信息网络建模与仿真

实验一实验报告

班级：2019211123

姓名：谷泽昆

学号：2019210194

2021 年 10 月

1.实验目标

1.1 深入理解均匀分布随机变量产生方法；

1.2 掌握由均匀分布随机变量产生其他分布随机变量的方法；

1.3 掌握两种常用的高斯分布随机变量的产生方法；

1.4 掌握泊松过程的产生方法；

2.实验内容

本实验主要实现了不同随机变量的产生与验证，包括利用线性同余法、梅森旋转法和 MRG32k3a 算法生成（0,1）区间上的均匀分布随机数，利用反函数法产生满足 Pareto 分布的随机数，利用中心极限定理和 Box-Muller 算法产生标准高斯分布随机变量，利用组合法产生目标要求的混合高斯随机变量以及产生泊松过程。

3.实验步骤与结果

3.1 均匀分布随机数的产生

3.1.1 线性同余法产生均匀分布随机数（三级标题，小四宋体加粗）

（一）算法分析

随机数生成：

线性同余法是产生伪随机数的方法，它是采用如下递归公式产生的：

$$x_{n+1} = [ax_n + b] mod(m)$$

线性同余法也分为混合同余法、加同余法和乘同余法，本实验采用 $a! = 0, b = 0$ 的乘同余法 mcg16807 实现随机数的产生。其中 a=16807，m=2147483647。我们利用数组+for 循环即可实现这一算法。

之后，利用 CPP 的文件流 fstream 将数据写入到 Linear.txt 中，方便后续使用 matlab 进行结果验证。

结果验证算法：

由于实际情况中的随机数都是离散的，因此描述随机变量的 CDF（Cumulative Distribution Function, 累积分布函数）不能用数学意义上的积分来完成，只能通过累加逼近积分操作，因此需要一个统计区间。我们将 CDF 的值域[0,1]分为 rangeNum 份，每个小区间为宽度为 1/rangeNum，我们使用 for 循环对落在前 i 个小区间的随机数进行统计，再除以数据总数，即可完成 CDF 的计算。

接下来计算 PDF（Probability Density Function, 概率密度函数）。数学上由 CDF 计算 PDF 时需要对 CDF 进行微分。由于我们计算的 CDF 是离散的，因此我们需要用差分来近似微分。利用相邻两个 CDF 值的差值再除以小区间宽度 1/rangeNum 即可得到该点的 PDF 值。但由于 CDF 离散，故 PDF 也为离散的，因此使用 ksdensity 进行平滑处理，可以使效果更加直观。

随机数的方差、均值等统计量可通过 var()、mean()完成。

（二）程序设计

随机数产生算法：

```
#include <stdio.h>
#include <math.h>
#include "mt19937ar.h"
#include <fstream>
#include <random>
using namespace std;

#define pi 3.1415926
#define LENGTH 20000
#define uint unsigned int

void LCG(int seed,unsigned int* vector, int length);

void main(){
    ofstream ofile;
    ofile.open("Linear.txt", ios::out);
    unsigned int res[LENGTH];
    LCG(1, res, LENGTH);
    double random01[LENGTH];
    //printf("0-1");
    for (int i = 0;i < LENGTH;i++) {
        random01[i] = double(res[i]) / double(2147483647);
```

```
        //printf("%f\r\n", random01[i]);
        ofile << random01[i] << endl;
    }
    }
    ofile.close();
}

void LCG(int seed, unsigned int *vector,int length)
{
    vector[0] = seed;
    for (int i = 1;i < length;i++) {
        vector[i] = (16807 * vector[i - 1]) % unsigned int(2147483647);
        //printf("%d\r\n",vector[i]);
    }
}
```

结果验证算法：

```
clear;clc;

rangeNum = 3000;
figure(1)
data = load("Linear.txt");
subplot(1,3,1);
plot(data)
title("srcData");

sigma2 = var(data);
u = mean(data);

disp(['均值:',num2str(u),'      误差:',num2str(abs(u-0.5)/0.5*100),'%']);

disp(['方差:',num2str(sigma2),'      误差:',num2str(abs(sigma2-1/12)*12*100),'%']);

CDF = zeros(1,rangeNum);

for i = 1:rangeNum
    CDF(i) = length(find(data<max(data)/rangeNum*i))/length(data);
end
targetCDF=linspace(0,1,rangeNum);

subplot(1,3,2);
plot(CDF);
```

```
hold on;
plot(targetCDF)
legend("仿真值","理论值")
title("CDF(累积分布函数)")
hold off

text(0,0.1,{"均值:",num2str(u)});
text(0,0.4,{"方差:",num2str(sigma2)});

PDF=zeros(1,rangeNum-1);

for i = 1:rangeNum-1
    PDF(i) = (CDF(i+1)-CDF(i))*rangeNum;
end
targetPDF=ones(1,rangeNum-1);

subplot(1,3,3);
plot(PDF);
hold on;
plot(targetPDF);
%hold on;
%plot(ksdensity(data));
legend("仿真值","理论值")
title("PDF(概率密度函数)")
u_PDF = mean(PDF);
disp(['概率密度仿真值:',num2str(u_PDF),'      误差:',num2str(abs(u_PDF
-1)*100),'%'])
hold off

figure(2);
[y,x]=ksdensity(data);
plot(x,y);
hold on;
plot(x,ones(1,length(x)));
legend("仿真值（平滑处理）","理论值");
hold off;
```

（三）测试结果

随机数：

图 1.1.1

结果分析：

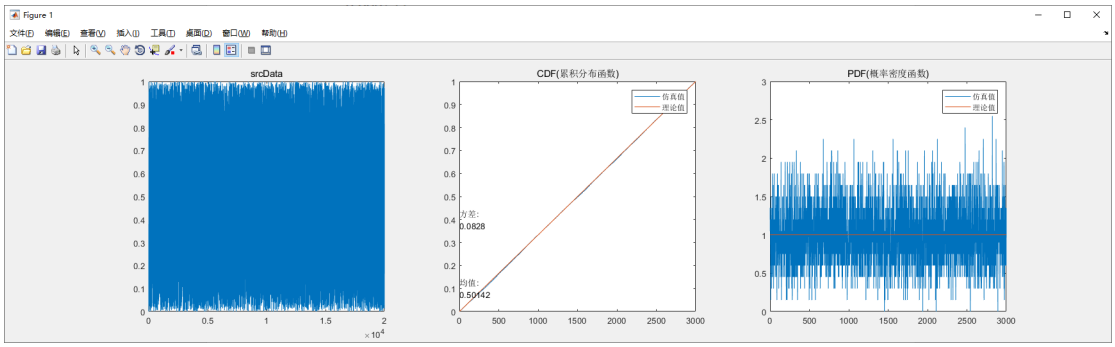

图 1.1.2

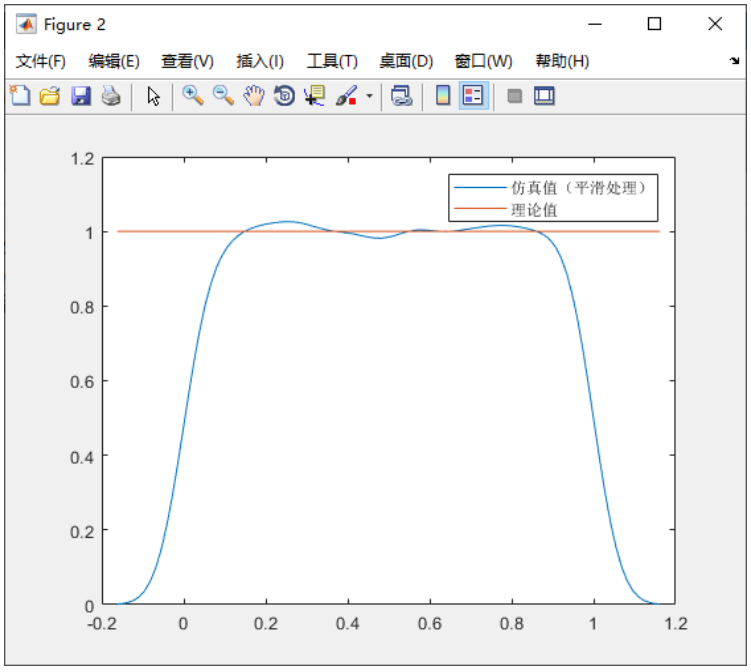

图 1.1.3

图 1.1.4

通过观察图 1.1.3 我们得知：仿真的（0,1）区间均匀分布的随机数的 PDF 很接近理论值 1，因此这可以证明我们的算法无误，同时，极小的误差同样可以说明这一点。

3.1.2Mersenne Twister 方法产生均匀分布随机数

（一）算法分析

随机数生成：

梅森旋转法（Mersenne Twister）是世界上迄今为止最好的随机数发生器之一。它遵循以下递推公式：

$$x_{k+n} := x_{k+m} \oplus (x_k^u | \ x_{k+1}^l) A, k = 0,1,\ldots$$

其中 x_i 是 w 位的字向量，

$(x_k^u \mid x_{k+1}^l)$：将 x_k 的前 w-r 位与 x_{k+1} 的后 r 位连接，组成一个新的 w 位字向量。

$$A = R = \begin{pmatrix} 0 & I_{w-1} \\ a_{w-1} & (a_{w-2},\ldots a_0) \end{pmatrix}$$

temper:为改善均匀性进一步引入运算：

$$\begin{aligned} y &:= x \oplus (x \gg u) \\ y &:= y \oplus ((y \ll s) \& b) \\ y &:= y \oplus ((y \ll t) \& c) \\ z &:= y \oplus (y \gg l) \end{aligned}$$

