量，设定迭代的初始值
- `niter` : 一个整数，指定迭代次数
- `T` : 一个浮点数，设定了“温度”参数。
- `stepsize` : 一个浮点数，指定了步长
- `minimizer_kwarg`s : 一个字典，给出了传递给 `scipy.optimize.minimize` 的额外的关键字参数。
- `take_step` : 一个可调用对象，给出了游走策略
- `accept_step` : 一个可调用对象，用于判断是否接受这一步
- `callback` : 一个可调用对象，每当有一个极值点找到时，被调用
- `interval` : 一个整数，指定 `stepsize` 被更新的频率
- `disp` : 一个布尔值，如果为 `True`，则打印状态信息
- `niter_success` : 一个整数。Stop the run if the global minimum candidate remains the same for this number of iterations.

返回值：一个 `OptimizeResult` 对象。其重要属性为：

- `x` : 最优解向量
- `success` : 一个布尔值，表示是否优化成功
- `message` : 描述了迭代终止的原因

假设我们要求解最小值的函数为： $f(x, y) = (1 - x)^2 + 100(y - x^2)^2$ ，于是有：

```
def fun(p):
    x,y=p.tolist()#p 为数组，形状为 (2,)
    return f(x,y)
```

```
In [1]: import numpy as np
from scipy.optimize import basinhopping
```

计算函数

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

的最小值

```
In [2]: def func(p):
    x, y=p.tolist()
    return (1-x)**2+100*(y-x**2)**2
```

```
In [3]: result=basinhopping(func, x0=np.array([10, 10]))
result
```

```
Out[3]:
fun: 3.5403866068371656e-13
lowest_optimization_result: fun: 3.5403866068371656e-13
hess_inv: array([[ 0.49970996,  0.99966947],
   [ 0.99966947,  2.00462507]])
jac: array([-1.14639539e-05,  9.78303649e-06])
message: 'Desired error not necessarily achieved due to precision loss.'
nfev: 168
nit: 19
njev: 39
status: 2
success: False
x: array([ 0.99999957,  0.99999919])
message: ['requested number of basinhopping iterations completed successfully']
minimization_failures: 32
nfev: 13054
nit: 100
njev: 3168
x: array([ 0.99999957,  0.99999919])
```

三、线性代数

1. `numpy` 和 `scipy` 都提供了线性代数函数库 `linalg`。但是 `scipy` 的线性代数库比 `numpy` 更加全面。

2. `numpy` 中的求解线性方程组 : `numpy.linalg.solve(a, b)`。而 `scipy` 中的求解线性方程组 :

```
scipy.linalg.solve(a, b, sym_pos=False, lower=False, overwrite_a=False,
                  overwrite_b=False, debug=False, check_finite=True)
```

- `a` : 方阵 , 形状为 `(M,M)`
- `b` : 一维向量 , 形状为 `(M,)`。它求解的是线性方程组 $\mathbf{Ax} = \mathbf{b}$ 。如果有 k 个线性方程组要求解 , 且 `a` , 相同 , 则 `b` 的形状为 `(M,k)`
- `sym_pos` : 一个布尔值 , 指定 `a` 是否正定的对称矩阵
- `lower` : 一个布尔值。如果 `sym_pos=True` 时 : 如果为 `lower=True` , 则使用 `a` 的下三角矩阵。默认使用 `a` 的上三角矩阵。
- `overwrite_a` : 一个布尔值 , 指定是否将结果写到 `a` 的存储区。
- `overwrite_b` : 一个布尔值 , 指定是否将结果写到 `b` 的存储区。
- `check_finite` : 如果为 `True` , 则检测输入中是否有 `nan` 或者 `inf`

返回线性方程组的解。

通常求解矩阵 $\mathbf{A}^{-1}\mathbf{B}$, 如果使用 `solve(A,B)` , 要比先求逆矩阵、再矩阵相乘来的快。

```
In [1]: import numpy as np
from scipy.linalg import solve
```

求解线性方程组：

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 9 & 1 \\ 27 & 1 & 8 \end{bmatrix} \mathbf{x} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

```
In [5]: a=np.array([[1,2,3],[4,9,1],[27,1,8]])
b=np.array([1,1,1])
x=solve(a,b)
x
```

```
Out[5]: array([-0.05030488,  0.10213415,  0.2820122 ])
```

3. 矩阵的 LU 分解：

```
scipy.linalg.lu_factor(a, overwrite_a=False, check_finite=True)
```

- `a` : 方阵，形状为 `(M,M)`，要求非奇异矩阵
- `overwrite_a` : 一个布尔值，指定是否将结果写到 `a` 的存储区。
- `check_finite` : 如果为 `True`，则检测输入中是否有 `nan` 或者 `inf`

返回:

- `lu` : 一个数组，形状为 `(N,N)`，该矩阵的上三角矩阵就是 `U`，下三角矩阵就是 `L` (`L` 矩阵的对角线元素并未存储，因为它们全部是1)
- `piv` : 一个数组，形状为 `(N,)`。它给出了 `P` 矩阵：矩阵 `a` 的第 `i` 行被交换到了第 `piv[i]` 行

矩阵 `LU` 分解：

$$\mathbf{A} = \mathbf{PLU}$$

其中：`P` 为转置矩阵，该矩阵任意一行只有一个1，其他全零；任意一列只有一个1，其他全零。`L` 为单位下三角矩阵（对角线元素为1），`U` 为上三角矩阵（对角线元素为0）

```
In [8]: import numpy as np
from scipy.linalg import lu_factor
```

矩阵LU分解：

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 9 & 1 \\ 27 & 1 & 8 \end{bmatrix}$$

```
In [9]: a=np.array([[1, 2, 3], [4, 9, 1], [27, 1, 8]])
lu, piv=lu_factor(a)
print(lu)
print(piv)
```

```
[[ 27.          1.          8.          ]
 [  0.14814815   8.85185185 -0.18518519]
 [  0.03703704   0.22175732   2.74476987]]
[2 1 2]
```

```
In [12]: P=np.array([[0, 0, 1], [0, 1, 0], [1, 0, 0]])
L=np.array([[1, 0, 0], [0.14814815, 1, 0], [0.03703704, 0.22175732, 1]])
U=np.array([[27, 1, 8], [0, 8.85185185, -0.18518519], [0, 0, 2.74476987]])
LU=np.dot(L, U)
print(P)
print(L)
print(U)
print(LU)
print(np.dot(P, LU))
```

```
[[0 0 1]
 [0 1 0]
 [1 0 0]]
[[ 1.          0.          0.          ]
 [ 0.14814815  1.          0.          ]
 [ 0.03703704  0.22175732  1.          ]]
[[ 27.          1.          8.          ]
 [  0.          8.85185185 -0.18518519]
 [  0.          0.          2.74476987]]
[[ 27.          1.          8.          ]
 [  4.00000005  9.          1.00000001]
 [  1.00000008  1.99999998  3.00000002]
 [[ 1.00000008  1.99999998  3.00000002]
 [  4.00000005  9.          1.00000001]
 [ 27.          1.          8.          ]]]
```

4. 当对矩阵进行了 LU 分解之后，可以方便的求解线性方程组。

```
scipy.linalg.lu_solve(lu_and_piv, b, trans=0, overwrite_b=False, check_finite=True)
```

- `lu_and_piv` : 一个元组，由 `lu_factor` 返回
- `b` : 一维向量，形状为 `(M,)`。它求解的是线性方程组 $\mathbf{Ax} = \mathbf{b}$ 。如果有 k 个线性方程组要求解，且 `a` 相同，则 `b` 的形状为 `(M, k)`
- `overwrite_b` : 一个布尔值，指定是否将结果写到 `b` 的存储区。

- `check_finite` : 如果为 `True` , 则检测输入中是否有 `nan` 或者 `inf`
- `trans` : 指定求解类型。
 - 如果为 0 , 则求解: $\mathbf{Ax} = \mathbf{b}$
 - 如果为 1 , 则求解: $\mathbf{A}^T \mathbf{x} = \mathbf{b}$
 - 如果为 2 , 则求解: $\mathbf{A}^H \mathbf{x} = \mathbf{b}$

5. `lstsq` 比 `solve` 更一般化 , 它不要求矩阵 \mathbf{A} 是方阵。 它找到一组解 \mathbf{x} , 使得 $\|\mathbf{b} - \mathbf{Ax}\|$ 最小 , 我们称得到的结果为最小二乘解。

```
scipy.linalg.lstsq(a, b, cond=None, overwrite_a=False, overwrite_b=False,
check_finite=True, lapack_driver=None)
```

- `a` : 为矩阵 , 形状为 (M, N)
- `b` : 一维向量 , 形状为 $(M,)$ 。 它求解的是线性方程组 $\mathbf{Ax} = \mathbf{b}$ 。如果有 k 个线性方程组要求解 , 且 `a` , 相同 , 则 `b` 的形状为 (M, k)
- `cond` : 一个浮点数 , 去掉最小的一些特征值。当特征值小于 `cond * largest_singular_value` 时 , 该特征值认为是零
- `overwrite_a` : 一个布尔值 , 指定是否将结果写到 `a` 的存储区。
- `overwrite_b` : 一个布尔值 , 指定是否将结果写到 `b` 的存储区。
- `check_finite` : 如果为 `True` , 则检测输入中是否有 `nan` 或者 `inf`
- `lapack_driver` : 一个字符串 , 指定求解算法。可以为 : `'gelsd'/'gelsy'/'gelss'` 。默认的 `'gelsd'` 效果就很好 , 但是在许多问题上 `'gelsy'` 效果更好。

