Numpy 和 Scipy

Table of Contents

0.1 Numpy 基础

0.1.1 创建数组

有许多种方法创建数组，下面是一些简单的例子，使用 `np.array()` 函数，将列表、元组转化为数组：

```
import numpy as np

a = np.array([1, 2, 3, 4])
print(a)
```

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

注意，与列表不同，Numpy 数组只能包含相同类型的数据，下面的例子中，`np.array()` 函数自动将列表中的整数转换为浮点数：

```
b = np.array([3.14, 4, 2, 3])
b
```

```
array([3.14, 4.  , 2.  , 3.  ])
```

列表总是一维的，Numpy 数组可以是多维的，例如下面的例子使用：

```
data = np.array([[1.5, -0.1, 3],
                 [0, -3, 6.5]])
print(data)
```

```
[[ 1.5 -0.1  3. ]
 [ 0.  -3.   6.5]]
```

数组 `data` 是二维数组，可以查看属性 `ndim` 和 `shape`：

```
data.ndim
data.shape
```

```
(2, 3)
```

可以对 `data` 进行通常的数学运算：

```
print(data * 10)
print(data + data)
```

```
[[ 15.  -1.  30.]
 [  0. -30.  65.]]
[[ 3.  -0.2  6. ]
 [ 0.  -6.  13. ]]
```

Numpy 也有函数来生成一些特定格式的数组, 如表 Table 1 所示：

Table 1: Numpy 中生成数组的函数

函数名	描述
`array`	将输入数据（列表、元组、数组或其他序列类型）转换为 ndarray，可以自动推断或显式指定数据类型；默认会复制输入数据
`asarray`	将输入转换为 ndarray，如果输入已经是 ndarray，则不会进行复制
`arange`	类似于内置的 range，但返回的是 ndarray 而不是列表
`ones`, `ones_like`	生成给定形状和数据类型的全 1 数组；ones_like 以另一个数组为模板，生成相同形状和数据类型的全 1 数组
`zeros`, `zeros_like`	类似于 `ones` 和 `ones_like`，但生成的是全 0 数组
`empty`, `empty_like`	通过分配新内存创建新数组，但不会像 ones 和 zeros 那样填充值
`full`, `full_like`	生成给定形状和数据类型的数组，所有值都设置为指定的“填充值”；`full_like` 以另一个数组为模板，生成相同形状和数据类型的填充值数组
`eye`,`identity`	生成单位矩阵（对角线为 1，其余为 0）

```
zeros = np.zeros(10)
print(zeros)
ones = np.ones((2,3), dtype=float)
print(ones)
```

```
# 单位矩阵
idents = np.identity(3)
print(idents)

evens = np.arange(0, 20, 2)
print(evens)
grids = np.linspace(0, 1, 21)
print(grids)
```

```
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[[1. 1. 1.]
 [1. 1. 1.]]
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
[ 0  2  4  6  8 10 12 14 16 18]
[0.   0.05 0.1  0.15 0.2  0.25 0.3  0.35 0.4  0.45 0.5  0.55 0.6  0.65
 0.7  0.75 0.8  0.85 0.9  0.95 1.  ]
```

Numpy 中 random 子库包含丰富的生成随机数的函数，例如：

```
# 生成正态分布
nums_norm = np.random.normal(loc=0, scale=1, size=(4, 3))
print(nums_norm)
nums_int = np.random.randint(low=1, high=11, size=(2, 10))
print(nums_int)
```

```
[[-0.00872898 -0.73263175  2.57475528]
 [-0.25004108  0.0971751  -0.90063461]
 [ 1.50056105  1.87891349  0.81721907]
 [-0.13468373  1.28202193 -1.06205947]]
[[8 1 2 9 3 4 2 4 4 9]
 [1 6 7 9 3 4 5 8 8 1]]
```