MT19937-32 参数：

$$\begin{aligned} &(w,n,m,r) = (32,624,397,31) \\ &a = 9908B0DF_{16} \\ &f = 1812433253 \\ &(u,d) = (11, FFFFFFFF_{16}) \\ &(s,b) = (7, 9D2C568016_{16}) \\ &(t,c) = (15, EFC6000016_{16}) \\ &l = 18 \end{aligned}$$

梅森旋转法有以下优点：

- 周期长
- 随机性好

结果验证：

与 LCG 相同

（二）程序代码分析

分析见注释

T2.cpp

调用 mt19937ar.h 的 genrand_real3()产生随机数，在使用 fstream 将数据写入 MT.txt 中

```
#include <stdio.h>
#include <math.h>
#include "mt19937ar.h"
#include "MT.h"
#include <fstream>
#include <time.h>
using namespace std;

#define pi 3.1415926
#define LENGTH 20000
#define uint unsigned int

void main()
{
    ofstream ofile;
    ofile.open("MT.txt", ios::out);
    int i;
    unsigned long init[4] = { 0x123, 0x234, 0x345, 0x456 }, length =
 4;
    init_by_array(init, length);
    for (i = 0; i < LENGTH; i++) {
        //printf("%10.8f ", genrand_real2());
        ofile << genrand_real2() << endl;
    }
    ofile.close();
    /*另一种方法
    ofstream ofile;
    ofile.open("MT.txt", ios::out);
    uint mt[MT_199332_N];
```

```
    uint index = 0;
    int n = MT_199332_N;

    mt_seed(time(NULL), mt, n);
    for(int i =0;i<LENGTH;i++)
    {
        ofile<<getrand_01(mt, n, &index)<<endl;
    }
    ofile.close();
    /*

}
```

mt19937ar.h

头文件

```
/* initializes mt[N] with a seed */
void init_genrand(unsigned long s);

/* initialize by an array with array-length */
/* init_key is the array for initializing keys */
/* key_length is its length */
void init_by_array(unsigned long init_key[], int key_length);

/* generates a random number on [0,0xffffffff]-interval */
unsigned long genrand_int32(void);

/* generates a random number on [0,0x7fffffff]-interval */
long genrand_int31(void);

/* These real versions are due to Isaku Wada, 2002/01/09 added */
/* generates a random number on [0,1]-real-interval */
double genrand_real1(void);

/* generates a random number on [0,1)-real-interval */
double genrand_real2(void);

/* generates a random number on (0,1)-real-interval */
double genrand_real3(void);

/* generates a random number on [0,1) with 53-bit resolution*/
double genrand_res53(void);
```

mt19937ar.cpp

源文件

```
#include <stdio.h>
#include "mt19937ar.h"

/* Period parameters */
#define N 624
#define M 397
#define MATRIX_A 0x9908b0dfUL   /* constant vector a */
#define UPPER_MASK 0x80000000UL /* most significant w-r bits */
#define LOWER_MASK 0x7fffffffUL /* least significant r bits */

static unsigned long mt[N]; /* the array for the state vector  */
static int mti = N + 1; /* mti==N+1 means mt[N] is not initialized *
/

/* initializes mt[N] with a seed */
void init_genrand(unsigned long s)
{
    mt[0] = s & 0xffffffffUL;
    for (mti = 1; mti < N; mti++) {
        mt[mti] =
            (1812433253UL * (mt[mti - 1] ^ (mt[mti - 1] >> 30)) + mt
i);
        /* See Knuth TAOCP Vol2. 3rd Ed. P.106 for multiplier. */
        /* In the previous versions, MSBs of the seed affect   */
        /* only MSBs of the array mt[].                        */
        mt[mti] &= 0xffffffffUL;
        /* for >32 bit machines */
    }
}

/* initialize by an array with array-length */
/* init_key is the array for initializing keys */
/* key_length is its length */
/* slight change for C++, 2004/2/26 */
void init_by_array(unsigned long init_key[], int key_length)
{
    int i, j, k;
    init_genrand(19650218UL);
    i = 1; j = 0;
    k = (N > key_length ? N : key_length);
    for (; k; k--) {
        mt[i] = (mt[i] ^ ((mt[i - 1] ^ (mt[i - 1] >> 30)) * 1664525U
L))
```

```
            + init_key[j] + j; /* non linear */
        mt[i] &= 0xffffffffUL; /* for WORDSIZE > 32 machines */
        i++; j++;
        if (i >= N) { mt[0] = mt[N - 1]; i = 1; }
        if (j >= key_length) j = 0;
    }
    for (k = N - 1; k; k--) {
        mt[i] = (mt[i] ^ ((mt[i - 1] ^ (mt[i - 1] >> 30)) * 156608394
1UL))
            - i; /* non linear */
        mt[i] &= 0xffffffffUL; /* for WORDSIZE > 32 machines */
        i++;
        if (i >= N) { mt[0] = mt[N - 1]; i = 1; }
    }

    mt[0] = 0x80000000UL; /* MSB is 1; assuring non-zero initial arra
y */
}

/* generates a random number on [0,0xffffffff]-interval */
// 使用梅森旋转法产生随机数
unsigned long genrand_int32(void)
{
    unsigned long y;
    static unsigned long mag01[2] = { 0x0UL, MATRIX_A };
    /* mag01[x] = x * MATRIX_A  for x=0,1 */

    if (mti >= N) { /* generate N words at one time */
        int kk;

        if (mti == N + 1)   /* if init_genrand() has not been called,
 */
            init_genrand(5489UL); /* a default initial seed is used */

        for (kk = 0;kk < N - M;kk++) {
            y = (mt[kk] & UPPER_MASK) | (mt[kk + 1] & LOWER_MASK);
            mt[kk] = mt[kk + M] ^ (y >> 1) ^ mag01[y & 0x1UL];
        }
        for (;kk < N - 1;kk++) {
            y = (mt[kk] & UPPER_MASK) | (mt[kk + 1] & LOWER_MASK);
            mt[kk] = mt[kk + (M - N)] ^ (y >> 1) ^ mag01[y & 0x1UL];
        }
        y = (mt[N - 1] & UPPER_MASK) | (mt[0] & LOWER_MASK);
        mt[N - 1] = mt[M - 1] ^ (y >> 1) ^ mag01[y & 0x1UL];
```

```
        mti = 0;
    }

    y = mt[mti++];

    /* Tempering */
     // temper 代码思路及数值选择见上述算法分析
    y ^= (y >> 11);
    y ^= (y << 7) & 0x9d2c5680UL;
    y ^= (y << 15) & 0xefc60000UL;
    y ^= (y >> 18);

    return y;
}

/* generates a random number on [0,1]-real-interval */
double genrand_real1(void)
{
    return genrand_int32() * (1.0 / 4294967295.0);
    /* divided by 2^32-1 */
}

/* generates a random number on [0,1)-real-interval */
double genrand_real2(void)
{
    return genrand_int32() * (1.0 / 4294967296.0);
    /* divided by 2^32 */
}

/* generates a random number on (0,1)-real-interval */
double genrand_real3(void)
{
    return (((double)genrand_int32()) + 0.5) * (1.0 / 4294967296.0);
    /* divided by 2^32 */
}
```

自己编写的 MT.h

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
```

```h
typedef unsigned int uint;

#define MT_199332_N 624
#define MT_199332_F 1812433253U
#define MT_199332_W 32
#define MT_199332_R 31
#define MT_199332_M 397
#define MT_199332_A 0x9908B0DF
#define MT_199332_U 11
#define MT_199332_D 0xffffffff
#define MT_199332_S 7
#define MT_199332_B 0x9d2c5680
#define MT_199332_T 15
#define MT_199332_C 0xefc60000
#define MT_199332_L 18
#define MT_199332_LOWER_MASK ((1U<<MT_199332_R)-1)
#define MT_199332_UPPER_MASK (1U<<MT_199332_R)

uint mt_seed(uint seed, uint *mt, int n);
uint mt_twist(uint *mt, int n);
uint mt_extract(uint *mt, int n, uint *index);

double getrand_01(uint *mt, int n, uint *index);
```

自己编写的 MT.c

```c
#include "MT.h"
uint mt_seed(uint seed, uint *mt, int n)
{
    int i;

    memset(mt, 0x00, n*sizeof(uint));

    for(i=1; i<n; i++)
    {
        mt[i] = MT_199332_F * (mt[i-1]^mt[i-1]>>(MT_199332_W-2))+i;
    }

    return mt[n-1];
}

uint mt_twist(uint *mt, int n)
{
    uint x, y;
    int i;
```

```
    for(i=0; i<n; i++)
    {
        x = (uint)((mt[i] & MT_199332_UPPER_MASK) + (mt[(i+1)%n] & M
T_199332_LOWER_MASK));
        y = (x >> 1);

        if((x & 1) != 0)
        {
            y = (y ^ MT_199332_A);
        }

        mt[i] = (mt[(i+MT_199332_M)%n] ^ y);
    }

    return 0;
}

uint mt_extract(uint *mt, int n, uint *index)
{
    uint x;

    if(*index >= n)
    {
        mt_twist(mt, n);
    }

    x = x ^ x >> MT_199332_U & MT_199332_D;
    x = x ^ x << MT_199332_S & MT_199332_B;
    x = x ^ x << MT_199332_T & MT_199332_C;
    x = x ^ x >> MT_199332_L;