返回值 :

- `x` : 最小二乘解 , 形状和 `b` 相同
- `residues` : 残差。如果 $rank(\mathbf{A})$ 大于 `N` 或者小于 `M` , 或者使用了 `gelsy` , 则是个空数组 ; 如果 `b` 是一维的 , 则它的形状是 $(1,)$; 如果 `b` 是二维的 , 则形状为 $(K,)$
- `rank` : 返回矩阵 `a` 的秩
- `s` : `a` 的奇异值。如果使用 `gelsy` , 则返回 `None`

```
In [13]: import numpy as np
from scipy.linalg import lstsq
```

求解线性方程组：

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 4 & 9 & 16 & 1 \\ 27 & 64 & 1 & 8 \end{bmatrix} \mathbf{x} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

```
In [14]: a=np.array([[1, 2, 3, 4], [4, 9, 16, 1], [27, 64, 1, 8]])
b=np.array([1, 1, 1])
lstsq(a, b)
```

```
Out[14]: (array([ 0.00205847, -0.01287038,  0.0558478 ,  0.21403472]),
           array([], dtype=float64),
           3,
           array([ 70.75236556,  15.95524212,  3.67872487]))
```

```
In [15]: np.dot(a, np.array([0.00205847, -0.01287038, 0.0558478 , 0.21403472]))
```

```
Out[15]: array([ 0.99999999,  0.99999998,  0.99999993])
```

6. 求解特征值和特征向量：

```
scipy.linalg.eig(a, b=None, left=False, right=True, overwrite_a=False,
                  overwrite_b=False, check_finite=True)
```

- `a`：一个方阵，形状为 `(M,M)`。待求解特征值和特征向量的矩阵。
- `b`：默认为 `None`，表示求解标准的特征值问题： $\mathbf{Ax} = \lambda\mathbf{x}$ 。也可以是一个形状与 `a` 相同的方阵，此时表示广义特征值问题： $\mathbf{Ax} = \lambda\mathbf{Bx}$
- `left`：一个布尔值。如果为 `True`，则计算左特征向量
- `right`：一个布尔值。如果为 `True`，则计算右特征向量
- `overwrite_a`：一个布尔值，指定是否将结果写到 `a` 的存储区。
- `overwrite_b`：一个布尔值，指定是否将结果写到 `b` 的存储区。
- `check_finite`：如果为 `True`，则检测输入中是否有 `nan` 或者 `inf`

返回值：

- `w`：一个一维数组，代表了 `M` 特特征值。
- `v1`：一个数组，形状为 `(M,M)`，表示正则化的左特征向量（每个特征向量占据一列，而不是一行）。仅当 `left=True` 时返回
- `vr`：一个数组，形状为 `(M,M)`，表示正则化的右特征向量（每个特征向量占据一列，而不是一行）。仅当 `right=True` 时返回

`numpy` 提供了 `numpy.linalg.eig(a)` 来计算特征值和特征向量

右特征值： $\mathbf{Ax}_r = \lambda\mathbf{x}_r$ ；左特征值： $\mathbf{A}^H\mathbf{x}_l = \text{conj}(\lambda)\mathbf{x}_l$ ，其中 $\text{conj}(\lambda)$ 为特征值的共轭。

令 $\mathbf{P} = [\mathbf{x}_{r1}, \mathbf{x}_{r2}, \dots, \mathbf{x}_{rM}]$ ，令

$$\Sigma = \begin{bmatrix} \lambda_1 & 0 & 0 & \cdots & 0 \\ 0 & \lambda_2 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & \lambda_M \end{bmatrix}$$

则有：

$$AP = P\Sigma \Rightarrow A = P\Sigma P^{-1}$$

```
In [16]: import numpy as np
from scipy.linalg import eig
```

求解特征值和特征向量：

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 9 & 1 \\ 27 & 1 & 8 \end{bmatrix}$$

```
In [20]: a=np.array([[1,2,3],[4,9,1],[27,1,8]])
w,vr=eig(a,right=True)
print(w)
print(vr)
```

```
[ -5.32474632+0.j  15.24184107+0.j   8.08290525+0.j]
[[ 0.44354388 -0.23677064  0.03133489]
 [-0.06143431 -0.29978965 -0.79591814]
 [-0.89414465 -0.92415682  0.60459278]]
```

```
In [26]: x1=np.array([ 0.44354388, -0.06143431 , -0.89414465]) #每列就是一个特征向量
x2=np.array([-0.23677064, -0.29978965, -0.92415682])
x3=np.array([ 0.03133489, -0.79591814,  0.60459278])
np.dot(a,x1)-w[0]*x1,np.dot(a,x2)-w[1]*x2,np.dot(a,x3)-w[2]*x3
```

```
Out[26]: (array([-4.52339002e-08+0.j, -3.63681493e-08+0.j, -1.88621807e-07+0.j]),
 array([ 6.53268666e-08+0.j, -2.97566833e-08+0.j, -1.14223861e-07+0.j]),
 array([ 3.02467473e-09+0.j, -5.43545120e-09+0.j, -2.72363758e-08+0.j]))
```

```
In [28]: eig(a, left=True)
```

```
Out[28]: (array([-5.32474632+0.j, 15.24184107+0.j, 8.08290525+0.j]),
 array([[ 0.9703333 , -0.85240141,  0.31398256],
 [-0.12085877, -0.33715609, -0.923773 ],
 [-0.20939544, -0.39967189,  0.21922225]]),
 array([[ 0.44354388, -0.23677064,  0.03133489],
 [-0.06143431, -0.29978965, -0.79591814],
 [-0.89414465, -0.92415682,  0.60459278]]))
```

```
In [30]: l_x1=np.array([ 0.9703333 , -0.12085877 , -0.20939544]) #每列就是一个特征向量
l_x2=np.array([-0.85240141, -0.33715609, -0.39967189])
l_x3=np.array([ 0.31398256, -0.923773 ,  0.21922225])
np.dot(a.T,l_x1)-w[0]*l_x1,np.dot(a.T,l_x2)-w[1]*l_x2,np.dot(a.T,l_x3)-w[2]*l_x3
```

```
Out[30]: (array([ 1.26747706e-08+0.j, -6.13360857e-08+0.j,  1.05016118e-08+0.j]),
 array([ 2.05208099e-08+0.j,  2.01388746e-08+0.j, -1.17903705e-08+0.j]),
 array([ 2.65042899e-08+0.j,  4.04809075e-09+0.j,  3.95545308e-09+0.j]))
```

7. 矩阵的奇异值分解：设矩阵 \mathbf{A} 为 $M \times N$ 阶的矩阵，则存在一个分解，使得： $\mathbf{A} = \mathbf{U}\Sigma\mathbf{V}^H$ ，其中 \mathbf{U} 为 $M \times M$ 阶酉矩阵； Σ 为半正定的 $M \times N$ 阶的对焦矩阵；而 \mathbf{V} 为 $N \times N$ 阶酉矩阵。
 Σ 对角线上的元素为 \mathbf{A} 的奇异值，通常按照从大到小排列。

```
scipy.linalg.svd(a, full_matrices=True, compute_uv=True, overwrite_a=False,
check_finite=True, lapack_driver='gesdd')
```

- `a`：一个矩阵，形状为 (M,N) ，待分解的矩阵。
- `full_matrices`：如果为 `True`，则 \mathbf{U} 的形状为 (M,M) 、 \mathbf{V}^H 的形状为 (N,N) ；否则 \mathbf{U} 的形状为 (M,K) 、 \mathbf{V}^H 的形状为 (K,N) ，其中 $K=\min(M,N)$
- `compute_uv`：如果 `True`，则结果中额外返回 \mathbf{U} 以及 \mathbf{V}^H ；否则只返回奇异值。
- `overwrite_a`：一个布尔值，指定是否将结果写到 `a` 的存储区。
- `overwrite_b`：一个布尔值，指定是否将结果写到 `b` 的存储区。
- `check_finite`：如果为 `True`，则检测输入中是否有 `nan` 或者 `inf`
- `lapack_driver`：一个字符串，指定求解算法。可以为：`'gesdd'/'gesvd'`。默认的 `'gesdd'`。

返回值：

- `U`： \mathbf{U} 矩阵
- `s`：奇异值，它是一个一维数组，按照降序排列。长度为 $K=\min(M,N)$
- `Vh`：就是 \mathbf{V}^H 矩阵

判断两个数组是否近似相等 `np.allclose(a1,a2)` (主要是浮点数的精度问题)

```
In [32]: import numpy as np
from scipy.linalg import svd
```