0.1.2 数组的索引

注意索引与列表一样，从 0 开始；选择元素时不包括右侧。

```
z = np.array((1,2,3,4,5))
z[0]
z[0:2]
z[-1]
z[::2]
z[::-1]
# 2D arrays
z = np.array([[1,2],
              [3, 4]])
z[0,0]
z[0,:]
z[:,1]
```

```
array([2, 4])
```

0.1.3 数组方法

数组方法众多，例如：

代码段

```
a = np.array((4,3,2,1))
a.sort()

a.sum()
a.mean()
a.max()
a.min()
a.var()
a.std()
a.argmax()
```

```
a.cumsum()
a.cumprod()
```

```
array([ 1,  2,  6, 24])
```

0.1.4 数组的数学运算

注意，运算符 +, - , *, / 和 **，都是逐元素运算。例如：

```
a = np.array([1,2,3,4])
b = np.array([5,6,7,8])
a + b
a * b
a + 10
a * 10
# 2D array
A = np.ones((2,2))
B = np.ones((2,2))
A + B
A+10
A * B
(A+1) ** 2
```

```
array([[4., 4.],
       [4., 4.]])
```

可以使用 @ 或 np.dot() 进行矩阵乘法。如果是向量则计算内积。

```
A = np.array([[1,2],
              [3,4]])
B = np.array([[5,6,],
              [7,8]])
A@B
#or
np.dot(A,B)
```

```
#
b = np.array([0, 1])
A@b
```

```
array([2, 4])
```

0.1.5 例：多项式计算

Numpy 中有一些列简便运算的函数。例如 `np.poly1d()`，多项式求和:

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_N x^N = \sum_{n=0}^{N} a_n x^n$$

```
p = np.poly1d([1,2,3])
print(p)
print(p(2))
```

```
   2
1 x + 2 x + 3
11
```

注意，`np.poly1d()` 函数高阶项在前面。

利用向量计算，自定义一个函数：

```
def poly1d(x, coef):
    X = np.ones_like(coef)
    X[1:] = x
    y = np.cumprod(X) # y = [1,x,x**2,...]
    return coef @ y[::-1]

coef = [1, 2, 3]
poly1d(2, coef=coef)
```

```
np.int64(11)
```

0.1.6 Random 子库

Numpy 中有大量的与随机数生成器有关的函数。

下面是一个例子，注意，没有设定随机种子数，因此每次运行结果会不同。

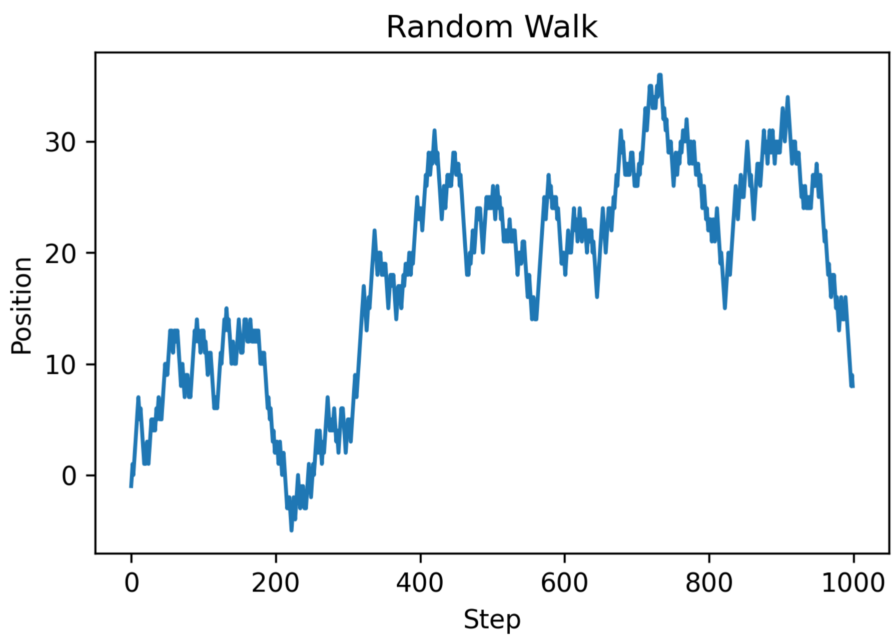
```
import numpy as np

# Define an array of choices
choices = np.array(['apple', 'banana', 'orange', 'grape', 'kiwi'])

# Perform random choice
random_choice = np.random.choice(choices)

# Print the random choice
print(random_choice)
```

```
banana
```