    (*index)++;

    return x;
}

double getrand_01(uint *mt, int n, uint *index)
{
    return (((double)mt_extract(mt,n, index)) + 0.5) * (1.0 / 429496
7296.0);
}
```

（三）测试结果与分析

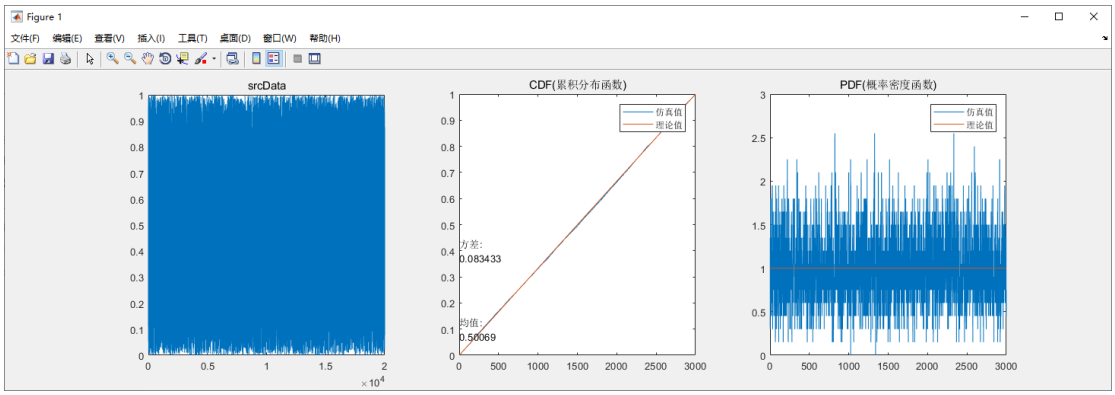

图 1.2.1

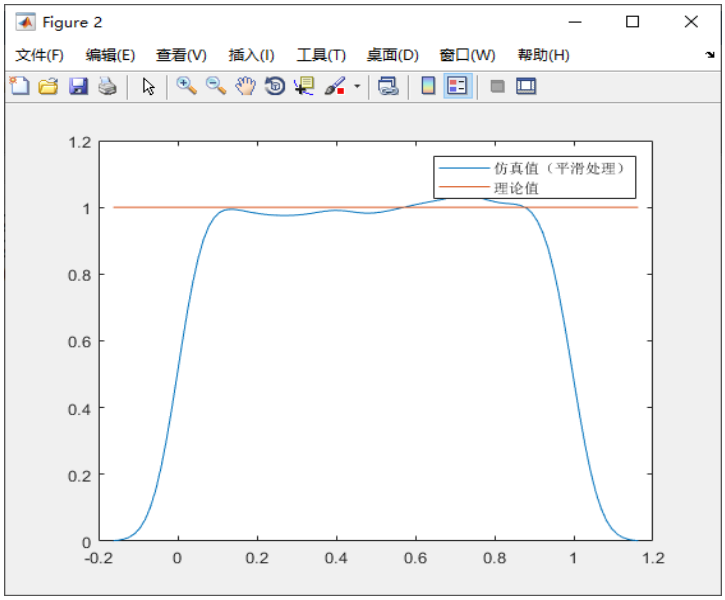

图 1.2.2

从 matlab 绘制的图像中可以看出，仿真计算得出的 CDF 十分接近真实的（0,1）均匀分布的 CDF，而 PDF 则有较大起伏，但在总体上可以看出其均值大概为 1。出现这种现象是因为仿真产生的 CDF 只是逼近一条直线，其实际为一条折线，因此对其求差分时，难免会出现不同的值，这才有了右图中的高低起伏。使用 ksdensity 平滑之后可以看到很接近理论值。我们对数据进一步分析，可以得出其 srcData 的均值与方差，以及 PDF 的均值，如图 1.2.3：

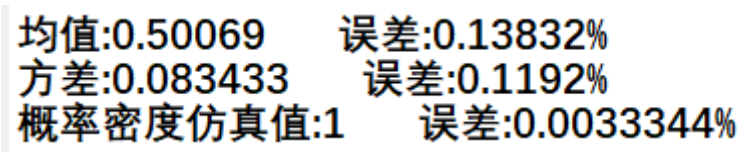

图 1.2.3

因此，梅森旋转法可以满足产生（0,1）均匀分布的随机数的要求。

3.1.3MRG32k3a 方法产生均匀分布随机数

（一）算法分析

随机数生成

通过阅读论文《AN OBJECT-ORIENTED RANDOM-NUMBER PACKAGE WITH MANY LONG STREAMS AND SUBSTREAMS》，我们可以了解 MRG32k3a 产生随机数的方法。

在“1.DESCRIPTION AND IMPLEMENTATION OF THE SOFTWARE”中，我们可以了解该方法产生随机数的详细步骤。

$$x_{1,n} = (1403580 \times x_{1n-2} - 810728 \times x_{1,n-3}) mod m_1$$
$$x_{2,n} = (527612 \times x_{2,n-1} - 1370589 \times x_{2,n-3}) mod m_2$$
$$m_1 = 2^{32} - 209 = 4294967087 m2 = 2^{32} - 22853 = 4294944443$$
$$z_n = (x_{1,n} - x_{2,n}) mod 4,294,967,087$$
$$u_n = \begin{cases} z_n / 4294967088 & \text{if } z_n > 0 \\ 4294967087 / 4294967088 & \text{if } z_n = 0 \end{cases}$$

我们仍可以通过 for 循环来实现该算法。首先，由于 x 的产生需要三个初始值，因此，第一条公式需要在第四次循环才可以执行，x1，x2 计算完毕后，再计算 Zn，最后根据 Un 与 Zn 的对应关系即可得出最后输出 Un。

结果验证

与 LCG 相同

（二）程序代码分析

T1.3.cpp

分析见注释

```
#include <stdio.h>
#include <math.h>
#include <fstream>
#include <random>
using namespace std;

#define pi 3.1415926     //定义pi 的值
#define LENGTH 20000     //定义产生随机数的个数
#define uint unsigned int   //定义uint，方便后续代码编写

void main()
{
```

```
    ofstream ofile;//使用文件流
    ofile.open("MRG32k3a.txt", ios::out);//打开MRG32k3a.txt

    uint x1[LENGTH];//初始化内存
    uint x2[LENGTH];
    uint zn[LENGTH];
    double un[LENGTH];
    x1[0] = 100, x1[1] = 500, x1[2] = 1000;//初始化 xn
    x2[0] = 10, x2[1] = 5, x2[2] = 100;
    for (int i = 0;i < LENGTH;i++)//进入主循环，产生随机数
    {
        if (i >= 3)
        {
            x1[i] = ((unsigned int)1403580 * x1[i - 2] - (unsigned i
nt)810728 * x1[i - 3]) % (unsigned int)4294967087;
            x2[i] = ((unsigned int)527612 * x2[i - 1] - (unsigned in
t)1370589 * x2[i - 3]) % (unsigned int)4294944443;
        }
        zn[i] = (x1[i] - x2[i]) % (unsigned int)4294967087;
        if (zn[i] == 0)//根据 zn 计算 un
            un[i] = (unsigned int)4294967087 / (unsigned int)4294967
08;
        else
            un[i] = double(zn[i]) / (double)4294967088;