奇异值分解：

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 4 & 9 & 16 & 1 \\ 27 & 64 & 1 & 8 \end{bmatrix}$$

```
In [33]: a=np.array([[1, 2, 3, 4], [4, 9, 16, 1], [27, 64, 1, 8]])
u, s, vh=svd(a)
print(u)
print(s)
print(vh)
```

```
[[ -0.03996069 -0.18773158 -0.98140715]
 [-0.15096438 -0.96978254  0.19165486]
 [-0.98773119  0.15581618  0.0104124 ]]
 [ 70.75236556  15.95524212  3.67872487]
 [[-0.38603035 -0.91379838 -0.049794  -0.11607607]
 [ 0.00878552  0.05444791 -0.99803557 -0.02971935]
 [ 0.01803534  0.11647322  0.03606376 -0.99237499]
 [-0.92226792  0.38528164  0.01204005  0.02889612]]
```

```
In [36]: u2, s2, vh2=svd(a, full_matrices=False)
print(u2)
print(s2)
print(vh2)
```

```
[[ -0.03996069 -0.18773158 -0.98140715]
 [-0.15096438 -0.96978254  0.19165486]
 [-0.98773119  0.15581618  0.0104124 ]]
 [ 70.75236556  15.95524212  3.67872487]
 [[-0.38603035 -0.91379838 -0.049794  -0.11607607]
 [ 0.00878552  0.05444791 -0.99803557 -0.02971935]
 [ 0.01803534  0.11647322  0.03606376 -0.99237499]]
```

```
In [37]: sigma=np.array([[70.75236556, 0, 0], [0, 15.95524212, 0], [0, 0, 3.67872487]])
np.allclose(a, np.dot(np.dot(u2, sigma), vh2))
```

Out[37]: True

四、统计

1. `scipy` 中的 `stats` 模块中包含了很多概率分布的随机变量

- 所有的连续随机变量都是 `rv_continuous` 的派生类的对象
- 所有的离散随机变量都是 `rv_discrete` 的派生类的对象

1. 连续随机变量

1. 查看所有的连续随机变量：

```
[k for k,v in stats.__dict__.items() if isinstance(v,stats.rv_continuous)]
```

```
In [1]: from scipy import stats
[k for k,v in stats.__dict__.items() if isinstance(v, stats.rv_continuous)]
Out[1]: ['foldcauchy',
 'uniform',
 'chi2',
 'lognorm',
 'cauchy',
 'vonmises_line',
 'genlogistic',
 'gausshyper',
 'exponweib',
 'chi',
 'wrapcauchy',
 'norm',
 'rayleigh',
 'gompertz',
 'dweibull',
 'expon',
 'gennorm',
 'gamma'
```

2. 连续随机变量对象都有如下方法：

- `rvs(*args, **kwds)` : 获取该分布的一个或者一组随机值
- `pdf(x, *args, **kwds)` : 概率密度函数在 `x` 处的取值
- `logpdf(x, *args, **kwds)` : 概率密度函数在 `x` 处的对数值
- `cdf(x, *args, **kwds)` : 累积分布函数在 `x` 处的取值
- `logcdf(x, *args, **kwds)` : 累积分布函数在 `x` 处的对数值
- `sf(x, *args, **kwds)` : 生存函数在 `x` 处的取值，它等于 `1-cdf(x)`
- `logsf(x, *args, **kwds)` : 生存函数在 `x` 处的对数值
- `ppf(q, *args, **kwds)` : 累积分布函数的反函数
- `isf(q, *args, **kwds)` : 生存函数的反函数
- `moment(n, *args, **kwds)` : n-th order non-central moment of distribution.
- `stats(*args, **kwds)` : 计算随机变量的期望值和方差等统计量
- `entropy(*args, **kwds)` : 随机变量的微分熵
- `expect([func, args, loc, scale, lb, ub, ...])` : 计算 `func(·)` 的期望值
- `median(*args, **kwds)` : 计算该分布的中值
- `mean(*args, **kwds)` : 计算该分布的均值
- `std(*args, **kwds)` : 计算该分布的标准差
- `var(*args, **kwds)` : 计算该分布的方差
- `interval(alpha, *args, **kwds)` : Confidence interval with equal areas around the median.
- `_call_(*args, **kwds)` : 产生一个参数冻结的随机变量
- `fit(data, *args, **kwds)` : 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
- `fit_loc_scale(data, *args)` : Estimate loc and scale parameters from data using 1st and 2nd moments.
- `nnlf(theta, x)` : 返回负的似然函数

其中的 `args/kwds` 参数可能为（具体函数具体分析）：

- `arg1, arg2, arg3, ...` : array_like. The shape parameter(s) for the distribution

- `loc` : array_like.location parameter (default=0)
- `scale` : array_like.scale parameter (default=1)
- `size` : int or tuple of ints.Defining number of random variates (default is 1).
- `random_state` : None or int or np.random.RandomState instance。 If int or RandomState, use it for drawing the random variates. If None, rely on self.random_state. Default is None.

```
In [2]: %matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats
```

```
In [7]: X=stats.gamma(2.0, scale=4)
X.stats()
```

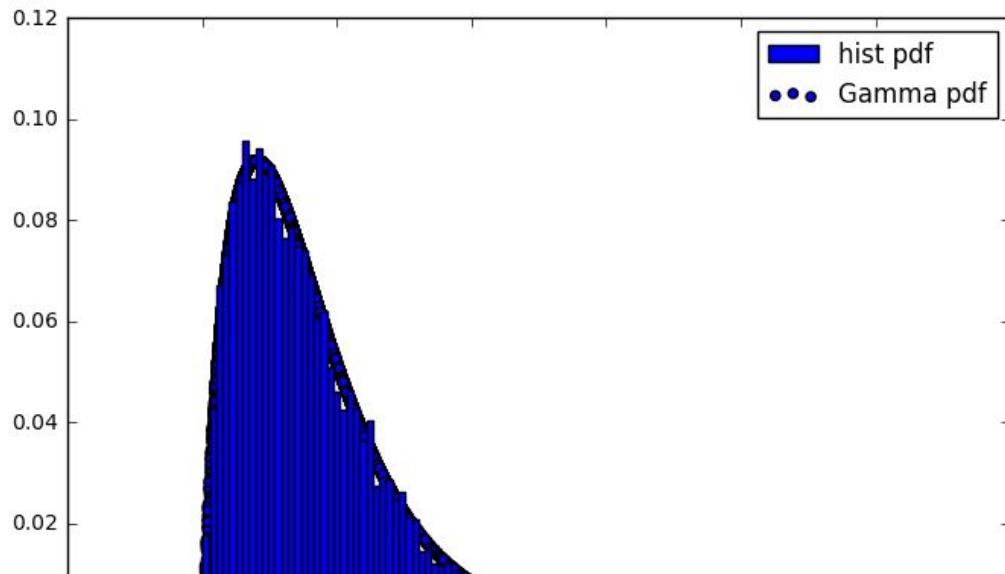
```
Out[7]: (<scipy.stats._distn_infrastructure.rv_frozen at 0x80c3320>,
(array(8.0), array(32.0)))
```

```
In [6]: result=X.rvs(size=10000) # 随机取 10000 个值
np.mean(result), np.var(result)
```

```
Out[6]: (7.8736944454203375, 31.280067608410452)
```

```
In [15]: fig=plt.figure(figsize=(8,6))
ax1=fig.add_subplot(1,1,1)
ax1.scatter(result, X.pdf(result), label="Gamma pdf")
ax1.hist(result, bins=100, normed=1, label="hist pdf")
ax1.legend(loc="best")
fig.show()
```

```
e:\software\python3_5\lib\site-packages\matplotlib\figure.py:397: UserWarning: matplotlib is currently using the figure
"matplotlib is currently using a non-GUI backend, "
```



3. 这些连续随机变量可以像函数一样调用，通过 `loc` 和 `scale` 参数可以指定随机变量的偏移和缩放系数。

- 对于正态分布的随机变量而言，这就是期望值和标准差

2. 离散随机变量

1. 查看所有的连续随机变量：

```
[k for k,v in stats.__dict__.items() if isinstance(v,stats.rv_discrete)]
```

```
In [18]: from scipy import stats
[k for k,v in stats.__dict__.items() if isinstance(v,stats.rv_discrete)]
```

```
Out[18]: ['hypergeom',
 'geom',
 'bernoulli',
 'poisson',
 'boltzmann',
 'skellam',
 'nbinom',
 'binom',
 'planck',
 'logser',
 'randint',
 'zipf',
 'dlaplace']
```

