

在简单的一维随机游走的例子中，只需要设置1个概率值，这是因为模型规定每次只能向左或向右走给定的距离。

如果一个系统可以在 N 种状态之间相互演化，就需要设置 N^2 个概率值，以只有3种状态的系统为例，需要设置的概率值有

$$\begin{bmatrix} q_{11} & q_{12} & q_{13} \\ q_{21} & q_{22} & q_{23} \\ q_{31} & q_{32} & q_{33} \end{bmatrix}$$

下标表示从一种状态演化到另一种状态或者保持不变，这个矩阵称为转移概率矩阵。

如果演化前体系处于3种状态的概率分别为 p_1, p_2, p_3 ，那么一次演化后的概率 p'_1, p'_2, p'_3 为

$$\begin{cases} p'_1 = p_1 \times q_{11} + p_2 \times q_{21} + p_3 \times q_{31} \\ p'_2 = p_1 \times q_{12} + p_2 \times q_{22} + p_3 \times q_{32} \\ p'_3 = p_1 \times q_{13} + p_2 \times q_{23} + p_3 \times q_{33} \end{cases}$$

用矩阵表示就是

$$\begin{bmatrix} p_1 & p_2 & p_3 \end{bmatrix} \begin{bmatrix} q_{11} & q_{12} & q_{13} \\ q_{21} & q_{22} & q_{23} \\ q_{31} & q_{32} & q_{33} \end{bmatrix} = \begin{bmatrix} p'_1 & p'_2 & p'_3 \end{bmatrix}$$

简记为 $\mathbf{P}(n)\mathbf{Q} = \mathbf{P}(n+1)$

当系统演化一定的步数后， $\mathbf{P}(n) = \mathbf{P}(0)\mathbf{Q}^n$

根据转移概率矩阵的性质，当 n 足够大时， \mathbf{Q}^n 会收敛到以下形式

$$\mathbf{Q}^n \rightarrow \begin{bmatrix} \pi_1 & \pi_2 & \pi_3 \\ \pi_1 & \pi_2 & \pi_3 \\ \pi_1 & \pi_2 & \pi_3 \end{bmatrix}$$

因此，不论初始分布 $\mathbf{P}(0)$ 如何，最终分布都是 $\mathbf{P}(n) = \boldsymbol{\pi} = (\pi_1, \pi_2, \pi_3)$ ，最终的分布可以由转移概率矩阵确定。

下面任意取一组转移概率来验证这一结论

In [1]:



```
1 import numpy as np
2 from numpy.random import default_rng
3 import matplotlib.pyplot as plt
4
5 # 设置转移概率矩阵
6 Q = np.array(
7     (
8         (0.5, 0.25, 0.25),
9         (0.8, 0.1, 0.1),
10        (0.3, 0.4, 0.3),
11    )
12 )
13
14 # 设置一个初始分布
15 pi = np.array((1.0, 0.0, 0.0))
16
17 # 设置演化步数
18 N_steps = 20
19
20 for _ in range(N_steps):
21     pi = pi.dot(Q)
22
23 print(f"N_steps: {N_steps}")
24 print(pi)
25 print(f"N_steps: {N_steps + 1}")
26 print(pi.dot(Q))
```

```
N_steps: 20
[0.52914798 0.24663677 0.22421525]
N_steps: 21
[0.52914798 0.24663677 0.22421525]
```

可以看到演化一定步数后，系统在各状态的分布不再变化

In [2]:



```
1 Qn = np.array(Q)
2
3 for _ in range(N_steps):
4     Qn = Qn.dot(Q)
5 print(Qn)
```

```
[[0.52914798 0.24663677 0.22421525]
 [0.52914798 0.24663677 0.22421525]
 [0.52914798 0.24663677 0.22421525]]
```

最终分布和 Q^n 的每一行一样

换一个转移概率矩阵

In [3]:

```
1 Q = np.array(  
2     (  
3         (0.2, 0.4, 0.4),  
4         (0.6, 0.2, 0.2),  
5         (0.1, 0.3, 0.6),  
6     )  
7 )  
8 pi = np.array((1.0, 0.0, 0.0))  
9 N_steps = 20  
10 for _ in range(N_steps):  
11     pi = pi.dot(Q)  
12     print(f"N_steps: {N_steps}")  
13     print(pi)  
14     print(f"N_steps: {N_steps + 1}")  
15     print(pi.dot(Q))
```

```
N_steps: 20  
[0.27659574 0.29787234 0.42553191]  
N_steps: 21  
[0.27659574 0.29787234 0.42553191]
```

