Randomness and Pseudorandom Numbers

1 March 2020

GROUP 13

Yigao Fang 518370910032

Haorong Lu 518370910194

Hongdan Wang 518370910209

Abstract

Random numbers are of great importance in applications. Figuring out some efficient and accurate methods to generate pseudo-random numbers, which are known as pseudo-random number generators, is what we need to consider. Based on statistics and mathematics principles, we introduce two commonly used algorithms that can deal with such problems, linear congruential generator(LCG) and Mersenne twister algorithm. We explore the influence of parameters in LCG equations and choose one of the best to do the simulation. For each method, we compute four sets of pseudo-random numbers with different sample sizes and draw the histograms and box plots of them. We also calculate the sample mean and variance and compare them with the theoretical to show the randomness of our results. Finally, we analyze the sensitivity of our simulation and give the conclusion.

Keywords: Pseudo-random number, Uniform distribution, LCG, Mersenne twister algorithm.

Contents

1 Introduction

1.1 Background

Considering a way to create truly random numbers is of great importance in many applications. However, it is very difficult and time-consuming to do so. Hence pseudo-random numbers are widely used in practice. The pseudo-random numbers are obtained from some typical mathematical equations so that they can have some random appearance. In most of the cases, these random numbers are based on some so-called *seed point*, such as the detailed time when the number was created. However, these numbers do not truly distribute randomly. The degree of their randomness depends on the mathematical function and the seed point we choose. In this paper, we will give two pseudo-random number generator algorithms(PNRG) that can generate pseudo-random numbers, the method of linear congruential generator and Mersenne twister algorithm. Then we will give some comments on the randomness and sensitivity of the numbers we obtained from the PNRG.

1.2 Pseudo-random Number Generator Algorithm(PNRG)

Linear Congruential Generator(LCG)

Linear congruential generator(LCG) is one of the earliest and most famous pseudo-random number generator algorithms. We first have the linear recursive formula as follows

$$X_{n+1} = (aX_n + c) \quad mod \quad m \tag{1}$$

where X_n is the pseudo-random number series we want to obtain, $m > 0$ is the modulus, a and c are constant parameters. We assume X_0 as the seed point which depends on the detailed time when the random number is generated.

Consequently, when we choose a suitable set of parameters m, a, and c, a series of pseudo-random numbers can be obtained. However, the randomness of these numbers is strongly related to the parameters we choose. Unbefitting parameters may result in terrible numbers which follow some obvious patterns so that they are not so random as we wish. Analyzing Equation (1), we can find out that the series X_n will follow a circulation with period T_0 concerning the value of a, c, and m. To obtain a pseudo-random number series, the period should be large enough so that the characteristic of circulation can be ignored. That is

$$T_0 >> n \tag{2}$$

The period T_0 will always less than or equals to m. Hence there are generally two ways to increase the period:

1. Make a full period(i.e: $T_0 = m$).
2. Make m large enough.

Zhang[1] gives a theorem to indicate the relationship between period T_0 and the parameters we choose. He points out that when the following conditions are satisfied, the period T_0 can be a full period which equals m:

1. $(c, m) = 1$

2. For any prime factor p of m, $a = 1(mod \quad p)$
3. If $4|m$, then $a = 1(mod \quad 4)$

Based on this theorem, the article provides a method to determine the best parameters for LCG. Table 1 gives some good parameters widely used in applications, and we will utilize the parameters of the C++ standard to generate pseudo-random numbers, which are

$$a = 214013$$
$$c = 2531011$$
$$m = 2^{32}$$

With these parameters, the series X_n will have a period of $2^{32} - 1$. This is an incredibly large number, hence the circulation property can be ignored under relative small sample size n. The pseudo-random numbers we obtained are integers ranging from 0 to $2^{32} - 1$. The next step is to compress the range to [0,99]. We divide the origin series X_n by 100 and find the reminder Y_n. Consequently, Y_n is a series of numbers range from 0 to 99 with a uniform distribution. This is exactly what we want to obtain.

The detailed results will be covered in the next section and our C++ code is listed in Appendix.