0.1.6.1 例：简单的随机游走

```
import numpy as np
import matplotlib.pyplot as plt

# 设置随机种子以便复现
np.random.seed(0)

# 步数
n_steps = 1000

# 生成每一步的随机步长（-1 或 1）
steps = np.random.choice([-1, 1], size=n_steps)

# 计算随机游走序列
```

```
walk = np.cumsum(steps)

# 绘制线形图
plt.plot(walk)
plt.title('Random Walk')
plt.xlabel('Step')
plt.ylabel('Position')
plt.show()
```



0.1.6.2 例：利用随机数模拟中心极限定理

中心极限定理 (Central Limit Theorem, CLT) 是概率论中一个非常强大的定理。它指出，当从任何形状的总体中抽取足够大的独立同分布 (i.i.d.) 样本时，这些样本均值的分布将近似于正态分布，无论原始总体分布如何。样本量越大，近似程度越好。

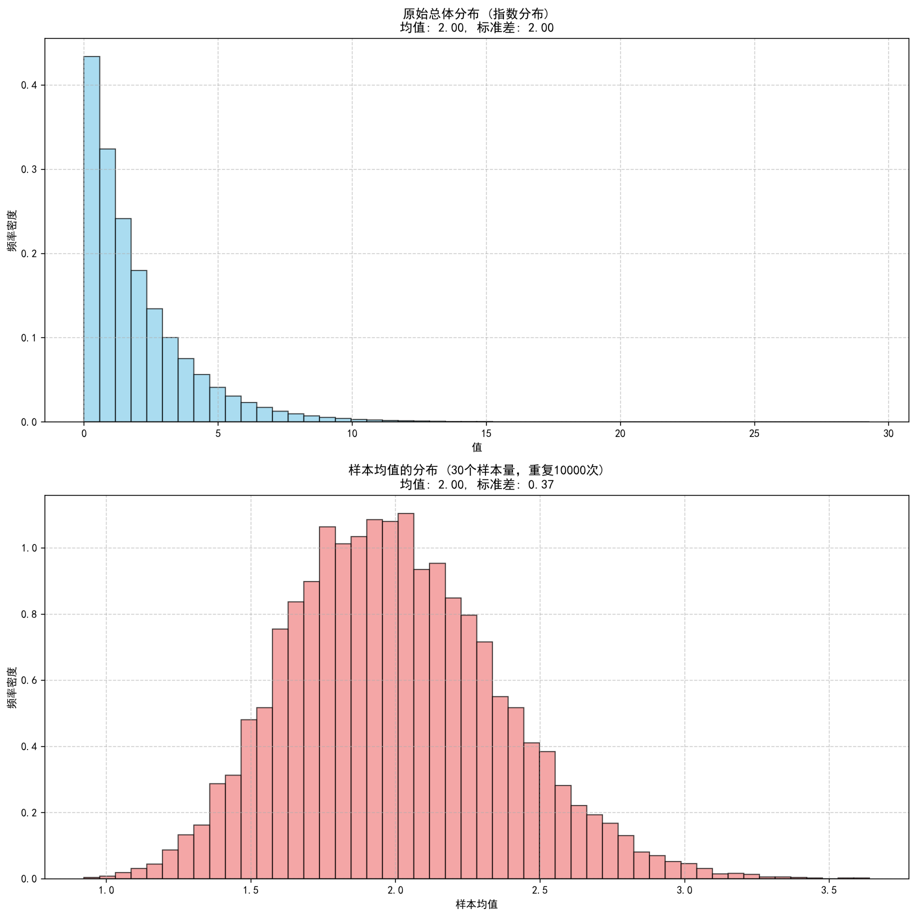
我们将通过以下步骤来模拟验证 CLT：

- 选择一个非正态分布的总体: 比如，一个指数分布或均匀分布，它们的形状都不是钟形的。
- 设置样本参数: 定义每次抽样的样本大小 (sample_size) 和重复抽样的次数 (num_samples)。
- 重复抽样并计算均值: 从总体中抽取 num_samples 次样本，每次抽取 sample_size 个数据点，并计算每次抽样的平均值。
- 可视化: 绘制样本均值的直方图，并与原始总体分布的直方图进行对比。

```
import numpy as np
import matplotlib.pyplot as plt
plt.rcParams['font.family'] = 'SimHei'
plt.rcParams['axes.unicode_minus'] = False

# --- 1. 设置模拟参数 ---
population_size = 1000000  # 原始总体的大小
sample_size = 30           # 每次抽样的样本量 (通常大于 30 就被认为是“大样本”)
num_samples = 10000        # 重复抽样的次数，即我们将有多少个样本均值
np.random.seed(123)

# --- 2. 选择一个非正态分布的总体 (例如: 指数分布) ---
# 指数分布 (Exponential Distribution) 是一种偏态分布，非常适合验证 CLT
# numpy.random.exponential(scale=1.0, size=None)
# scale 参数是均值，这里我们设置均值为 2.0
population_data = np.random.exponential(scale=2.0, size=population_size)
# 也可以用均匀分布作为总体进行验证
# population_data_uniform = np.random.uniform(low=0.0, high=10.0, size=population_size)

# --- 3. 重复抽样并计算均值 ---
sample_means = []
for _ in np.arange(num_samples):