        ofile << un[i] << endl;//写入随机数，换行，便于matlab 读取
    }
    oflie.close();//关闭文件流
}
```

（三）测试结果与分析

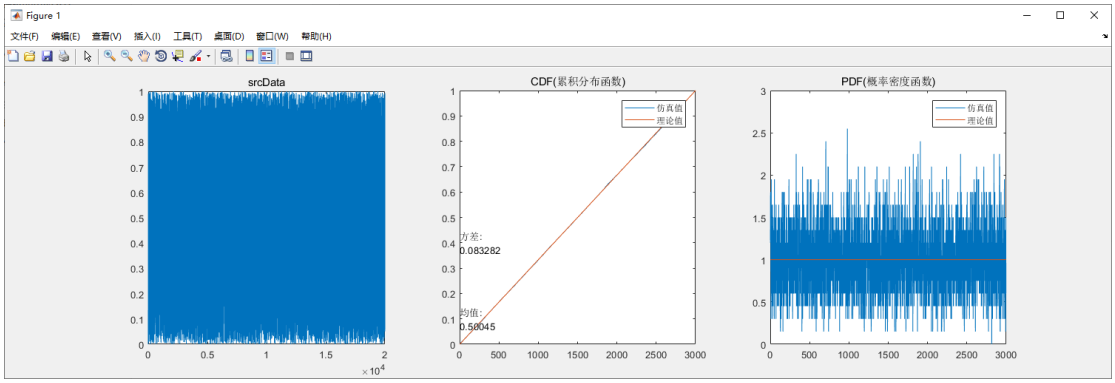

图 1.3.1

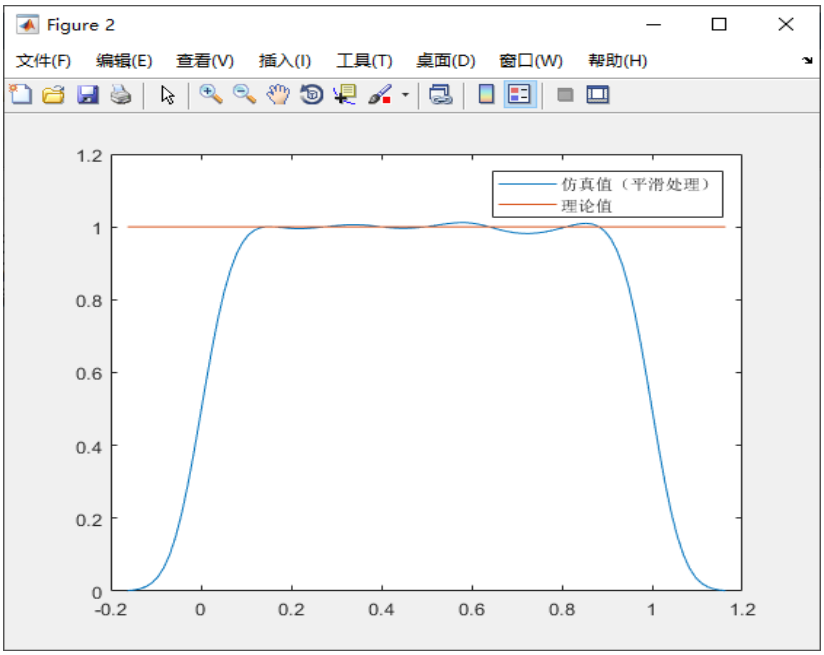

图 1.3.2

均值:0.50045　　误差:0.089952%
方差:0.083282　　误差:0.06177%
概率密度仿真值:0.99993　　误差:0.0066689%

图 1.3.3

对比

在 matlab 仿真中，我们计算了每种产生（0,1）均匀分布的随机数的算法的误差。总结如下：

算法	均值误差	方差误差	概率密度误差
线性同余法	0.28344%	0.64044%	0.031677%
梅森旋转法	0.13832%	0.1192%	0.0033344%
MRG32k3a	0.089952%	0.06177%	0.0066689%

可以看出，线性同余法的各项指标最差，均值、方差以及概率密度误差均最大；MRG32k3a 的均值与方差误差在三种算法中最小，但概率密度误差比梅森旋转法稍大。因此，我们可以推断：梅森旋转法和 MRG32k3a 的性能较优，线性同余法的性能最差。这也解释了为什么在一些编程语言的随机数产生模块中大多采用梅森旋转法或者 MRG32k3a 算法。

3.2 高斯分布随机变量的产生

3.2.1 中心极限定理法产生高斯分布随机变量

（一）算法分析

随机数产生

在信息网络仿真中会常常碰到高斯随机变量，经过上述实验，我们已经掌握（0,1）均匀分布的随机数的产生方法。通过概率论的学习，我们得知：中心极限定理指出，当$N \to \infty$时，N 个独立随机变量之和的分布趋近于高斯分布。因此，我们可以利用中心极限定理，采用以下步骤产生高斯随机变量：

- 产生若干个（0,1）区间上的均匀分布随机数U_i
- $U_i - \frac{1}{2}$ 服从（－1/2， 1/2 ）的均匀分布，方差为 1/12
- N 个$U_i - \frac{1}{2}$ 相加，近似服从均值为 0，方差为 N/12 的高斯分布
- 取 $B = \sqrt{\frac{12}{N}}$，则 $X = B\sum_{i=1}^{N}(U_i - \frac{1}{2})$ 近似服从标准高斯分布
- 一般，取：N=12

中心极限定理法的截尾现象

- 按上述参数得到的随机变量只能在区间(-6,6)区间上取值
- 利用中心极限定理方法产生高斯分布随机变量时，实际产生变量的范围总是无法达到高斯分布的尾部，这种现象称为截尾
- 增大 N 值，可以减小这种影响，却会增大计算量

结果验证

matlab 代码与（0,1）均匀分布的随机数稍有不同，为了更好的显示效果，我们将 rangeNum 增加至 50000，同时，我们在进行 CDF 的计算时，将查找区间进行平移，修改后为 max(data0)/rangeNum(i-0.5rangeNum)))/length(data0)。这样就可以对负数进行查找，使满足高斯分布的特点。

（二）程序代码分析

```
#include <stdio.h>
#include <math.h>
#include "mt19937ar.h"
#include <fstream>
#include <random>
```

```
using namespace std;

#define pi 3.1415926
#define LENGTH 20000
#define uint unsigned int

void centralLimit();

void main()
{
    centralLimit();
}

void centralLimit()
{
    //按照PPT中的基本步骤，这里N取值为12
    //因此，公式得到简化
    ofstream ofile;
    ofile.open("centralLimit.txt", ios::out);
    int i;
    unsigned long init[4] = { 0x123, 0x234, 0x345, 0x456 }, length =
4;//初始化种子seed
    init_by_array(init, length);
    for (i = 0; i < LENGTH; i++) {
        double temp = 0;
        for (int j = 0;j < 12;j++)
        {
            temp += genrand_real2();
        }
        ofile << temp - 6.0 << endl;//B=1，故可以直接输出到txt
    }
    ofile.close();
}
```

结果验证

```
clear;clc;

rangeNum = 50000;

data0 = load("centralLimit.txt");
data1 = load("BoxMuller.txt");
data1 = data1(:,1);

figure(1);
```

```
subplot(2,3,1);
plot(data0)
title("srcData(中心极限定理)");

subplot(2,3,4);
plot(data1)
title("srcData(BoxMuller)");

sigma2_0 = var(data0);
u0 = mean(data0);

sigma2_1 = var(data1);
u1 = mean(data1);

disp(["均值:",num2str(u0)]);
disp(["方差:",num2str(sigma2_0)]);

disp(["均值:",num2str(u1)]);
disp(["方差:",num2str(sigma2_1)]);

CDF0 = zeros(1,rangeNum);
CDF1 = zeros(1,rangeNum);

for i = 1:rangeNum
    CDF0(i) = length(find(data0<max(data0)/rangeNum*(i-0.5*rangeNu
m)))/length(data0);
    CDF1(i) = length(find(data1<max(data1)/rangeNum*(i-0.5*rangeNu
m)))/length(data1);
end

subplot(2,3,2);
plot(CDF0);
title("CDF(累积分布函数)")
text(0,0.1,{"均值:",num2str(u0)});
text(20000,0.1,{"绝对误差:",num2str(abs(u0))});
text(0,0.4,{"方差:",num2str(sigma2_0)});
text(20000,0.4,{"绝对误差:",num2str(abs(sigma2_0-1))});

subplot(2,3,5);
plot(CDF0);
title("CDF(累积分布函数)")
text(0,0.1,{"均值:",num2str(u1)});
text(20000,0.1,{"绝对误差:",num2str(abs(u1))});
text(0,0.4,{"方差:",num2str(sigma2_1)});
```

```
text(20000,0.4,{"绝对误差:",num2str(abs(sigma2_1-1))});


PDF0=zeros(1,rangeNum-1);
PDF1=zeros(1,rangeNum-1);

for i = 1:rangeNum-1
    PDF0(i) = (CDF0(i+1)-CDF0(i))*rangeNum;
    PDF1(i) = (CDF1(i+1)-CDF1(i))*rangeNum;
end

subplot(2,3,3);
plot(PDF0)
hold on;
x = linspace(-2,2,rangeNum-1);
y = normpdf(x,0,1);
plot(y*25)
hold off;
legend('仿真值','理论值');
title("PDF(概率密度函数)")

subplot(2,3,6);
plot(PDF1)
hold on;
plot(y*25)
hold off;
legend('仿真值','理论值');
title("PDF(概率密度函数)")

figure(2);
[y0,x0]=ksdensity(data0);
[y1,x1]=ksdensity(data1);
subplot(1,2,1);
plot(x0,y0);
hold on;
plot(x0,normpdf(x0));
title("srcData(中心极限定理)");
legend("仿真值（平滑处理）","理论值");
hold off;