Source	**Modulus** m	**Multiplier** a	**Increment** c
Numerical Recipes	2^{32}	1664525	1013904223
Borland C/C++	2^{32}	22695477	1
Microsoft Visual/Quick C/C++	2^{32}	214013	2531011
cc65	2^{32}	16843009	826366247
random0	$2^3 7^5$	8121	28411

Table 1: Parameters of LCG Widely Used in Applications[2]

Mersenne Twister Algorithm

Mersenne Twister algorithm is an efficient and accurate pseudo-random number generator algorithm that was put forward in 1997. It is now widely used as a random number generator in programming languages and libraries. Wikipedia[3] gives an example of a pseudocode of Mersenne twister algorithm, and we apply Python to do the coding based on this. The main process of this algorithm is illustrated in Figure 1.1. For an incredibly large sample size, this algorithm works better than the method of LCG. The detailed results of this method will be covered in the next section and our Python code is listed in the Appendix.

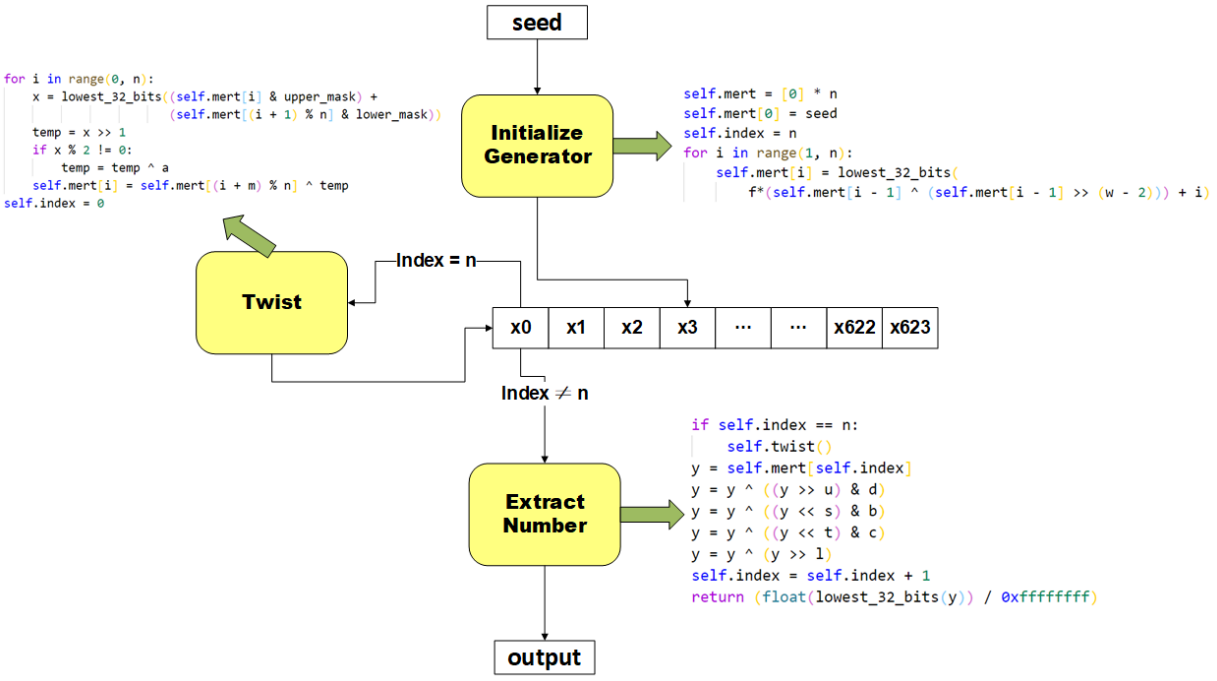


Figure 1.1: Basic flow sheet for Mersenne Twister Algorithm

2 Results

2.1 Theoretical Result Calculation

Theoretically, the expectation of a uniform distribution within a range of [a,b] should be

$$E[X] = \lim_{n\to+\infty} \sum_{i=1}^{n} x_i \times \frac{1}{n} \tag{3}$$

$$= \frac{a+b}{2} \tag{4}$$

while the variance of this uniform distributed random variable is given by

$$Var[X] = E[X^2] - E[X]^2 \tag{5}$$

$$= \frac{a^2+ab+b^2}{3} - \frac{(a+b)^2}{4} \tag{6}$$

$$= \frac{(a-b)^2}{12} \tag{7}$$

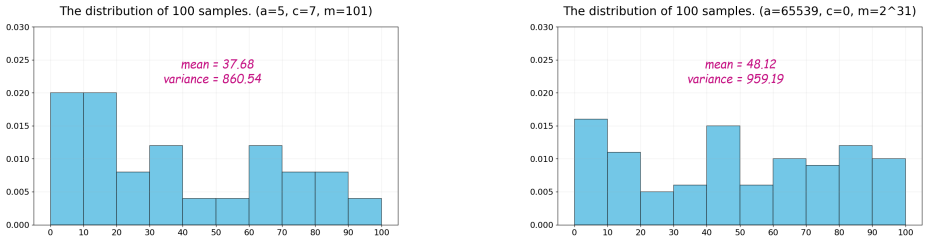
Insert $a = 0, b = 100$ into Equation (4) and Equation (7), we obtain the theoretical value of expectation and variance of uniform distribution numbers we previously generated using LCG and Mersenne Twister Algorithm, which are