    # 从总体中随机抽取 sample_size 个数据点
    sample = np.random.choice(population_data, size=sample_size, replace=True)
    # 计算样本的均值并添加到列表中
```

```
    sample_means.append(np.mean(sample))

# 将样本均值列表转换为 NumPy 数组，方便后续处理和绘图
sample_means = np.array(sample_means)

# --- 4. 可视化结果 ---
plt.figure(figsize=(12, 12))

# 绘制原始总体分布的直方图
plt.subplot(2, 1, 1) # 1 行 2 列的第一个图
plt.hist(population_data, bins=50, density=True, color='skyblue', edgecolor='black
plt.title(f'原始总体分布 (指数分布)\n均值: {np.mean(population_data):.2f}, 标准差: {
plt.xlabel('值')
plt.ylabel('频率密度')
plt.grid(True, linestyle='--', alpha=0.6)

# 绘制样本均值分布的直方图
plt.subplot(2, 1, 2) # 1 行 2 列的第二个图
plt.hist(sample_means, bins=50, density=True, color='lightcoral', edgecolor='black
plt.title(f'样本均值的分布 ({sample_size}个样本量, 重复{num_samples}次)\n均值: {np.me
plt.xlabel('样本均值')
plt.ylabel('频率密度')
plt.grid(True, linestyle='--', alpha=0.6)

plt.tight_layout() # 调整子图布局，避免重叠
plt.show()

# --- 5. 额外验证: 比较均值和标准差 ---
print("\n--- 模拟结果验证 ---")
print(f" 原始总体的均值 ( ): {np.mean(population_data):.4f}")
print(f" 原始总体的标准差 ( ): {np.std(population_data):.4f}")
print(f" 样本均值的均值 ( _x̄): {np.mean(sample_means):.4f}")
# 根据中心极限定理，样本均值的标准差 (标准误差) 应该约等于 总体标准差 / sqrt(样本量)
```

```
expected_std_of_means = np.std(population_data) / np.sqrt(sample_size)
print(f" 样本均值的标准差 ( _x̄): {np.std(sample_means):.4f}")
print(f" 理论上样本均值的标准差 (  / sqrt(n)): {expected_std_of_means:.4f}")
```




```
--- 模拟结果验证 ---
原始总体的均值 ( ): 1.9988
原始总体的标准差 ( ): 1.9992
样本均值的均值 ( _x̄): 1.9984
样本均值的标准差 ( _x̄): 0.3653
理论上样本均值的标准差 (  / sqrt(n)): 0.3650
```