2. 离散随机变量对象都有如下方法：

- `rvs(<shape(s)>, loc=0, size=1)` : 生成随机值
- `pmf(x, <shape(s)>, loc=0)` : 概率密度函数在 `x` 处的值
- `logpmf(x, <shape(s)>, loc=0)` : 概率密度函数在 `x` 处的对数值
- `cdf(x, <shape(s)>, loc=0)` : 累积分布函数在 `x` 处的取值
- `logcdf(x, <shape(s)>, loc=0)` : 累积分布函数在 `x` 处的对数值
- `sf(x, <shape(s)>, loc=0)` : 生存函数在 `x` 处的值
- `logsf(x, <shape(s)>, loc=0, scale=1)` : 生存函数在 `x` 处的对数值
- `ppf(q, <shape(s)>, loc=0)` : 累积分布函数的反函数
- `isf(q, <shape(s)>, loc=0)` : 生存函数的反函数
- `moment(n, <shape(s)>, loc=0)` : non-central n-th moment of the distribution. May not work for array arguments.
- `stats(<shape(s)>, loc=0, moments='mv')` : 计算期望方差等统计量
- `entropy(<shape(s)>, loc=0)` : 计算熵
- `expect(func=None, args=(), loc=0, lb=None, ub=None, conditional=False)` : Expected value of a function with respect to the distribution. Additional kwds arguments passed to integrate.quad
- `median(<shape(s)>, loc=0)` : 计算该分布的中值
- `mean(<shape(s)>, loc=0)` : 计算该分布的均值
- `std(<shape(s)>, loc=0)` : 计算该分布的标准差
- `var(<shape(s)>, loc=0)` : 计算该分布的方差
- `interval(alpha, <shape(s)>, loc=0)` Interval that with alpha percent probability contains a random realization of this distribution.
- `__call__(<shape(s)>, loc=0)` : 产生一个参数冻结的随机变量

3. 我们也可以通过 `rv_discrete` 类自定义离散概率分布：

```
x=range(1,7)
p=(0.1,0.3,0.1,0.3,0.1,0.1)
stats.rv_discrete(values=(x,p))
```

只需要传入一个 `values` 关键字参数，它是一个元组。元组的第一个成员给出了随机变量的取值集合，第二个成员给出了随机变量每一个取值的概率

3. 核密度估计

- 通常我们可以用直方图来观察随机变量的概率密度。但是直方图有个缺点：你选取的直方图区间宽度不同，直方图的形状也发生变化。核密度估计就能很好地解决这一问题。
- 核密度估计的思想是：对于随机变量的每一个采样点 x_i ，我们认为它代表一个以该点为均值、 s 为方差的一个正态分布的密度函数 $f_i(x_i; s)$ 。将所有这些采样点代表的密度函数叠加并归一化，则得到了核密度估计的一个概率密度函数：

$$\frac{1}{N} \sum_{i=1}^N f_i(x_i; s)$$

其中：

- 归一化操作就是 $\frac{1}{N}$ ，因为每个点代表的密度函数的积分都是 1
- s 就是带宽参数，它代表了每个正态分布的形状

如果采用其他的分布而不是正态分布，则得到了其他分布的核密度估计。

- 核密度估计的原理是：如果某个样本点出现了，则它发生的概率就很高，同时跟他接近的样本点发生的概率也比较高。
- 正态核密度估计：

```
class scipy.stats.gaussian_kde(dataset, bw_method=None)
```

参数：

- `dataset`：被估计的数据集。
- `bw_method`：用于设定带宽 s 。可以为：
 - 字符串：如 `'scott'/'silverman'`。默认为 `'scott'`
 - 一个标量值。此时带宽是个常数
 - 一个可调用对象。该可调用对象的参数是 `gaussian_kde`，返回一个标量值

属性：

- `dataset`：被估计的数据集
- `d`：数据集的维度
- `n`：数据点的个数
- `factor`：带宽
- `covariance`：数据集的相关矩阵

方法：

- `evaluate(points)`：估计样本点的概率密度
- `__call__(points)`：估计样本点的概率密度
- `pdf(x)`：估计样本的概率密度

```
In [16]: %matplotlib inline
from scipy import stats
import matplotlib.pyplot as plt
import numpy as np
```

```
In [17]: data=np.random.normal(loc=3, scale=4, size=100)
```

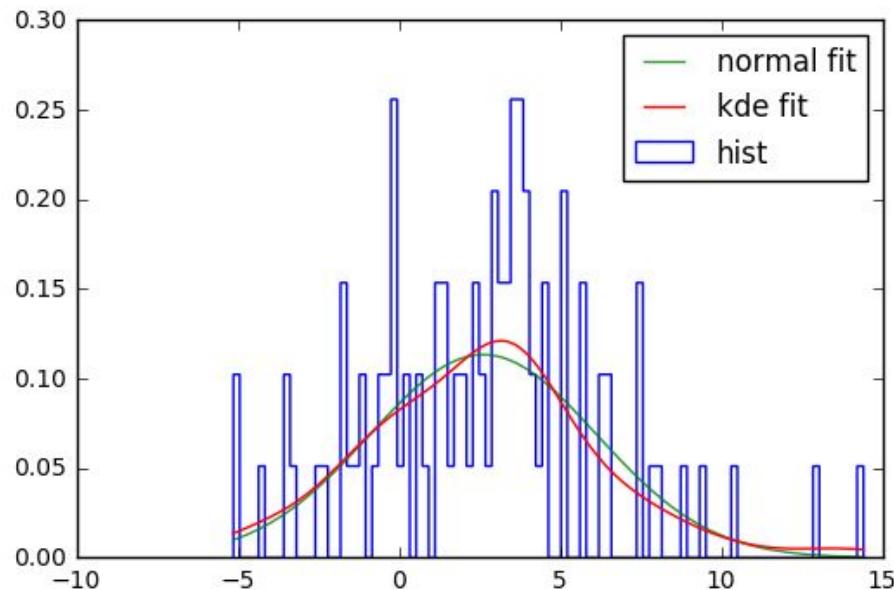
```
In [18]: #正态分布拟合
mean, std=stats.norm.fit(data)
mean, std
```

```
Out[18]: (2.6220224305446096, 3.5181864379690739)
```

```
In [19]: #核密度估计
kernel = stats.gaussian_kde(data)
```

```
In [20]: fig=plt.figure()
ax=fig.add_subplot(1, 1, 1)
_, bins, step=ax.hist(data, bins=100, normed=True, histtype="step", label="hist")
x=np.linspace(bins[0], bins[-1], num=100)
ax.plot(x, stats.norm(mean, std).pdf(x), alpha=0.8, label="normal fit")
ax.plot(x, kernel.pdf(x), label="kde fit")
ax.legend(loc="best")
fig.show()
```

```
e:\software\python3_5\lib\site-packages\matplotlib\figure.py:397: UserWarning: matplotlib is currently using a non-GUI backend, "
```



5. 带宽系数对核密度估计的影响：当带宽系数越大，核密度估计曲线越平滑。

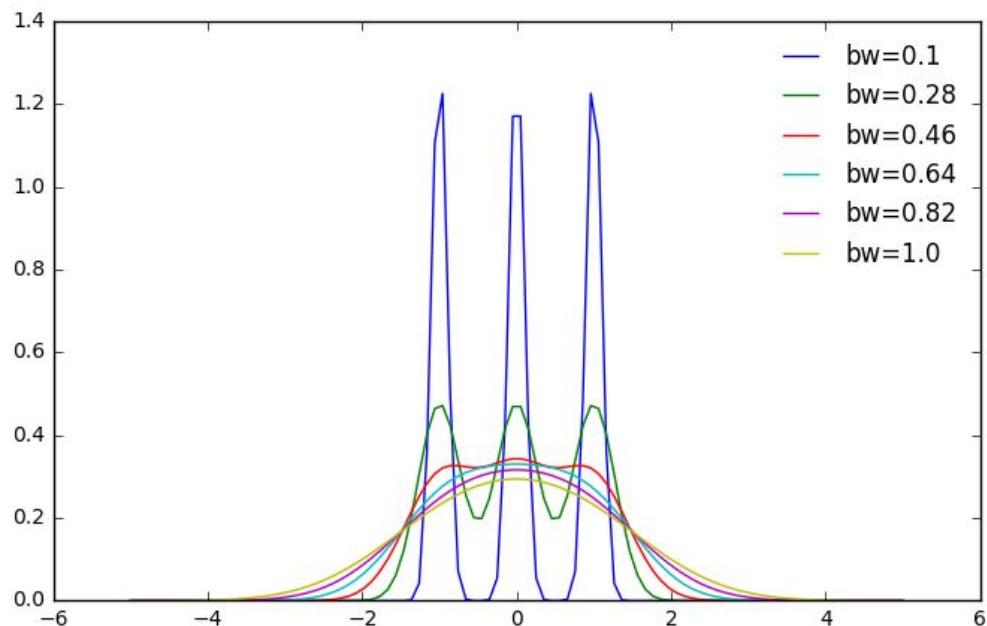
```
In [25]: %matplotlib inline
from scipy import stats
import matplotlib.pyplot as plt
import numpy as np
```

```
In [32]: data=[-1, 0, 1]
x=np.linspace(-5, 5, num=100)

fig=plt.figure(figsize=(8, 5))
ax=fig.add_subplot(1, 1, 1)

for bw in np.linspace(0.1, 1, num=6):
    kde=stats.gaussian_kde(data, bw_method=bw)
    ax.plot(x, kde(x), label="bw=%s"%bw)
ax.legend(loc="best", frameon=False, framealpha=0.3)
fig.show()

e:\software\python3_5\lib\site-packages\matplotlib\figure.py:397: UserWarning: matplotlib is currently using
the figure
    "matplotlib is currently using a non-GUI backend,"
```