$$E[X]_t = \frac{0+100}{2} = 50 \tag{8}$$

$$Var[X]_t = \frac{(0-100)^2}{12} = 833.3 \tag{9}$$

2.2 Result of Linear Congruential Generator

We first apply the method of linear congruence to generate uniform distribution within the range of [0,100]. As mentioned in Section 1.2, the value of parameters a, c, and m we choose will strongly influence the randomness of numbers we obtain. Figure 2.1 illustrates two examples of results that the LCG will generate under improper choices of parameters.



(a) a5, c=7, m=101 (b) a=65539, c=0, m=2^{31}

Figure 2.1: Examples of Some Improper Choice of Parameters

Figure 2.1 (a) indicates the histogram of the distribution of 100 samples with the parameters a=5, c=7, m=101. This histogram does not have an obvious characteristic of uniform distribution. The mean value $\overline{X}$ and variance S^2 also have a huge difference with the theoretical value. To make a deeper insight into this series of samples, we list the fifty twenty data of the 100 samples below:

42	15	82	13	72	64	24	26	36	86
33	71	59	0	2	17	92	63	19	1
12	67	39	0	7	42	15	82	13	72
64	24	26	36	86	33	71	59	0	2
17	92	63	19	1	12	67	39	0	7

According to these samples, we easily find out that this series follows a circulation with period equals 25. The first twenty-five samples are the same as the next twenty-five samples. These bad results may result from the small value of parameters we choose, as we mentioned before in Section 1.2. This is not what we want to obtain. Hence we consider a case with large parameters.

Figure 2.1 (b) indicates the 100 samples with the parameters a=65539, c=0, m=2^{31}. The selection of these three parameters is known as the RANDU algorithm of IBM[2] which is widely used in the 1960s. The distribution will become much more random when the three parameters become larger. Here we choose to have a huge value of parameters, and hence the histogram we obtain shows more randomness than Figure 2.1 (a) does. This one may seem much better, but still has a terrible appearance. Here we list the first twenty samples below:

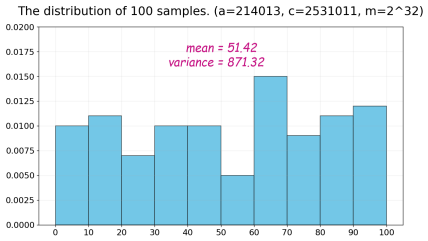
4	80	96	48	16	0	16	80	40	12
0	48	80	40	12	4	16	56	76	8

Obviously, this is a bad series, since all the numbers are even, without any odd numbers. Since $65539 =$

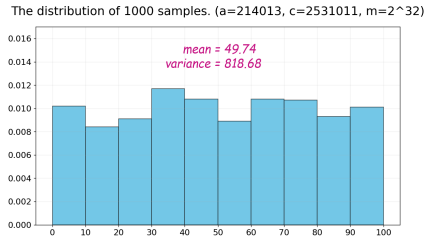
$2^{16} + 3$, we can further obtain that

$$\begin{aligned} X_{n+2} &= (2^{16} + 3)X_{n+1} \\ &= (2^{16} + 3)^2 X_n \\ &= [6(2^{16} + 3) - 9]X_n \\ &= 6X_{n+1} - 9X_n \end{aligned}$$

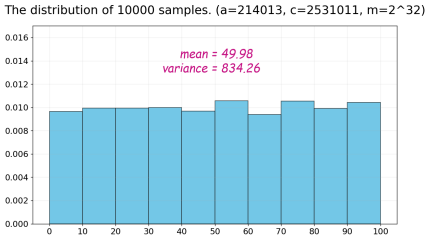
We should of course avoid such circumstances when generating pseudo-random numbers. Hence we need to find the most suitable parameters for Equation (1) so that the LCG can work properly and give the most random distributed series without any regular patterns. We apply the parameters we choose in Section 1.2. We insert these parameters into Equation (1) and choose different sample sizes with n=100, n=1000, n=10000, and n=100000 respectively. Since $n << T_0 = m$, this simulation will work well. The histograms are shown below in Figure 2.2