0.1.7 通用函数

Numpy 中许多函数是通用函数 (universal functions)，是一种在 ndarray 数据中进行逐元素操作的函数，大多数数学函数属于此类。

例如 `np.cos()` 函数：

```
np.cos(1.0)
np.cos(np.linspace(0, 1, 3))
```

```
array([1.        , 0.87758256, 0.54030231])
```

例如，我们想计算 $\frac{0}{1}, \frac{1}{2}, \cdots, \frac{4}{5}$:

```
np.arange(5) / np.arange(1, 6)
```

```
array([0.        , 0.5       , 0.66666667, 0.75      , 0.8       ])
```

Table 2: Numpy 中算术运算子和函数

运算符	对应的 ufunc	描述	示例
+	np.add	加法	1 + 1 = 2
-	np.subtract	减法	3 - 2 = 1
-	np.negative	一元取反	-2
*	np.multiply	乘法	2 * 3 = 6
/	np.divide	除法	3 / 2 = 1.5
//	np.floor_divide	向下取整除法	3 // 2 = 1
**	np.power	幂运算	2 ** 3 = 8
%	np.mod	取模/余数	9 % 4 = 1

0.1.8 例：通用函数

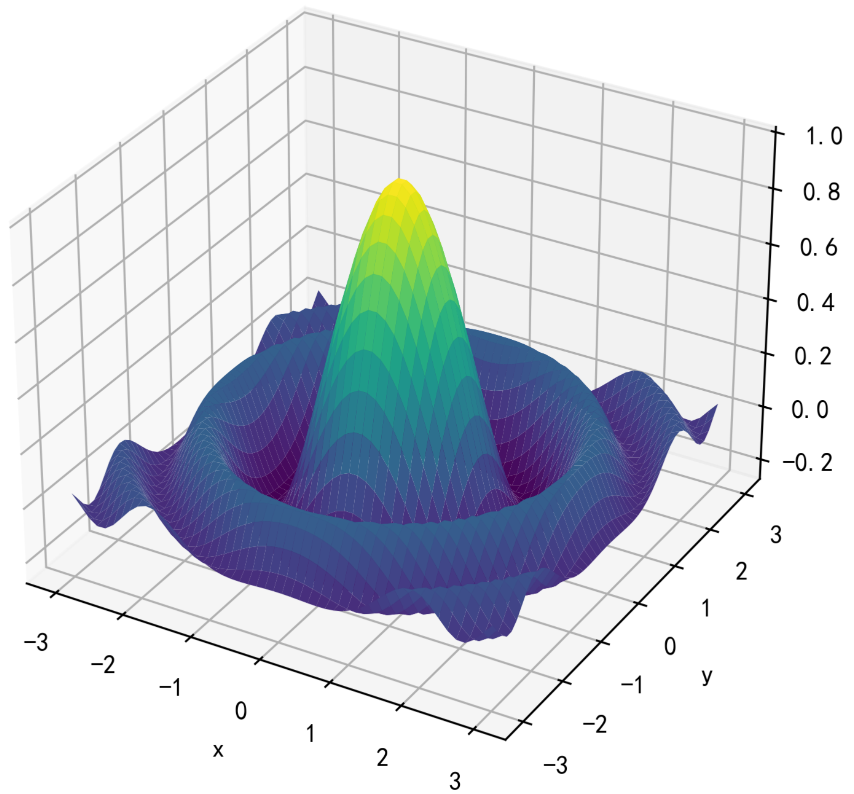
考察最大化函数 $f(x, y)$ 在区间 $ [−a,a] ×[−a, a]$ 上的最大值：