4. 常见分布

1. 二项分布：假设试验只有两种结果：成功的概率为 p ，失败的概率为 $1 - p$ 。则二项分布描述了独立重复地进行 n 次试验中，成功 k 次的概率。

- 概率质量函数：

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

- 期望： np
- 方差： $np(1-p)$

`scipy.stats.binom` 使用 `n` 参数指定 n ；`p` 参数指定 p ；`loc` 参数指定平移

```
In [33]: import numpy as np
import scipy.stats as stats

In [37]: p=0.2
n=6
stats.binom.pmf(range(n+1), n, p), stats.binom.mean(n, p), stats.binom.var(n, p)

Out[37]: (array([- 2.62144000e-01,   3.93216000e-01,   2.45760000e-01,
       8.19200000e-02,   1.53600000e-02,   1.53600000e-03,
       6.40000000e-05]), 1.2000000000000002, 0.96000000000000019)
```

2. 泊松分布：泊松分布使用 λ 描述单位时间（或者单位面积）中随机事件发生的平均次数（只知道平均次数，具体次数是个随机变量）。

- 概率质量函数：

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

其物理意义是：单位时间内事件发生 k 次的概率

- 期望： λ
- 方差： λ

在二项分布中，如果 n 很大，而 p 很小。乘积 np 可以视作 λ ，此时二项分布近似于泊松分布。

泊松分布用于描述单位时间内随机事件发生的次数的分布的情况。

`scipy.stats.poisson` 使用 `mu` 参数来给定 λ ，使用 `loc` 参数来平移。

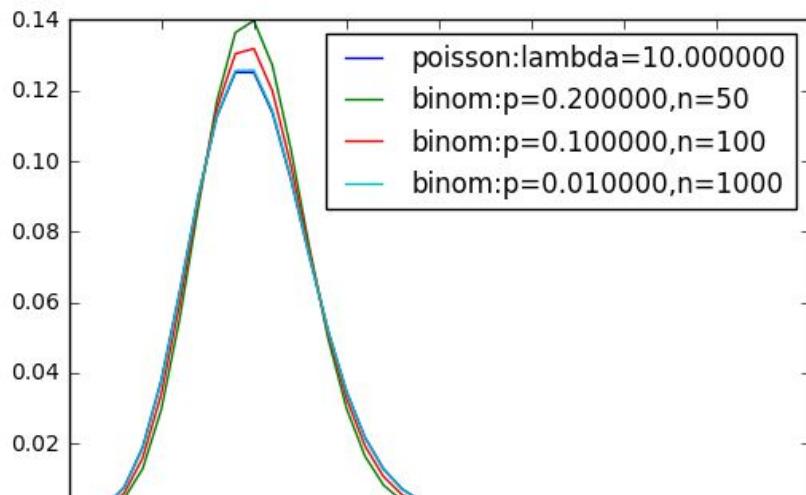
```
In [6]: lmd=10
k=20
stats.poisson.pmf(range(k), lmd), stats.poisson.mean(lmd), stats.poisson.var(lmd)
```

```
Out[6]: (array([ 4.53999298e-05, 4.53999298e-04, 2.26999649e-03,
   7.56665496e-03, 1.89166374e-02, 3.78332748e-02,
   6.30554580e-02, 9.00792257e-02, 1.12599032e-01,
  1.25110036e-01, 1.25110036e-01, 1.13736396e-01,
  9.47803301e-02, 7.29079462e-02, 5.20771044e-02,
  3.47180696e-02, 2.16987935e-02, 1.27639962e-02,
  7.09110899e-03, 3.73216263e-03]), 10.0, 10.0)
```

```
In [7]: %matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
```

```
In [12]: lmd=10
k=40
x=range(k)
fig=plt.figure()
ax=fig.add_subplot(1, 1, 1)
ax.plot(x, stats.poisson.pmf(x, lmd), label="poisson:lambda=%f"%lmd)
for n in [50, 100, 1000]:
    ax.plot(x, stats.binom.pmf(x, n, lmd/n), label="binom:p=%f, n=%d"%(lmd/n, n))
ax.legend(loc="best")
fig.show()
```

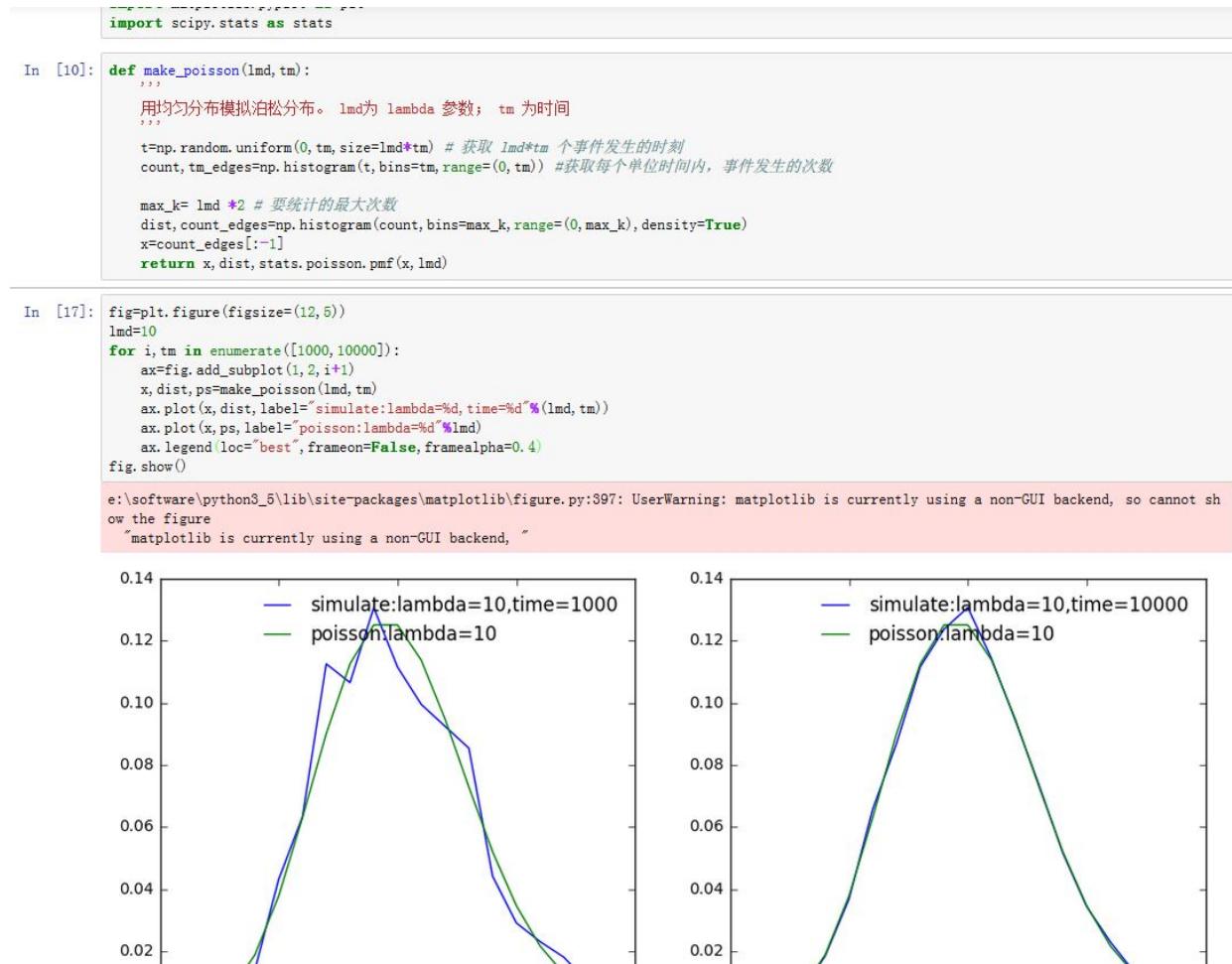
e:\software\python3_5\lib\site-packages\matplotlib\figure.py:397: UserWarning: matplotlib is currently using the figure
"matplotlib is currently using a non-GUI backend, "



3. 用均匀分布模拟泊松分布：

```
def make_poisson(lmd, tm):
    ...
    用均匀分布模拟泊松分布。 lmd为 lambda 参数； tm 为时间
    ...
    t=np.random.uniform(0,tm,size=lmd*tm) # 获取 lmd*tm 个事件发生的时刻
    count,tm_edges=np.histogram(t,bins=tm,range=(0,tm))#获取每个单位时间内，事件发生的次数
    max_k= lmd *2 # 要统计的最大次数
    dist,count_edges=np.histogram(count,bins=max_k,range=(0,max_k),density=True)
    x=count_edges[:-1]
    return x,dist,stats.poisson.pmf(x,lmd)
```

该函数首先随机性给出了 $lmd*tm$ 个事件发生的时间 (时间位于区间 $[0, tm]$) 内。然后统计每个单位时间区间内，事件发生的次数。然后统计这些次数出现的频率。最后将这个频率与理论上的泊松分布的概率质量函数比较。



4. 指数分布：若事件服从泊松分布，则该事件前后两次发生的时间间隔服从指数分布。由于时间间隔是个浮点数，因此指数分布是连续分布。

- 概率密度函数： $f(t; \lambda) = \lambda e^{-\lambda t}$ ， t 为时间间隔
- 期望： $\frac{1}{\lambda}$
- 方差： $\frac{1}{\lambda^2}$