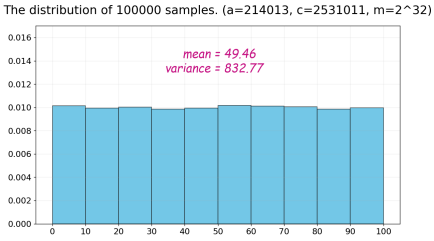
(a) The distribution of 100 samples



(b) The distribution of 1000 samples



(c) The distribution of 10000 samples



(d) The distribution of 100000 samples

Figure 2.2: Examples of Some Improper Choice of Parameters

According to Figure 2.2 (a), the distribution of 100 samples is not so uniform as we expected due to the limit of sample size. However, when the amount of sample becomes larger, the distribution becomes much more uniform, as illustrated in Figure 2.2 (b), (c), and (d). The mean value $\overline{X}$ and the variance S^2 of the sample are also calculated and illustrated in Figure 2.2.

The first twenty random numbers we generate in the 100000 samples distribution are listed below. We see that these numbers distribute randomly and do not have any regular patterns.

42	41	0	83	2	9	68	67	86	21
72	91	70	9	44	35	26	57	44	59

To gain a more direct perspective towards the characteristic of the distribution, we draw the box plot of 100000 samples, which is recorded below in Figure 2.3 As illustrated in Figure 2.3, the box plot has

an obvious symmetric characteristic, with its quartiles q_1 roughly equals 24, q_2 roughly equals 50, and q_3 roughly equals 74. Moreover, a_1 equals 0 and a_2 equals 99. All of the above are characteristics of a uniform distribution. Hence we claim that the pseudo-random numbers we generate using LCG follow a uniform distribution within [0,100].

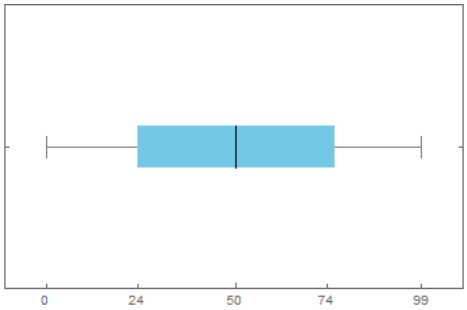
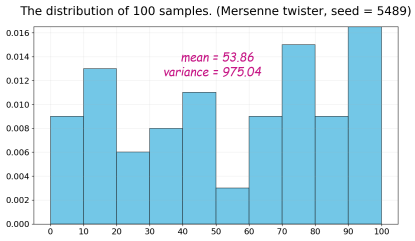


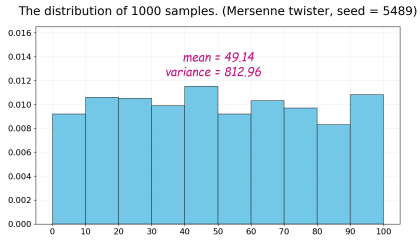
Figure 2.3: The box plot of the distribution of 100000 samples

2.3 Mersenne Twister Algorithm

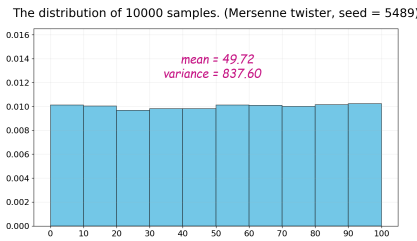
Except for LCG, another useful method to generate pseudo-random numbers is Mersenne twister algorithm. [3] gives the pseudocode of this algorithm. Based on the pseudocode introduced in [3], we apply Python to draw four histograms with sample size 100, 1000, 10000, and 10000 respectively, as illustrated in Figure 2.4 below.



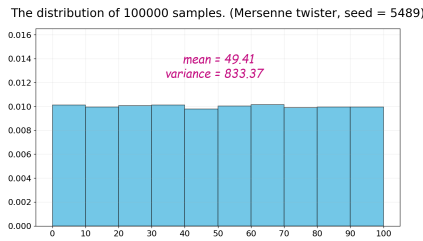
(a) The distribution of 100 samples



(b) The distribution of 1000 samples



(c) The distribution of 10000 samples



(d) The distribution of 100000 samples

Figure 2.4: Examples of Some Improper Choice of Parameters

Comparing with Figure 2.2 and Figure 2.4, we find that both methods can generate pseudo-random numbers with a large sample size. Mersenne twister algorithm is a much accurate method to deal with a large sample size of random numbers.