$$f(x, y) = \frac{cos(x^2 + y^2)}{1 + x^2 + y^2}$$

令 $a = 3$。我们定义一个函数，然后生成数组，计算对应的-值，通过栅格（grid）搜索最大值（等于 1）。

```
import numpy as np
from mpl_toolkits.mplot3d import Axes3D

import matplotlib.pyplot as plt

def f(x, y):
    return np.cos(x**2 + y**2) / (1 + x**2 + y**2)

grid = np.linspace(-3, 3, 50)
x, y = np.meshgrid(grid, grid)
z = f(x, y)

# 最大值
max_value = np.max(z)
print(" 函数的最大值:", max_value)

# 绘制 3D 图像
fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(x, y, z, cmap='viridis')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('f(x, y)')
plt.show()
```

```
函数的最大值: 0.9925310162998334
```



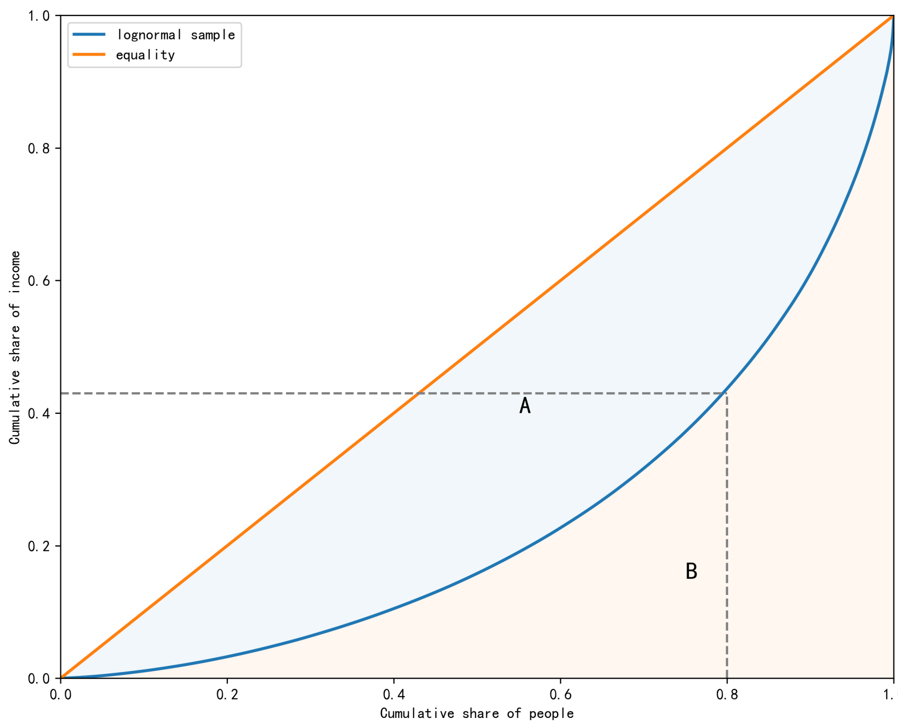
图示

0.1.9 例：洛伦茨曲线和基尼系数

```
import numpy as np # 载入 numpy 库
def lorenz_curve(y):
    n = len(y)
    y = np.sort(y) # 从小到大排序
    s = np.zeros(n + 1) # 生成 n+1 个数值零
    s[1:] = np.cumsum(y) # 从第 2 个数（索引 1）累计求和，使第一个数据点为（0，0）
    cum_people = np.linspace(0, 1, n + 1)
```

```
    cum_income = s / s[n] # s[n] 为最后的值，即所有值的和
    return cum_people, cum_income
```

```
n = 2000
np.random.seed(1)
sample = np.exp(np.random.randn(n))
f_vals, l_vals = lorenz_curve(sample)
#
fig, ax = plt.subplots(figsize=(10, 8))
ax.plot(f_vals, l_vals, label=f'lognormal sample', lw = 2)
ax.plot([0, 1], [0, 1], label='equality', lw = 2)
ax.fill_between(f_vals,l_vals, f_vals, alpha=0.06)
ax.fill_between(f_vals, l_vals, np.zeros_like(f_vals),alpha=0.06)
ax.vlines([0.8], [0], [0.43], linestyles='--', colors='gray')
ax.hlines([0.43], [0], [0.8], linestyles='--', colors='gray')
ax.set_xlim((0,1))
ax.set_ylim((0,1))
ax.text(0.55, 0.4,"A", fontsize=16)
ax.text(0.75,0.15,"B",fontsize=16)
ax.set_xlabel('Cumulative share of people')
ax.set_ylabel('Cumulative share of income')
ax.legend()
plt.show()
```