subplot(1,2,2);
plot(x1,y1);
hold on;
plot(x1,normpdf(x1));
```

```
title("srcData(BoxMuller)");
legend("仿真值（平滑处理）","理论值");
hold off;
```

（三）测试结果与分析

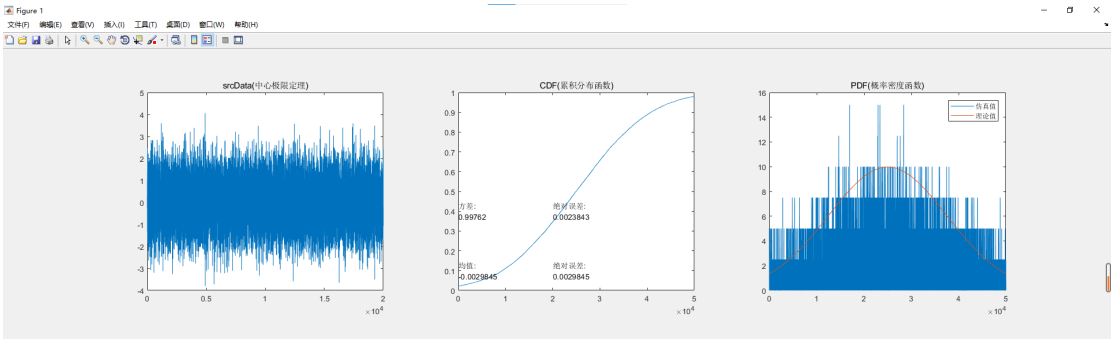

图 2.1.1

可以看出 CDF 的确满足理论上的高斯随机变量 CDF，PDF 同样由于 CDF 的离散性，因此会出现阶梯状分布，但其总体分布依然符合高斯随机变量的 PDF，证明我们的算法没有问题。

接着我们使用 ksdensity 进行平滑处理，如图 2.1.2：

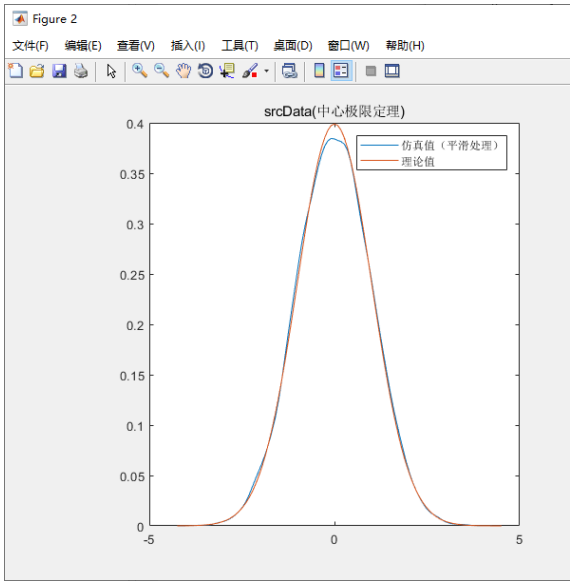

图 2.1.2

3.2.2 Box-Muller 法产生高斯分布随机变量

（一）算法分析

随机数产生

Box-Muller 方法的步骤

- 产生两个独立的(0,1)区间上均匀分布随机变量 u1 和 u2；
- 根据瑞利分布的累积分布函数，对 u1 进行反变换
- $F_R(r)=1-\exp(-\frac{r^2}{2\sigma^2}),r>0$　　u_1即$1-F_R(r)$
- 产生两个独立的、均值为 0 方差为 1 的高斯分布随机变量。

$$X=\sqrt{-2\ln u_1}\cos(2\pi u_2)$$
$$Y=\sqrt{-2\ln u_1}\sin(2\pi u_2)$$

结果验证

同中心极限定理

（二）程序代码分析

随机数产生

```
#include <stdio.h>
#include <math.h>
#include "mt19937ar.h"
#include <fstream>
#include <random>
using namespace std;

#define pi 3.1415926
#define LENGTH 20000
#define uint unsigned int

void Box_Muller();

void main()
{
    Box_Muller();
}

void Box_Muller()
{
   ofstream ofile;
```

```
    ofile.open("BoxMuller.txt", ios::out);
    int i;
    unsigned long init[4] = { 0x123, 0x234, 0x345, 0x456 }, length =
4;
    init_by_array(init, length);
    for (i = 0; i < LENGTH; i++) {
        double u1, u2;
        u1 = genrand_real2();
        u2 = genrand_real2();
        ofile << sqrtf(-2.0f * logf(u1)) * cosf(2.0f * pi * u2) <<" "
<< sqrtf(-2.0 * logf(u1)) * sinf(2.0 * pi * u2) << endl;

    }
    ofile.close();
}
```

结果验证

同中心极限定理

（三）测试结果与分析

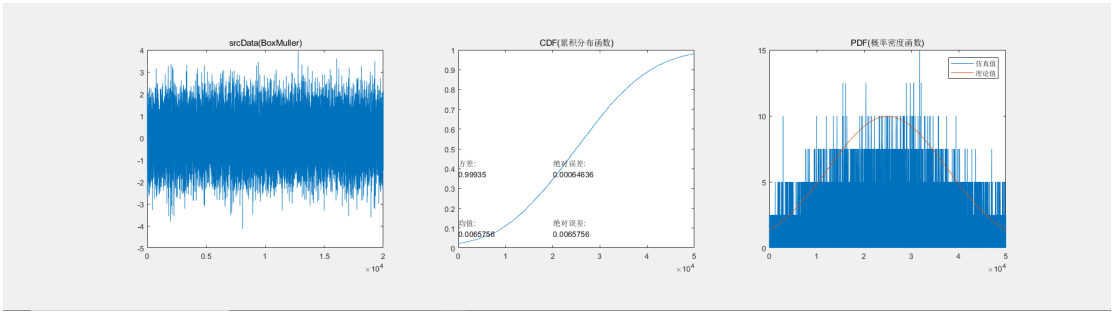

图 2.2.1

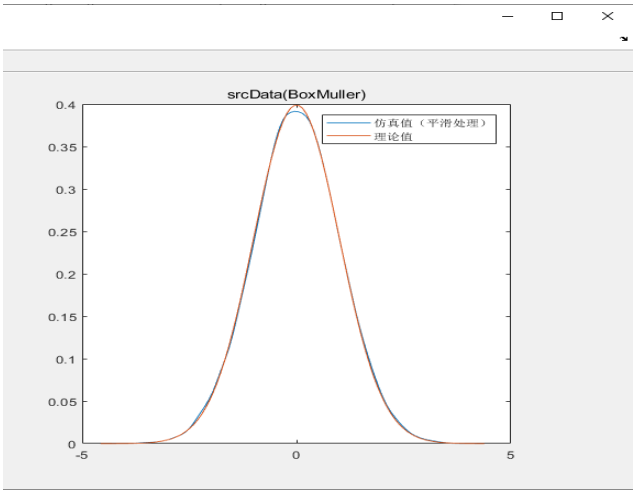

图 2.2.2

除部分在中心极限定理已经提及的图像解释以外，我们对相应统计变量进行分析。

算法	均值绝对误差	方差绝对误差
中心极限定理	0.0029845	0.0023843
Box-Muller	0.0065756	0.00064636

可以看出，中心极限定理的均值绝对误差较小，但 Box-Muller 的方差绝对误差较小。再加上 N=12 时的中心极限定理方法计算量较小，因此在一些对数据不敏感的场合下可以采用中心极限定理方法，以提高计算速度；在对数据要求较高的情况下，可以采用 Box-Muller 法。

3.3 帕累托分布随机变量的产生

（一）算法分析

随机数产生

反变换法是一种最常用、最直观的随机变量产生方法，是将累积分布函 数的反函数作用于均匀分布随机变量产生目标随机变量的方法。利用反函数法产生随机变量的前提是目标随机变量的累积分布函数具有闭式表达式，都可以得到其反函 数，其反函数的定义域为[0,1]区间。

由均匀分布的不相关随机序列 U 经过如下变换即可获得具有概 率分布函数的不相关序列：

$$X = F_X^{-1}(U)$$

我们已知 Pareto 分布的 PDF，通过积分即可获得其 CDF：

$$p(x) = \frac{ab^a}{x^{a+1}} \qquad b \le x < \infty$$

$$F(x) = \int_{-\infty}^{x} p(t)dt = 1 - \frac{b^a}{x^a}$$

再计算其反变换：

$$F^{-1}(u)=\frac{b}{\sqrt[a]{1-u}}=\frac{b}{\sqrt[a]{u}}$$

Pareto 随机变量的产生方法：

- 产生(0,1)区间上均匀分布的随机变量 U
- 产生服从参数为 a，b 的 Pareto 分布的随机变量 $X=\frac{b}{\sqrt[a]{u}}$

结果验证

同线性同余法。

（二）程序代码分析

随机数产生

要产生不同参数的 Pareto 分布时，只需要改变调用 Pareto()函数的两个参数即可。

本次实验选用 1.a=2,b=2;2.a=2,b=5;3.a=5,b=2;三种参数进行仿真与验证。

```
#include <stdio.h>
#include <math.h>
#include "mt19937ar.h"
#include <fstream>
#include <random>
using namespace std;

#define pi 3.1415926
#define LENGTH 20000
#define uint unsigned int

void Pareto(double a, double b);

void main()
{
    Pareto(2.0, 10.0);
}

void Pareto(double a,double b) {
    ofstream ofile;
    ofile.open("Pareto.txt", ios::out);
    int i;
```