在 `scipy.stats.expon` 中，`scale` 参数为 $\frac{1}{\lambda}$ ；而 `loc` 用于对函数平移

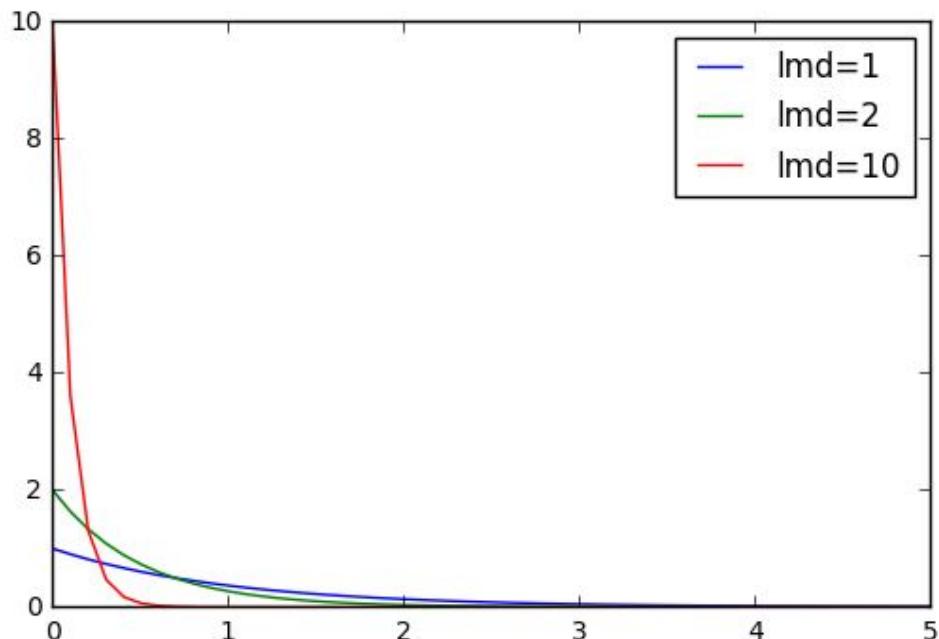
```
In [2]: import numpy as np
import scipy.stats as stats

In [6]: lmd=10
rv=stats.expon(loc=0,scale=1/lmd) #loc 表示偏移; scale 表示 1/lambda
t=np.linspace(0,5,num=10)
rv.pdf(t),rv.mean(),rv.var()

Out[6]: (array([ 1.00000000e+01,   3.86592014e-02,   1.49453385e-04,
      5.77774852e-07,   2.23368144e-09,   8.63504075e-12,
      3.33823780e-14,   1.29053607e-16,   4.98910939e-19,
      1.92874985e-21]), 0.10000000000000001, 0.010000000000000002)

In [7]: %matplotlib inline
import matplotlib.pyplot as plt
fig=plt.figure()
ax=fig.add_subplot(1,1,1)
t=np.linspace(0,5)
for lmd in [1,2,10]:
    ax.plot(t,stats.expon.pdf(t,loc=0,scale=1/lmd),label="lmd=%d"%lmd)
ax.legend(loc="best")
fig.show()

e:\software\python3_5\lib\site-packages\matplotlib\figure.py:397: UserWarning: matpl
ow the figure
    "matplotlib is currently using a non-GUI backend, "
```



5. 用均匀分布模拟指数分布：

```

def make_expon(lmd,tm):
    ...
    用均匀分布模拟指数分布。 lmd为 lambda 参数； tm 为时间
    ...
    t=np.random.uniform(0,tm,size=lmd*tm) # 获取 lmd*tm 个事件发生的时刻
    sorted_t=np.sort(t) #时刻升序排列
    delt_t=sorted_t[1:]-sorted_t[:-1] #间隔序列
    dist,edges=np.histogram(delt_t,bins="auto",density=True)
    x=edges[:-1]
    return x,dist,stats.expon.pdf(x,loc=0,scale=1/lmd) #scale 为 1/lambda

```

```

In [38]: def make_expon(lmd, tm, bins):
    ...
    用均匀分布模拟指数分布。 lmd为 lambda 参数； tm 为时间； bins为分段数量
    ...
    t=np.random.uniform(0,tm,size=lmd*tm) # 获取 lmd*tm 个事件发生的时刻
    sorted_t=np.sort(t) #时刻升序排列
    delt_t=sorted_t[1:]-sorted_t[:-1] #间隔序列

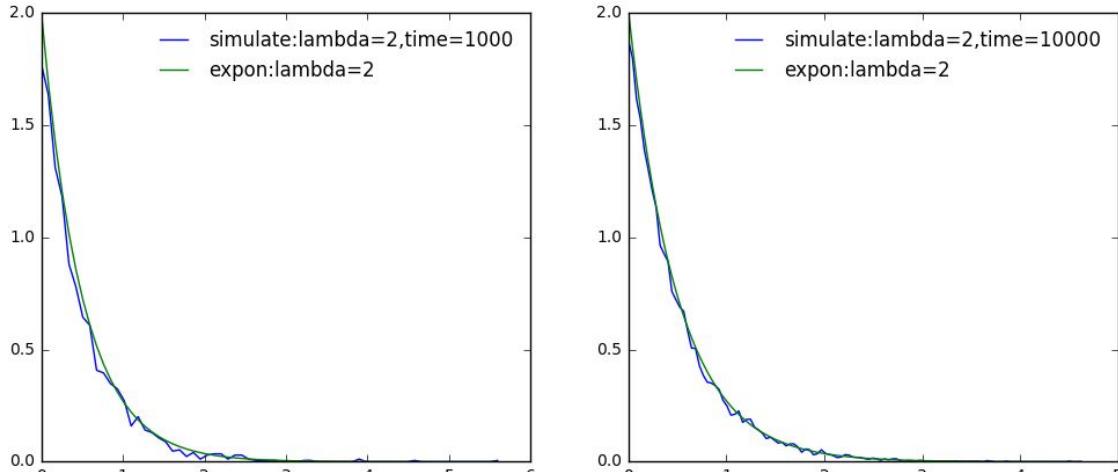
    dist,edges=np.histogram(delt_t,bins="auto",density=True)
    x=edges[:-1]
    return x,dist,stats.expon.pdf(x,loc=0,scale=1/lmd) #scale 为 1/lambda

```

```

In [39]: fig=plt.figure(figsize=(12, 5))
lmd=2
for i,ti in enumerate([1000,10000]):
    ax=fig.add_subplot(1, 2, i+1)
    x, dist,ep=make_expon(lmd,ti, 100)
    ax.plot(x, dist, label="simulate:lambda=%d,time=%d"%(lmd,ti))
    ax.plot(x, ep, label="expon:lambda=%d"%lmd)
    ax.legend(loc="best",frameon=False,framealpha=0.4)
fig.show()
e:\software\python3_5\lib\site-packages\matplotlib\figure.py:897: UserWarning: matplotlib is currently using a non-GUI backend, so cannot show the figure
"matplotlib is currently using a non-GUI backend, "

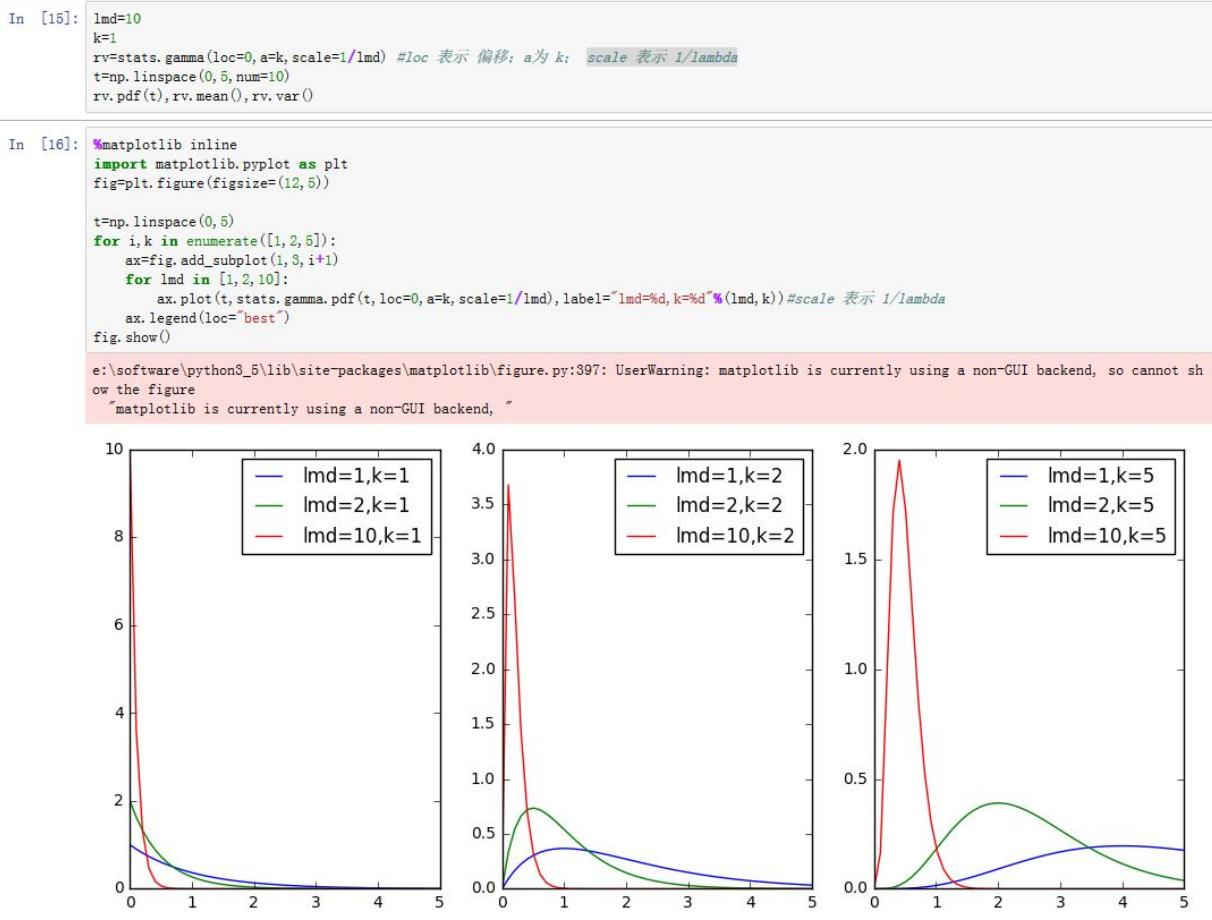
```