换一个初始分布

In [4]:

```
1 pi = np.array((0.5, 0.5, 0.0))  
2 N_steps = 20  
3 for _ in range(N_steps):  
4     pi = pi.dot(Q)  
5     print(f"N_steps: {N_steps}")  
6     print(pi)  
7     print(f"N_steps: {N_steps + 1}")  
8     print(pi.dot(Q))
```

```
N_steps: 20  
[0.27659574 0.29787234 0.42553191]  
N_steps: 21  
[0.27659574 0.29787234 0.42553191]
```

In [5]:

```
1 Qn = np.array(Q)  
2  
3 for _ in range(N_steps):  
4     Qn = Qn.dot(Q)  
5     print(Qn)
```

```
[[0.27659574 0.29787234 0.42553191]  
 [0.27659574 0.29787234 0.42553191]  
 [0.27659574 0.29787234 0.42553191]]
```

可见最终的分布只和转移概率矩阵有关。

把以上结论应用到随机游走中，只要构造合适的转移概率矩阵，就可以让随机游走的结果服从特定的分布。

但是如果转移概率矩阵很大，计算起来就非常困难，此时可以在形式上定义完整的转移概率矩阵，只计算需要用

到的部分。

要想让随机游走在特定分布上保持平衡，只需要让此分布下任意两状态相互转化的概率相等即可，即

$$f(x)w(x \rightarrow x') = f(x')w(x' \rightarrow x)$$

这个条件称为细致平衡条件，在Metropolis方法中一般构造以下形式的过渡概率

$$w(x \rightarrow x') = \min \left[1, \frac{f(x')}{f(x)} \right]$$

使用Metropolis方法对随机数进行抽样的流程如下：

假设游走到达了位置 x ，选取一个试探位置 x'

计算 $r = \frac{f(x')}{f(x)}$ ，如果大于1，则 $w(x \rightarrow x') = 1$ ，接受这一步试探

否则 $w(x \rightarrow x') = r$ ，产生一个随机数 ξ ，如果 $\xi \leq r$ ，接受这一步试探

否则拒绝这一步试探，停在原地，进行下一步试探

因为Metropolis方法中计算的是密度函数的比值，所以不用考虑归一化系数，因此可以用来对难以归一化的密度函数进行采样。

上述流程没有考虑抽样区域的边界，如果抽样的区域有边界就需要做特殊处理。

用Metropolis方法对自由度为4的 χ^2 分布进行抽样。

$$f(x) = \frac{1}{2^{n/2} \Gamma(n/2)} x^{n/2-1} e^{-x/2}, x > 0, n = 4$$

在Metropolis方法中计算的是

$$\frac{f(x')}{f(x)} = \frac{x'}{x} \exp \left(\frac{x - x'}{2} \right)$$

如果不考虑边界附近的情况

In [6]:

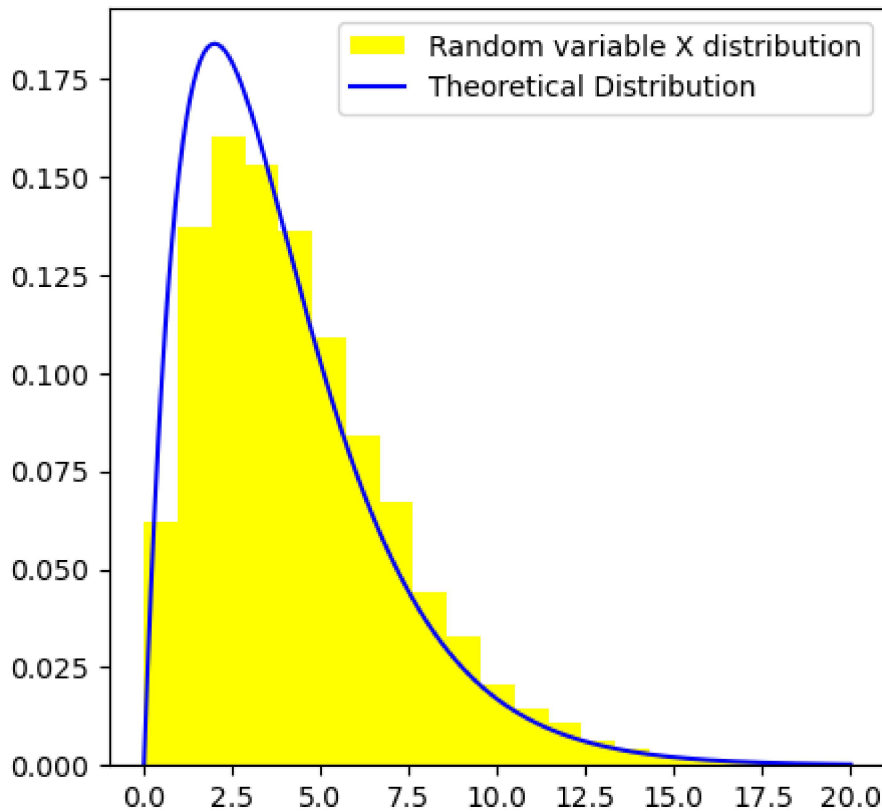


```
1 rng = default_rng()
2
3 def r_fx_fy(x, y):
4     return x/y*np.exp((y-x)/2)
5
6 # 总步数
7 N = 100000
8 # 每一步的步长范围
9 delta = 7
10
11 # 起点
12 x0 = 1e-5
13 walk_path = [x0]
14 for _ in range(N):
15     # 取一个试探位置
16     x_try = rng.uniform(x0-delta, x0+delta)
17     # 如果试探位置不在定义域内，则拒绝
18     if x_try <= 0:
19         continue
20     # 计算r，判断是否接受
21     r = r_fx_fy(x_try, x0)
22     if r > 1:
23         x0 = x_try
24         walk_path.append(x0)
25         continue
26     if r >= rng.random():
27         x0 = x_try
28         walk_path.append(x0)
29         continue
30     walk_path.append(x0)
31
```

In [7]:



```
1 # 理论密度曲线
2 def chi2_2(x):
3     return x/4*np.exp(-x/2)
4
5 x = np.linspace(0, 20, 201)
6 y = chi2_2(x)
7
8 plt.figure(figsize=(5, 5), dpi=100)
9 plt.hist(walk_path, bins=21, range=(0.0, 20.0), density=True, color='yellow', label='Random va
10 plt.plot(x, y, color='blue', label='Theoretical Distribution')
11 plt.legend(loc='upper right')
12 plt.show()
```



可以看到边界附近的偏差较大。

对于单边有界的情况，可以做对称处理，从而允许游走进入定义域外的区域

In [8]:



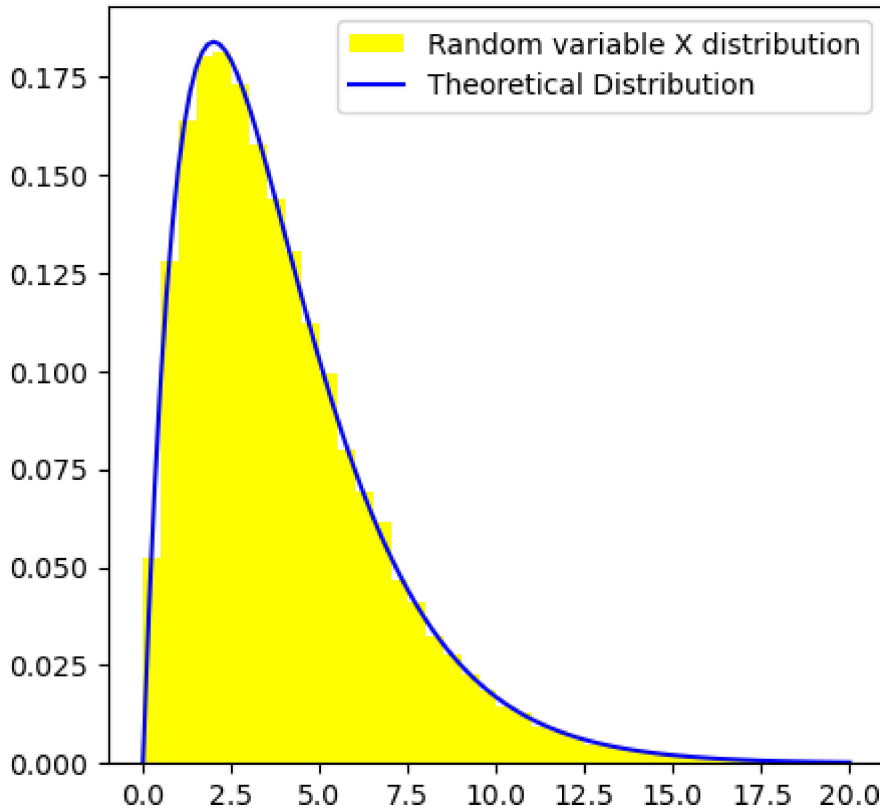
```
1 def r_fx_fy(x, y):
2     x = abs(x)
3     y = abs(y)
4     return x/y*np.exp((y-x)/2)
5
6 # 总步数
7 N = 100000
8 # 每一步的步长范围
9 delta = 15
10
11 # 起点
12 x0 = 1e-5
13 walk_path = [x0]
14
15 # 用于记录步数
16 n_accept = 0
17
18 for _ in range(N):
19     # 取一个试探位置
20     x_try = rng.uniform(x0-delta, x0+delta)
21     # 计算r, 判断是否接受
22     r = r_fx_fy(x_try, x0)
23     if r > 1:
24         n_accept += 1
25         x0 = x_try
26         walk_path.append(abs(x0))
27         continue
28     if r >= rng.random():
29         n_accept += 1
30         x0 = x_try
31         walk_path.append(abs(x0))
32         continue
33     walk_path.append(abs(x0))
34
35 print(f"n_accept/N={n_accept/N:.4}")
```

n_accept/N=0.4542

In [9]:



```
1 x = np.linspace(0, 20, 121)
2 y = chi2_2(x)
3
4 plt.figure(figsize=(5, 5), dpi=100)
5 plt.hist(walk_path, bins=40, range=(0.0, 20.0), density=True, color='yellow', label='Random va
6 plt.plot(x, y, color='blue', label='Theoretical Distribution')
7 plt.legend(loc='upper right')
8 plt.show()
```



对于双边有界的情况可以设置周期性边界条件，将定义域外的试探平移到定义域内
用Metropolis方法对以下分布进行抽样

$$f(x) = Ax^2e^{-x}, x \in [0, 4]$$

In [10]:



```
1 def r_fx_fy(x, y):
2     return x*x/y/y*np.exp(y-x)
3
4 # 用于将试探点移动到定义域内
5 def bc(x, b1, b2):
6     length = b2 - b1
7     if x < b1:
8         return bc(x+length, b1, b2)
9     if x > b2:
10        return bc(x-length, b1, b2)
11    return x
12
13 # 总步数
14 N = 100000
15 # 每一步的步长范围
16 delta = 1
17
18 # 起点
19 x0 = 1e-5
20 walk_path = [x0]
21
22 # 用于记录步数
23 n_accept = 0
24
25 for _ in range(N):
26     # 取一个试探位置
27     x_try = rng.uniform(x0-delta, x0+delta)
28     x_try = bc(x_try, 0, 4)
29     # 计算r, 如果大于1, 则接受
30     r = r_fx_fy(x_try, x0)
31     if r > 1:
32         n_accept += 1
33         x0 = x_try
34         walk_path.append(x0)
35         continue
36     if r >= rng.random():
37         n_accept += 1
38         x0 = x_try
39         walk_path.append(x0)
40         continue
41     walk_path.append(x0)
42
43 print(f"n_accept/N={n_accept/N:.4}")
```

n_accept/N=0.8553

In [11]:



```
1 def fx(x):
2     return x*x*np.exp(-x)/1.5238
3
4 x = np.arange(0, 4, 0.1)
5 y = fx(x)
6
7 plt.figure(figsize=(5, 5), dpi=100)
8 plt.hist(walk_path, bins=40, range=(0.0, 4.0), density=True, color='yellow', label='Random var
9 plt.plot(x, y, color='blue', label='Theoretical Distribution')
10 plt.legend(loc='upper left')
11 plt.ylim((0, 0.5))
12 plt.show()
```