```
    unsigned long init[4] = { 0x123, 0x234, 0x345, 0x456 }, length =
4;//初始化 seed
    init_by_array(init, length);
    for (i = 0; i < LENGTH; i++) {
        ofile<<b/powf(genrand_real2(),1.0/a)<<endl;//根据计算出的公式产
生 Pareto 分布随机变量
    }
    ofile.close();
}
```

结果验证

同线性同余法。

（三）测试结果与分析

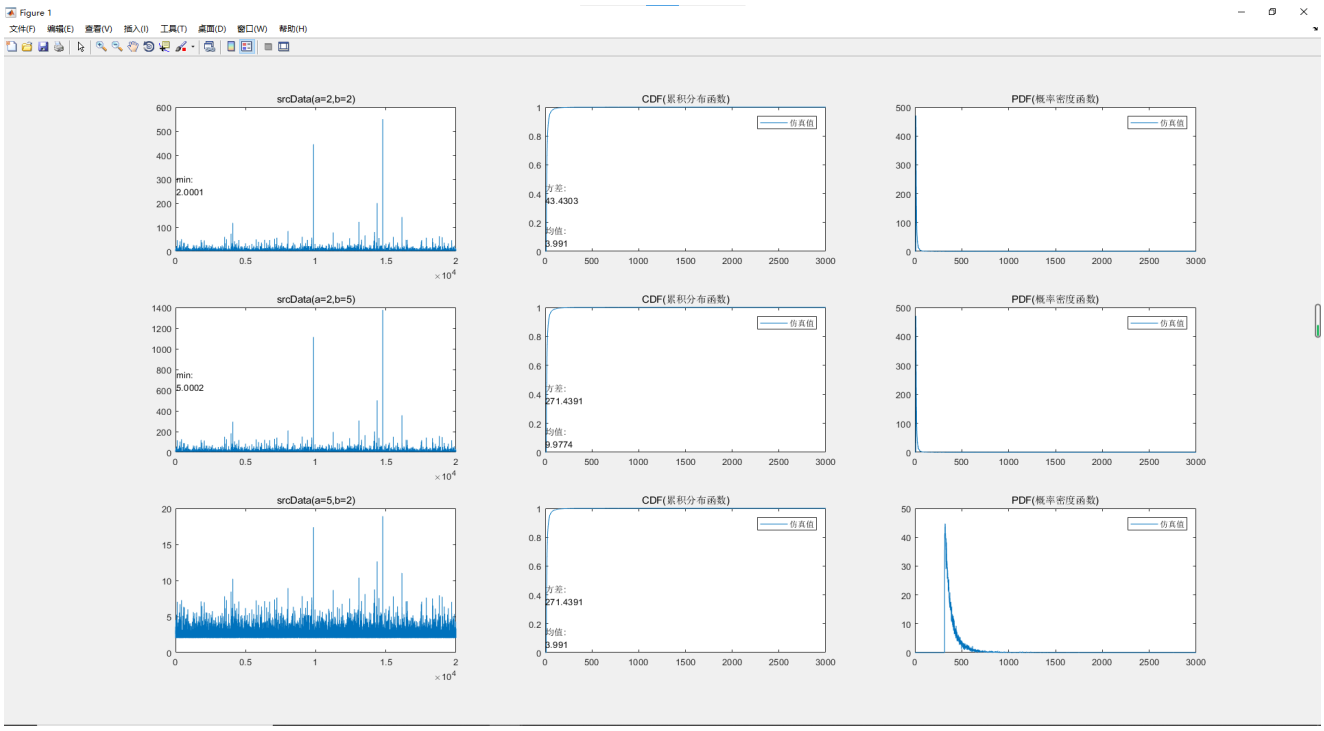

图 3.1

```
>> disp(["min(data0):",num2str(min(data0)),"min(data1):",num2str(min(data1)),'
   "min(data0):"    "2.0001"    "min(data1):"    "5.0002"    "min(data2):"    "2"
```

图 3.2

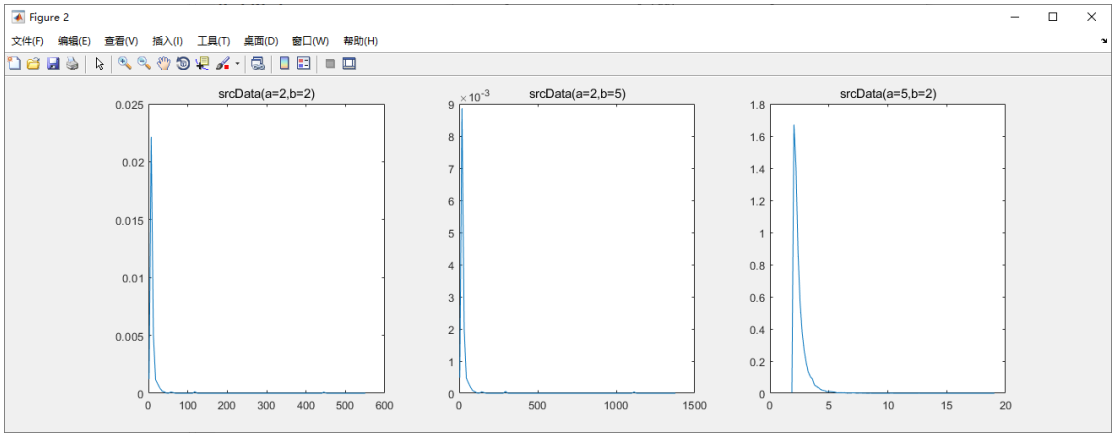

图 3.3

结果分析

Pareto 分布的 PDF 可知，Pareto 随机变量的最小值理论值是 b，经 matlab 验证，三种情况的随机数仿真值的最小值在误差范围内符合理论值 b。这支持了我们的算法的正确性。

观察 CDF 和 PDF 的图像，再跟理论值进行比较，可以看出图像符合预期。

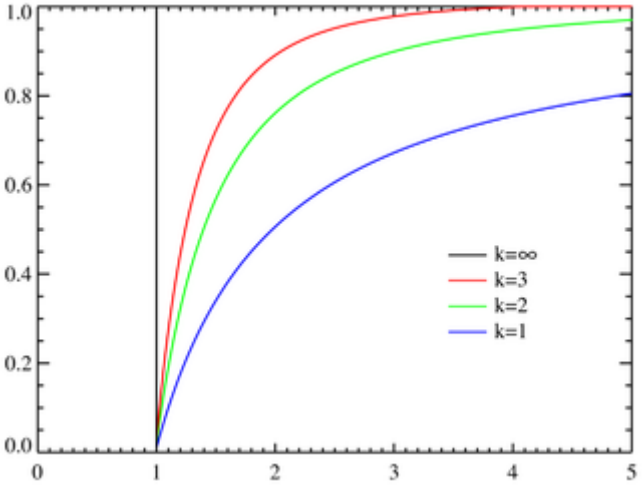

图 3.3 CDF

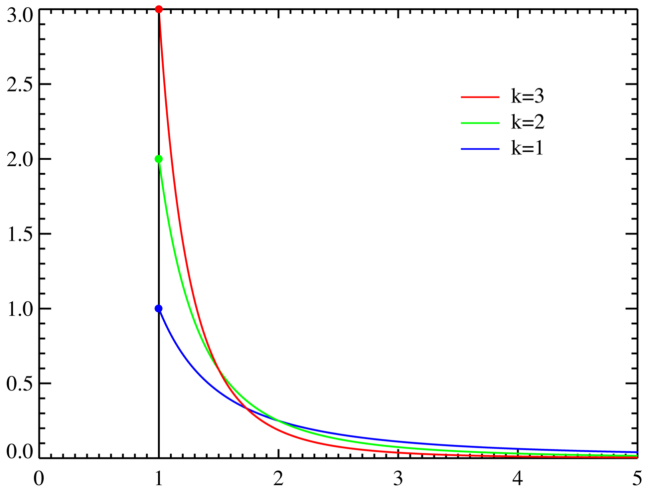

图 3.4 PDF

三种情况对比：

由 PDF 公式 $p(x)=\frac{ab^a}{x^{a+1}}$ $b\le x<\infty$ 以及反变换 $F^{-1}(u)=\frac{b}{\sqrt[a]{1-u}}=\frac{b}{\sqrt[a]{u}}$ 可以得知，当 a 增大时，由于根号的次数增加，因此输出的值会更接近 b，因此输出的结果会比 a 较小时更小，这种情况通过观察 srcData 的纵轴数据即可清楚地观察到；当 b 较大时，反变换的分子就会增大，因此输出的结果就会增大，同样可以通过观察纵轴数据观察到。

3.4 组合法产生混合高斯分布随机变量

（一）算法分析

随机数产生

当目标随机变量的累积分布函数可以表示为多个其它累积分布函数加权和的形式，而这些分布函数比目标变量分布函数更容易采样时，适合采用分布分解法，或称为组合法。

组合法产生随机变量的步骤如下：

•产生随机整数J，使$p\{J=j\}=p_j$—— 确定采用哪一个分布函数来取样，可 采用离散反变换法来实现

•产生具有分布函数$F_j(x)$的随机变量x_j—— 以该分布函数产生随机变量，可采用反变换法

- 令$x=x_j$

第一步可以用离散反变换法产生，确定所选择的分布函数

第二步则直接对所选择的分布函数进行采样，可以通过反变换法等方法实现

题目中所给的 PDF 为 $f(x)=\sum_{i=1}^{3}p_i\frac{1}{b_i}\phi(\frac{x-a_i}{b_i})$，经过化简，可以得出 $y_i=\frac{1}{\sqrt{2\pi}b_i}\exp(-\frac{(x-a_i)^2}{2b_i^2})$ 即为高斯分布的 PDF，因此题目所给的 PDF 可以拆分成三个高斯随机变量，$N(-1,1/4^2);N(0,1);N(1,1/2^2)$。

再利用反变换法产生每一个高斯随机变量，方法采用上题中的 Box-Muller 法。（标准高斯随机变量经过线性变换即可获得其他参数的高斯随机变量）

最后根据概率 p 选择不同的高斯分布进行，即可实现题目要求。

结果验证

同高斯随机变量

（二）程序代码分析

随机数产生