Our code is recorded in the Appendix and the detailed discussion of randomness and sensitivity will be covered in the next section.

3 Discussion and Conclusion

3.1 Randomness Analysis

According to the what we obtain in Result Section, we have

$$\overline{X}_{LCG,100000} = 49.46$$
$$Var[X]_{LCG,100000} = 832.77$$
$$\overline{X}_{MTA,100000} = 49.41$$
$$Var[X]_{MTA,100000} = 833.37$$

Comparing the mean and variance of the generated sample with size 100000 with the theoretical value as illustrated in Equation (7) and (8), we find out that their relative uncertainty is less than 1%. Hence we can conclude that random numbers we obtain using both these two methods have a uniform distribution characteristic. Furthermore, based on Figure 2.2, Figure 2.3, and Figure 2.4, we can also find out that their distribution follows high randomness, as we wish to have.

3.2 Sensitivity Analysis

Selection of Parameters

As illustrated in Section 2.2, the selection of LCG parameters will strongly affect the randomness of numbers we get. Small values will result in an obvious regular pattern of the series, and hence the pseudo-random numbers are not what we want to obtain. On the contrary, larger parameters will make the characteristic of circulation less obvious, and hence can obtain a better pseudorandom number.

Sample Size

Figure 2.2 and Figure 2.4 indicate that with a larger sample size, the numbers we get will distribute much more randomly. Compare the four histograms, one can find out that the distribution of 100 samples does not have a random and uniform characteristic. The numbers of samples vary a lot in each bin. Things get much better when we generate 1000 samples and analyze its property. The histogram looks nicer and the mean and variance are much closer to the theoretical value. When we choose 10000 or 100000 samples, the distribution looks extremely random and uniform. Hence we can claim that LCG as well as Mersenne twister algorithm can generate a large sample of pseudo-random numbers under high randomness.

3.3 Conclusion

In this project, we apply the method of LCG and Mersenne twister algorithm to generate pseudo-random numbers under the uniform distribution. We analyze the influence of parameters in the LCG equation and the sample size of the numbers. We find out that choosing an incredible large value of parameters a, c, and m will lead to a reasonable histogram graph. Also, the series mean and variance fit the theoretical value well.

Moreover, a larger sample size will lead to a better histogram for the uniform distribution. Hence our algorithms can deal with circumstances that need a large amount of uniformly distributed random numbers. The numbers we generated have high randomness which we desire.

References

[1] Zhang, G., & Zhang, X. (2007). The Analysis on the Period of Mixed Linear Congruential Generators. Journal of Shangqiu Normal University. 1672-3600 (2007) 06-0040-03, 41-42.

[2] Shine-lee. (2018). Algorithm of PNRG. [Web log post]. Retrieved from https://www.cnblogs.com/shine-lee/p/9516757.html

[3] Mersenne Twister. (2020). Retrieved April 1, 2020. Retrieved from: https://wiki2.org/en/Mersenne Twister

A LCG Code

[fragile]

```
#include<iostream>
#include<time.h>
using namespace std;

class PNRG
{
public:
    unsigned int seed, a, c; long long m;
    void SetSeed()
    //EFFECTS: set the initial value "seed" of PRNG, usually we use
        time as seed.
    //here time(NULL) represents number of seconds since 1970-1-1
    //MODIFIES: seed
    {
        seed = (unsigned int) time(NULL);
        return;
    }
    void SetParameters(unsigned int a_value, unsigned int c_value,
       long long m_value)
    //EFFECTS: set values of parameters a, c, m
    //MODIFIES: a, c, m
    {
        a = a_value; c = c_value; m = m_value;
        return;
    }
    unsigned int rand()
    //EFFECTS: generate random numbers using seed and parameters
    //MODIFIES: seed
    {
        seed = (a*seed + c) % m;
        return seed;
    }
};
int main()
{
    PNRG myPNRG;
    myPNRG.SetSeed();        //initialize seed
    myPNRG.SetParameters(214013,2531011,4294967296);
                            //set parameters(a, c, m)

    int n = 100000;          //number of random numbers
    unsigned int *array = new unsigned int[n]();
    unsigned int sum = 0;
```

```
        float sum_of_squares = 0;
        for (int i = 0; i < n; i++)
        {
            unsigned int a_PNRG = myPNRG.rand() % 100;
            array[i] = a_PNRG;
            cout << a_PNRG << ", ";
            sum += a_PNRG;
        }
        float mean = float(sum) / n;
        cout << "\nThe mean is " << mean << endl;
        for (int i = 0; i < n; i++) sum_of_squares += (array[i] - mean)
           *(array[i] - mean);
        float variance = sum_of_squares / n;
        cout << "The variance is " << variance << endl;
        delete[]array;
        return 0;
    }
```