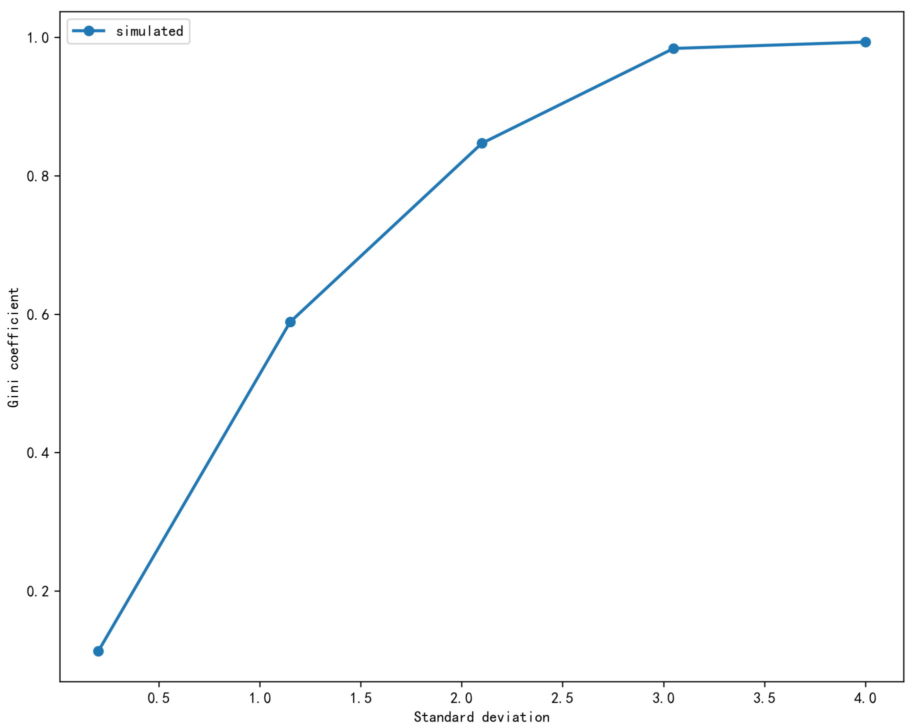
0.1.10 基尼系数

```
def gini(y):
    n = len(y)
    y_1 = np.reshape(y, (n, 1))
    y_2 = np.reshape(y, (1, n))
    g_sum = np.sum(np.abs(y_1 - y_2)) # 利用了 numpy 的广播（broadcasting）
    return g_sum / (2 * n * np.sum(y))
```

```
# 模拟对数正态数据
np.random.seed(1)
k = 5
sigmas = np.linspace(0.2, 4, k)
n = 2000
ginis = []
```

```
for sigma in sigmas:
    mu = -sigma ** 2 / 2
    y = np.exp(mu + sigma * np.random.randn(n))
    ginis.append(gini(y))
```

```
fig, ax = plt.subplots(figsize=(10, 8))
ax.plot(sigmas, ginis, marker = 'o',label='simulated', lw = 2)
ax.set_xlabel('Standard deviation')
ax.set_ylabel('Gini coefficient')
ax.legend()
plt.show()
```



0.2 Scipy 基础