```
#include <stdio.h>
#include <math.h>
#include "mt19937ar.h"
#include <fstream>
#include <random>
using namespace std;

#define pi 3.1415926
#define LENGTH 20000
#define uint unsigned int

void combineGaussion();

void main()
{
    combineGaussion();
}

void combineGaussion()
{
   ofstream ofile;
```

```
    ofile.open("combineGaussion.txt", ios::out);
    int i;
    unsigned long init[4] = { 0x123, 0x234, 0x345, 0x456 }, length =
4;
    init_by_array(init, length);
    for (i = 0; i < LENGTH; i++) {
        double u1, u2, u11,u21,u12,u22,u3;
        u1 = genrand_real3();
        u2 = genrand_real3();
        u11= genrand_real3();
        u21 = genrand_real3();
        u12 = genrand_real3();
        u22 = genrand_real3();

        u3 = genrand_real3();
        if (u3 < 0.5)
        {
            ofile<<-1.0 + 1.0/4.0*sqrtf(-2.0f * logf(u1))* cosf(2.0f *
 pi * u2)<<endl;//chansheng0，1gaussion
        }
        else if (u3 < 5.0 / 6.0)
            ofile << sqrtf(-2.0f * logf(u1)) * cosf(2.0f * pi * u2) <<
 endl;
        else
            ofile << 1.0 + 1.0 / 2.0 * sqrtf(-2.0f * logf(u1)) * cosf
(2.0f * pi * u2) << endl;
        //ofile << 1. / 2. * (-1.0 + 1.0 / 4.0 * sqrtf(-2.0f * logf(u
1)) * cosf(2.0f * pi * u2)) + 1. / 3. * (sqrtf(-2.0f * logf(u11)) *
cosf(2.0f * pi * u21)) + 1. / 6. * (1.0 + 1.0 / 2.0 * sqrtf(-2.0f *
logf(u12)) * cosf(2.0f * pi * u22)) << endl;
    }
    ofile.close();
}
```

结果验证

同高斯随机变量

（三）测试结果与分析

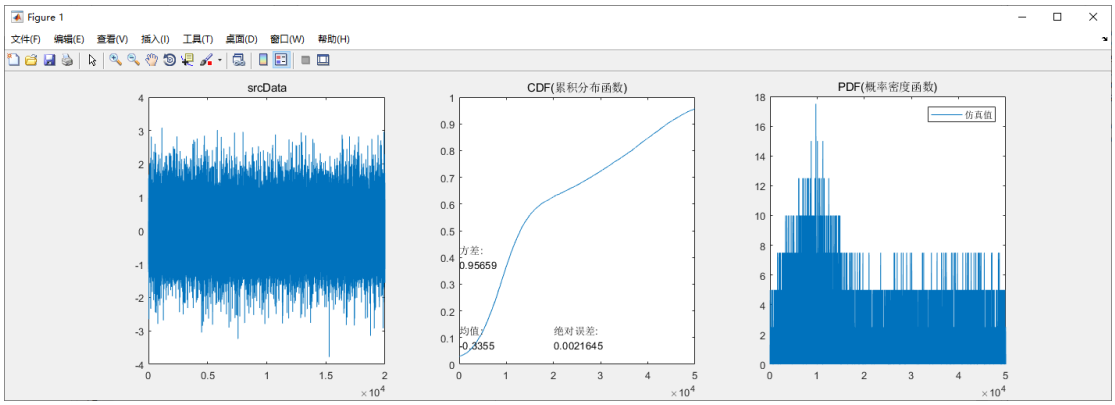

图 4.1

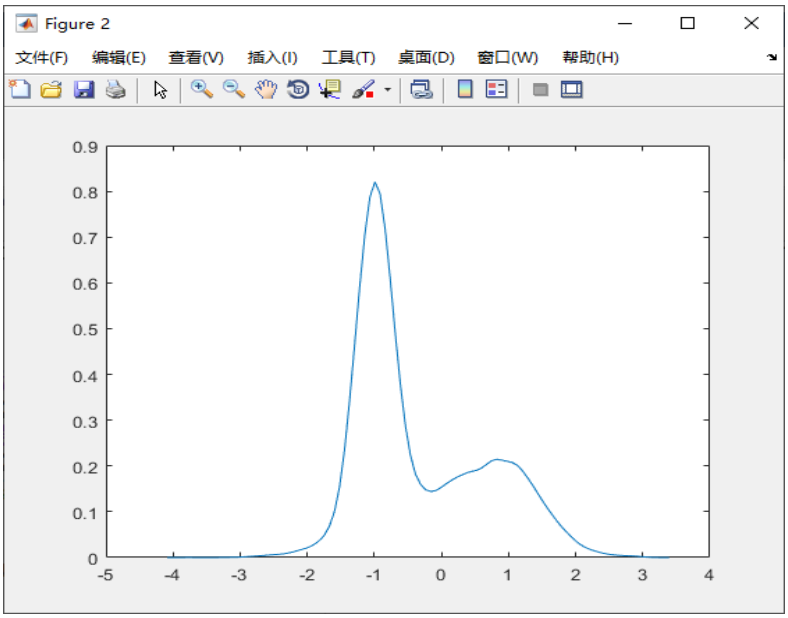

图 4.2

在图 4.1 中，由于$N(-1,1/4^2)$发生的概率最高，因此 PDF 图像主要集中在左侧（图像横轴映射为(-0.6233，+0.6233)）。在图 4.2 中，可以观察到明显的 PDF 曲线。

"均值:" "-0.3355"

"方差:" "0.95659"

图 4.3

输出的均值为-0.3355，其理论值为$-1*\frac{1}{2}+0+\frac{1}{6}*1=-\frac{1}{3}$，相对误差为 0.0065%；输出的方差为 0.95659。这同样可以验证我们的结果正确。

3.5 泊松过程的产生

（一）算法分析

随机数产生

泊松过程的两种解释：

- 某一时间内的事件发生次数服从泊松分布；
- 相邻两次事件发生的时间间隔服从指数分布；

利用第二种解释，可以产生泊松过程

- 产生均匀分布随机数
- 利用反变换法产生指数分布随机数
- 把指数分布随机数作为时间间隔，可 以得到一个时间序列

反变换为 $X=-\frac{1}{\lambda}\ln U$，即可获得指数分布的随机数，将其作为间隔产生泊松过程。

结果验证

绘制图像部分代码与 LCG 相同，但是，由于我们产生的随机数的间隔才是我们想要的泊松过程，因此在进行绘制之前要做差分。

（二）程序代码分析

随机数产生

```
#include <stdio.h>
#include <math.h>
#include "mt19937ar.h"
#include <fstream>
#include <random>
using namespace std;

#define pi 3.1415926
#define LENGTH 20000
#define uint unsigned int

void Poisson();

void main()
{
    Poisson();
}
```

```
void Poisson()
{
    ofstream ofile;
    ofile.open("Poisson.txt", ios::out);
    int i;
    double lambda = 2.0;
    double tpn_1 = 0;
    unsigned long init[4] = { 0x123, 0x234, 0x345, 0x456 }, length =
4;
    init_by_array(init, length);
    for (i = 0; i < LENGTH; i++) {
        double u1;
        u1 = genrand_real2();
        if (i == 0)
        {
            tpn_1 = -1.0 / lambda * log(u1);
            ofile << tpn_1 << endl;
        }
        else
        {
            tpn_1 = tpn_1-1.0 / lambda * log(u1);
            ofile << tpn_1 << endl;
        }
    }
    ofile.close();
}
```

结果验证