B Mersenne Twister Algorithm Code

```
# -*- coding:utf-8 -*-
'''
This code implements the pseudo-random algorithm Mersenne Twister in python.
Date: 2020/4/2
Arthur: Haorong Lu
'''
import time
import numpy as np
from matplotlib import pyplot as plt


'''
Coefficients for MT19937, which is a Mersenne Twister
pseudo-random generator of 32-bit numbers
with a state size of 19937 bits.
Reference: http://www.cplusplus.com/reference/random/mt19937/
'''
w, n, m, r = 32, 624, 397, 31  # word size, state size, shift size, mask bits
a = 0x9908b0df  # XOR mask
u, s, t, l = 11, 7, 15, 18  # tempering u, s, t, l
d, b, c = 0xffffffff, 0x9d2c5680, 0xefc60000  # tempering d, b, c
f = 1812433253  # initialization multiplier
upper_mask = 0x80000000  # most significant w-r bits
lower_mask = 0x7fffffff  # least significant r bits


def lowest_32_bits(x):
    '''
    EFFECTS: Take the lowest 32 bits of a binary number.
    '''
```

```
    return int(0xffffffff & x)


class MT19937:
    def __init__(self, seed):
        '''
        EFFECTS: Initialize N states according to the given seed point x_0,
                 generate subsequent n-1 states x_1 to x_{n-1} through operations
                 such as shift, xor, multiplication, addition, etc.
        '''
        self.mert = [0] * n
        self.mert[0] = seed
        self.index = n
        for i in range(1, n):
            self.mert[i] = lowest_32_bits(
                f*(self.mert[i - 1] ^ (self.mert[i - 1] >> (w - 2))) + i)

    def extract_number(self):
        '''
        EFFECTS: Extract a tempered value based on mert[index]
                 and call twist() every n numbers.
        '''
        if self.index == n:
            self.twist()
        y = self.mert[self.index]
        y = y ^ ((y >> u) & d)
        y = y ^ ((y << s) & b)
        y = y ^ ((y << t) & c)
        y = y ^ (y >> l)
        self.index = self.index + 1
        return (float(lowest_32_bits(y)) / 0xffffffff)

    def twist(self):
        '''
        EFFECTS: Generate the next n values from the series x_i.
        '''
        for i in range(0, n):
            x = lowest_32_bits((self.mert[i] & upper_mask) +
                               (self.mert[(i + 1) % n] & lower_mask))
            temp = x >> 1
            if x % 2 != 0:
                temp = temp ^ a
            self.mert[i] = self.mert[(i + m) % n] ^ temp
        self.index = 0


def hist_plot(length, seed, data):
    '''
    EFFECTS: Plot the distribution of given random samples.
    '''
    mean = np.mean(data)
    var = np.var(data)
    plt.figure(figsize=(11, 6), dpi=128)
    title = 'The distribution of ' + \
        str(length) + ' samples. (Mersenne twister, seed = ' + str(seed) + ')'
```

```
    plt.hist(data, bins=range(0, 110, 10), color='#38B0DE',
             alpha=0.7, edgecolor='black', density=True)
    plt.grid(alpha=0.2)
    plt.title(title, fontsize=20, pad=20)
    plt.xticks(np.arange(0, 101, 10), fontsize=14)
    plt.yticks(fontsize=14)
    plt.ylim(0, 0.0165)
    plt.text(34, 0.0124, '     mean = %.2f \nvariance = %.2f' % (mean, var),
             family='fantasy', fontsize=20, style='italic', color='mediumvioletred')
    plt.savefig('mt' + str(length))


def main():
    '''
    The main function.
    '''
    seed = 5489  # 5489 is the default seed used on construction or seeding in C++.
    my_MT = MT19937(seed)
    result = []
    length = 100000
    for _ in range(length):
        result.append(int(my_MT.extract_number() * 100))
    print(result)
    hist_plot(length, seed, result)


if __name__ == '__main__':
    main()
```