6. 伽玛分布：若事件服从泊松分布，则事件第 i 次发生和第 $i + k$ 次发生的时间间隔为伽玛分布。由于时间间隔是个浮点数，因此指数分布是连续分布。

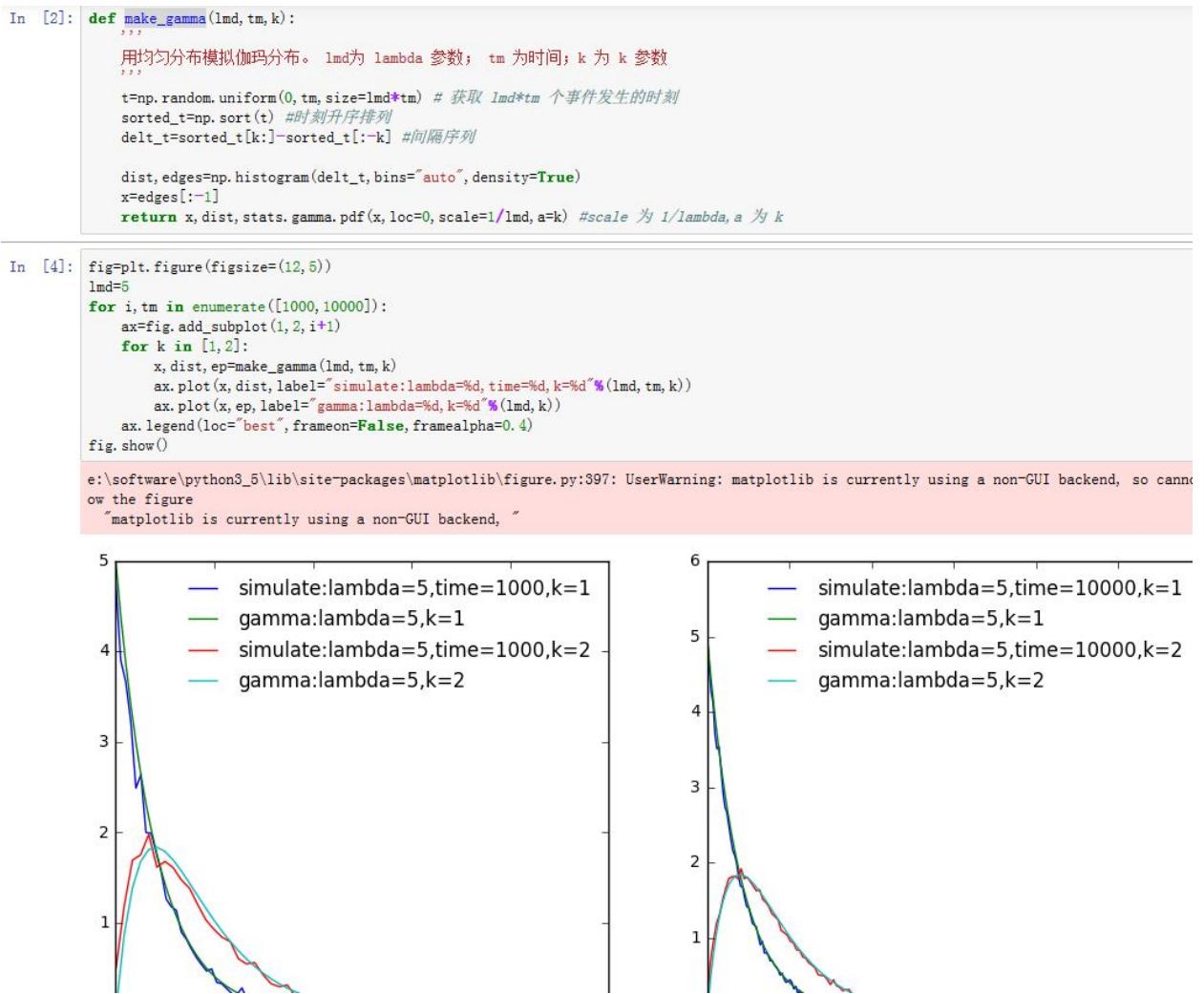
- 概率密度函数： $f(t; \lambda, k) = \frac{t^{(k-1)} \lambda^k e^{-\lambda t}}{\Gamma(k)}$ ， t 为时间间隔
- 期望： $\frac{k}{\lambda}$
- 方差： $\frac{k}{\lambda^2}$

在 `scipy.stats.gamma` 中，`scale` 参数为 $\frac{1}{\lambda}$ ；而 `loc` 用于对函数平移，参数 `a` 指定了 k



7. 用均匀分布模拟伽玛分布：

```
def make_gamma(lmd,tm,k):
    ...
    用均匀分布模拟伽玛分布。 lmd为 lambda 参数； tm 为时间；k 为 k 参数
    ...
    t=np.random.uniform(0,tm,size=lmd*tm) # 获取 lmd*tm 个事件发生的时刻
    sorted_t=np.sort(t) #时刻升序排列
    delt_t=sorted_t[k:]-sorted_t[:-k] #间隔序列
    dist,edges=np.histogram(delt_t,bins="auto",density=True)
    x=edges[:-1]
    return x,dist,stats.gamma.pdf(x,loc=0,scale=1/lmd,a=k) #scale 为 1/lambda,a 为 k
```



五、数值积分

1. `scipy` 的 `integrate` 模块提供了集中数值积分算法，其中包括对常微分方程组 `ODE` 的数值积分。

1. 积分

1. 数值积分函数：

```
scipy.integrate.quad(func, a, b, args=(), full_output=0, epsabs=1.49e-08,
epsrel=1.49e-08, limit=50, points=None, weight=None, wvar=None,
wopts=None, maxp1=50, limlst=50)
```

- `func`：一个 `Python` 函数对象，代表被积分的函数。如果它带有多个参数，则积分只在第一个参数上进行。其他参数，则由 `args` 提供
- `a`：积分下限。用 `-numpy.inf` 代表负无穷
- `b`：积分上限。用 `numpy.inf` 代表正无穷
- `args`：额外传递的参数给 `func`
- `full_output`：如果非零，则通过字典返回更多的信息
- 其他参数控制了积分的细节。参考官方手册

返回值：

- `y` : 一个浮点标量值，表示积分结果
- `abserr` : 一个浮点数，表示绝对误差的估计值
- `infodict` : 一个字典，包含额外的信息

```
In [2]: import numpy as np
import scipy.integrate as integrate
```

计算曲线积分：

$$y = \sqrt{1 - ax^2}$$

```
In [4]: def func(x, a):
    return (1-a*x**2)**0.5
print(integrate.quad(func, -1, 1, args=(1,))) # 半圆, 返回积分和绝对误差
print(integrate.quad(func, -1, 1, args=(0.5,))) # 半椭圆, 返回积分和绝对误差
(1.5707963267948986, 1.0002356720661965e-09)
(1.817827515726139, 7.042942808293354e-11)
```

2. 多重定积分可以通过多次调用 `quad()` 实现。为了调用方便，`integrate` 模块提供了 `dblquad()` 计算二重定积分，提供了 `tplquad()` 计算三重定积分。

3. 二重定积分：

```
scipy.integrate.dblquad(func, a, b, gfun, hfun, args=(),
epsabs=1.49e-08, epsrel=1.49e-08)
```