```
data = load("Poisson.txt");
data = gradient(data);
time = 1./data;
[y,x]=ksdensity(time);
plot(x,y);
%后续画图部分与 LCG 相同
```

（三）测试结果与分析

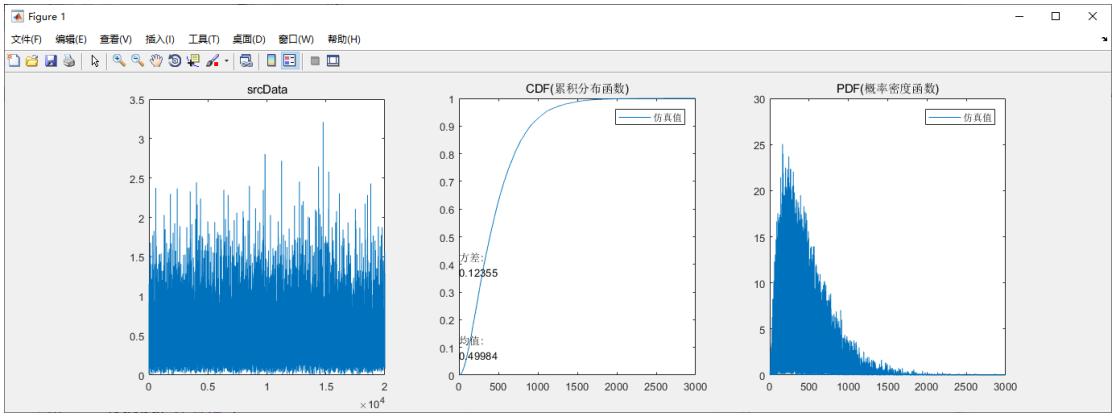

图 5.1

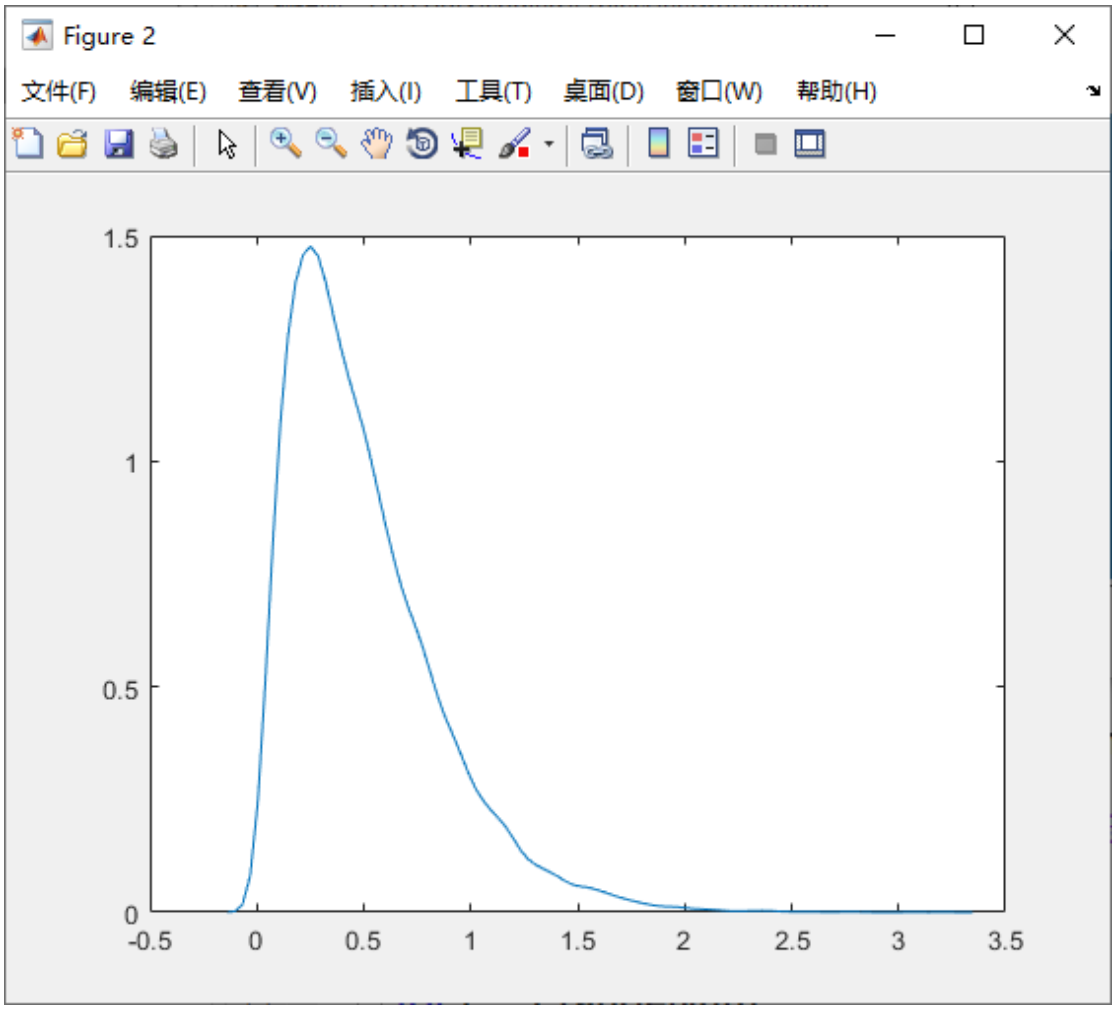

图 5.2

我们的理论结果应为数据满足泊松分布，与其 CDP、PDF 进行对比：

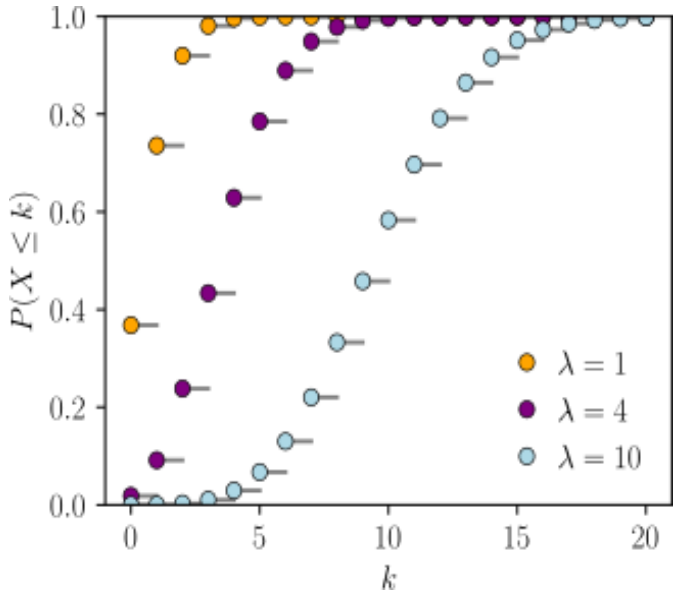

图 5.3 CDF

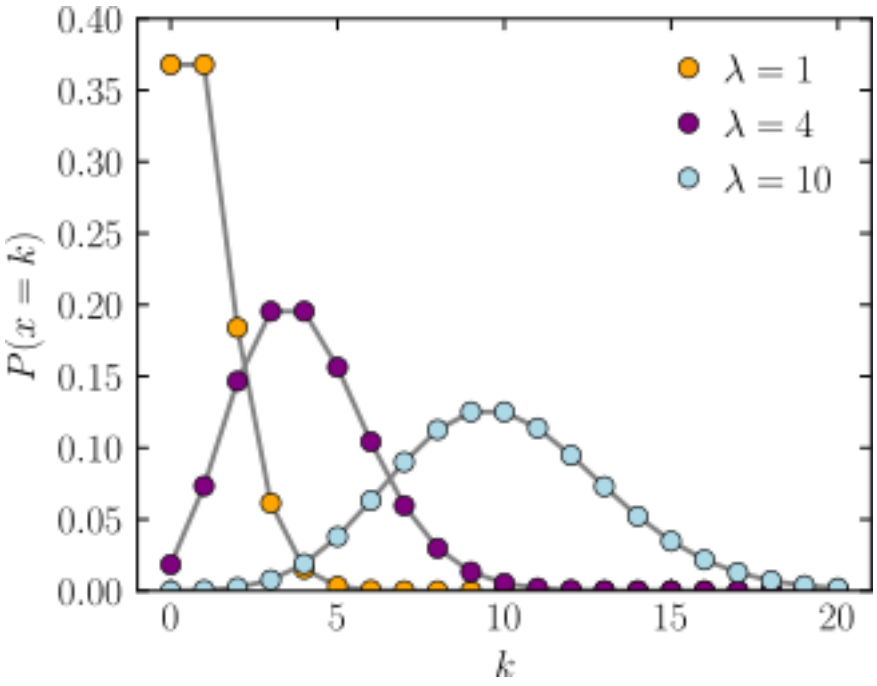

图 5.4 PDF

可以观察到在误差允许的范围内，CDF 和 PDF 图像与预期值相等，因此可以认定我们的算法成功产生了泊松随机过程。

4.实验结论与心得体会

结论：

- 本次实验中的三中产生均匀随机变量的方法中，梅森旋转法以及 MRG32k3a 算法的性能较好。

- 产生高斯随机变量时，中心极限定理以及 Box-Muller 法各有胜负。
- Pareto 分布可以通过反变换法产生。
- 利用组合法可以简化部分复杂随机变量的产生过程
- 泊松过程利用相邻两次事件发生的时间间隔服从指数分布的方法产生较为方便

体会：

这次实验充分体会了随机变量以及随机过程的产生原理以及实现方法，接触到了很多新的理论。在代码编写过程中，使用 cpp 编写随机数产生函数，matlab 分析随机数结果。matlab 代码编写时也了解了 CDF 和 PDF 的绘制方法。这次实验的一些遗憾是混合高斯随机变量的显示效果不是很理想，但目前还没有想出更好的解决方案。

总的来说，这次实验获益匪浅。