- `func` : 一个 `Python` 函数对象，代表被积分的函数。它至少有两个参数：`y` 和 `x`。其中 `y` 为第一个参数，`x` 为第二个参数。这两个参数为积分参数。如果有其他参数，则由 `args` 提供
- `a` : `x` 的积分下限。用 `-numpy.inf` 代表负无穷
- `b` : `x` 的积分上限。用 `numpy.inf` 代表正无穷
- `gfun` : `y` 的下边界曲线。它是一个函数或者 `lambda` 表达式，参数为 `x`, 返回一个浮点数。
- `hfun` : `y` 的上界曲线。它是一个函数或者 `lambda` 表达式，参数为 `x`, 返回一个浮点数。
- `args` : 额外传递的参数给 `func`
- `epsabs` : 传递给 `quad`
- `epsrel` : 传递给 `quad`

返回值：

- `y` : 一个浮点标量值，表示积分结果
- `abserr` : 一个浮点数，表示绝对误差的估计值

计算二重积分：

$$\int_{-1}^1 \int_{-\sqrt{1-x^2}}^{\sqrt{1-x^2}} \sqrt{1-x^2 - \frac{y^2}{2}} dy dx$$

```
In [9]: def dblfunc(y, x):
    return (1-x**2-0.5*y**2)**0.5
def gfun(x):
    return -(1-x**2)**0.5
def hfun(x):
    return -gfun(x)
print(integrate.dblquad(dblfunc, -1, 1, gfun, hfun)) # 半球, 返回积分和绝对误差
(2.423770020968185, 2.6909252836750778e-14)
```

4. 三重定积分：

```
scipy.integrate.tplquad(func, a, b, gfun, hfun, qfun, rfun, args=(),
epsabs=1.49e-08, epsrel=1.49e-08)
```

- `func`：一个 Python 函数对象，代表被积分的函数。它至少有三个参数：`z`、`y` 和 `x`。其中 `z` 为第一个参数，`y` 为第二个参数，`x` 为第三个参数。这三个参数为积分参数。如果有其他参数，则由 `args` 提供
- `a`：`x` 的积分下限。用 `-numpy.inf` 代表负无穷
- `b`：`x` 的积分上限。用 `numpy.inf` 代表正无穷
- `gfun`：`y` 的下边界曲线。它是一个函数或者 `lambda` 表达式，参数为 `x`，返回一个浮点数。
- `hfun`：`y` 的上界曲线。它是一个函数或者 `lambda` 表达式，参数为 `x`，返回一个浮点数。
- `qfun`：`z` 的下边界曲面。它是一个函数或者 `lambda` 表达式，第一个参数为 `x`，第二个参数为 `y`，返回一个浮点数。
- `rfun`：`z` 的上边界曲面。它是一个函数或者 `lambda` 表达式，第一个参数为 `x`，第二个参数为 `y`，返回一个浮点数。
- `args`：额外传递的参数给 `func`
- `epsabs`：传递给 `quad`
- `epsrel`：传递给 `quad`

返回值：

- `y`：一个浮点标量值，表示积分结果
- `abserr`：一个浮点数，表示绝对误差的估计值

2. 求解常微分方程组

1. 求解常微分方程组用：

```
scipy.integrate.odeint(func, y0, t, args=(), Dfun=None, col_deriv=0, full_output=0,
ml=None, mu=None, rtol=None, atol=None, tcrit=None, h0=0.0, hmax=0.0,
hmin=0.0, ixpr=0, mxstep=0, mxhnil=0, mxordn=12, mxords=5, printmessg=0)
```

- `func` : 梯度函数。第一个参数为 `y` , 第二个参数为 `t0` , 即计算 `t0` 时刻的梯度。其他的参数由 `args` 提供
- `y0` : 初始的 `y`
- `t` : 一个时间点序列。
- `args` : 额外提供给 `func` 的参数。
- `Dfun` : `func` 的雅可比矩阵，行优先
- `col_deriv` : 一个布尔值。如果 `Dfun` 未给出，则算法自动推导。该参数决定了自动推导的方式
- `full_output` : 如果 `True` , 则通过字典返回更多的信息
- `printmessg` : 布尔值。如果为 `True` , 则打印收敛信息
- 其他参数用于控制求解的细节

返回值 :

- `y` : 一个数组 , 形状为 `(len(t),len(y0))` 。它给出了每个时刻的 `y` 值
- `infodict` : 一个字典 , 包含额外的信息

```
In [10]: import numpy as np
import matplotlib.pyplot as plt
import scipy.integrate as integrate
```

计算洛伦茨吸引子的轨迹

$$\begin{aligned}\frac{dx}{dt} &= \sigma(y - x) \\ \frac{dy}{dt} &= x(\rho - z) - y \\ \frac{dz}{dt} &= xy - \beta z\end{aligned}$$

```
In [11]: def func(w, t0, sigma, rho, beta):
    x, y, z=w.tolist()
    return np.array([sigma*(y-x), x*(rho-z)-y, x*y-beta*z])
def Dfunc(w, t0, sigma, rho, beta):
    x, y, z=w.tolist()
    return np.array([[[-sigma, sigma, 0], [sigma-z, -1, -x], [y, x, -beta]]])
```

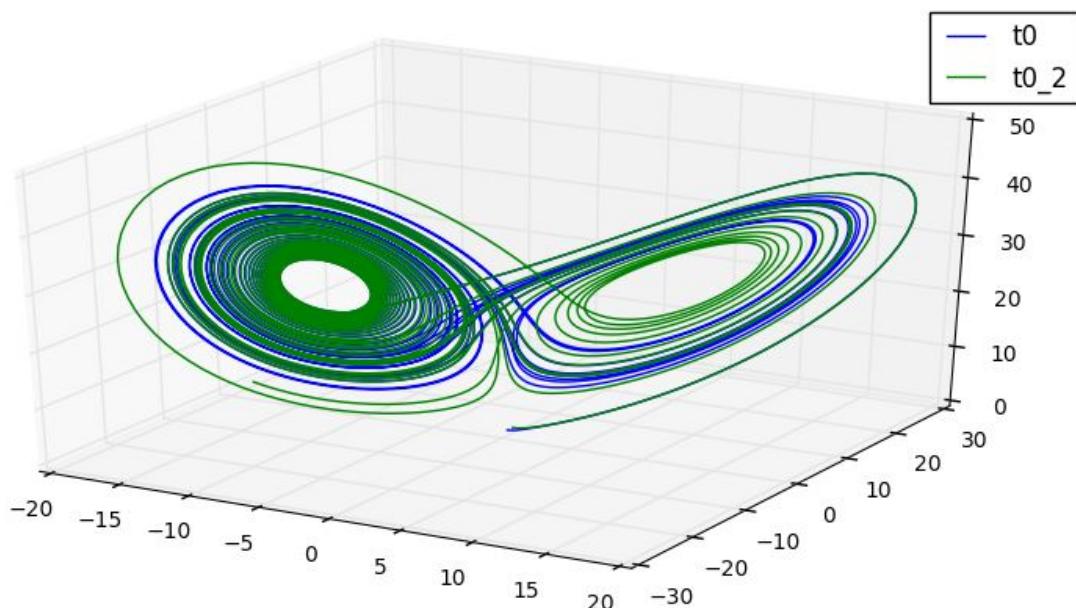
```
In [36]: t=np.linspace(0, 30, num=5000)
t0=[0.0, 1.00, 0.0]
t0_2=[0.0, 2, 0.0]

track1=integrate.odeint(func, t0, t, args=(10.0, 28.0, 3.0), Dfun=Dfunc)
track2=integrate.odeint(func, t0_2, t, args=(10.0, 28.0, 3.0), Dfun=Dfunc)
```

```
In [37]: %matplotlib inline
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm

fig = plt.figure(figsize=(10, 5))
ax = fig.add_subplot(111, projection='3d')
ax.plot(track1[:, 0], track1[:, 1], track1[:, 2], label="t0")
ax.plot(track2[:, 0], track2[:, 1], track2[:, 2], label="t0_2")
ax.legend(loc="best")
fig.show()

e:\software\python8_5\lib\site-packages\matplotlib\figure.py:397: UserWarning: matplotlib is
ow the figure
    "matplotlib is currently using a non-GUI backend, "
```



六、稀疏矩阵

1. 稀疏矩阵是那些矩阵中大部分为零的矩阵。这种矩阵只用保存非零元素的相关信息，从而节约了内存的使用。`scipy.sparse` 提供了多种表示稀疏矩阵的格式。`scipy.sparse.linalg` 提供了对稀疏矩阵进行线性代数运算的函数。`scipy.sparse.csgraph` 提供了对稀疏矩阵表示的图进行搜索的函数。

2. `scipy.sparse` 中有多种表示稀疏矩阵的格式：

- `dok_matrix`：采用字典保存矩阵中的非零元素：字典的键是一个保存元素（行，列）信息的元组，对应的值为矩阵中位于（行，列）中的元素值。这种格式很适合单个元素的添加、删除、存取操作。通常先逐个添加非零元素，然后转换成其他支持高效运算的格式
- `lil_matrix`：采用两个列表保存非零元素。`data` 保存每行中的非零元素，`row` 保存非零元素所在的列。
- `coo_matrix`：采用三个数组 `row/col/data` 保存非零元素。这三个数组的长度相同，分别保存元素的行、列和元素值。`coo_matrix` 不支持元素的存取和增删，一旦创建之后，除了将之转换成其他格式的矩阵，几乎无法对其进行任何操作和矩阵运算。