Numerical Analysis Project: Random Number Generators

Team 8
109021129 陳冠丞
109021226 呂柏緯
110021118 賴杰弘
110021120 林元鴻
110021134 林暐晉
111198523 陳亦安

National Tsing Hua University

March 7, 2024

Contents

Numerical Analysis Project: Random-number generators

You

March 7, 2024

1 Introduction

In real world, if we want to get some **random results**, we could do things like **flipping the coin** or **tossing a dice** to get the corresponding outcomes(head or tail, 1 to 6), although these events are not actually random, they involve so many physical factors(e.g. gravity, air resistance, or humidity, etc...) that they are nearly impossible to predict or control. However, computers aren't designed to take advantage of these physics properties, that is, computers can **NOT** toss a coin or dice, we need to design some **algorithms** so that the results looks random.

To simulate randomness, programs typically use a **pseudo-random number generator**. A **pseudo-random number generator** (abbreviate it as **PRNG**) is an **algorithm** that generates a sequence of numbers whose properties simulate **a sequence of random numbers**.

What makes a good random number generator?

1. The period should be long enough, or even non-existent.
2. To generate a large number of random values in a short period.
3. There should be no distinct preference for certain values (the distribution should be sufficiently uniform) and should not be easily predictable.

2 Pseudo random number generator(PRNG)

Although the random number generator in computer is NOT "random" as we do in real world, the PNRG still play an important role on generating a sequence of

"random number". Hence the knowledge about PRNG is also important. We've just shown the definition of a Pseudo random number generator(PRNG). In this section, we are going to show the structure of a PRNG first, then introducing some PRNGs that we usually use.

Structure of PRNG

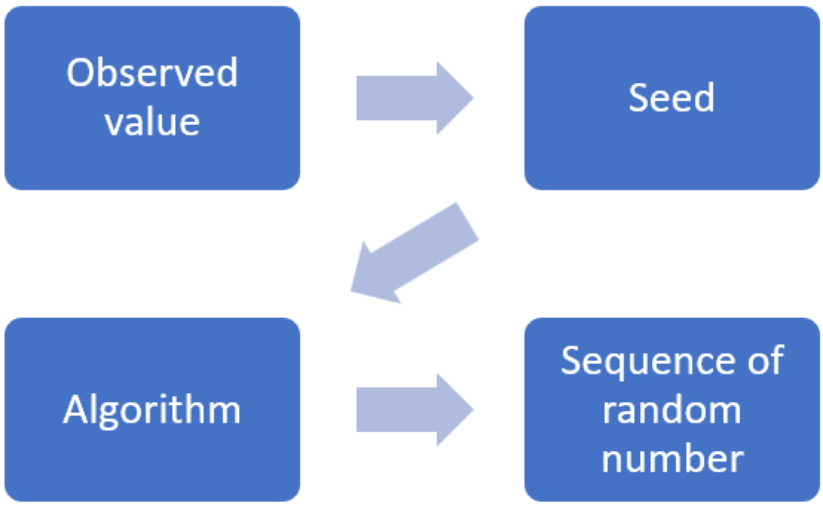


Figure 1: Structure of PRNG

How to get a "good" seed

The easiest way is pick current time to be seed In order to make seed more unpredictable, let seed

```
def f(time,size_of_domain):
    '''
    time: float
    size_of_domain: integer
    '''
    return int((time-int(time))*size_of_domain)
```

2.1 Linear congruential generator(LCG)

2.1.1 Algorithm of LCG

Linear congruential generator(LCG) is an algorithm that yields a sequence of pseudo-randomized numbers generating by following formula:

$$x_{n+1} = ax_n + c \ (mod \ m)$$

where $0 < m$ —the "modulus", $0 < a < m$ called the 'multiplier', $0 \leq c < m$ called the 'increment' and $0 \leq x_0 < m$ called the 'seed' or 'start value'.

2.1.2 Example of LCG

There are three common kinds of parameter choice:

1. m is prime, c = 0
2. m is power of 2, c = 0
3. $c \neq 0$, m and c are relatively prime, a-1 is divisible by all prime factors of m, a-1 is divisible by 4 if m is divisible by 4.[2]

The first and the third parameter selections have a period of m, the lenthsecond one have at most m/4.

Then let us take a quick example:

```
def LCG(seed=1):
    a = 5
    c = 1
    m = 2**3
    x = seed
    while (True):
        x = (a*x + c) % m
        yield x
```

We start from seed = 1. The result is as following:
1-> 6-> 7-> 4-> 5-> 2-> 3-> 0-> 1
Note that the period is 8(=m).

Let see a specific example:
when a = 6364136223846793005, c = 1, m = 2^{64} we have the following equation.

$$x_{n+1} = 6364136223846793005x_n + 1 \ (mod \ 2^{64})$$

the following python code is LCG with above parameters

```
def LCG(seed):
    a = 6364136223846793005
    c = 1
    m = 2**64
    x = seed
    while (True):
        x = (a*x + c) % m
        yield x
```

we use this function to generate 10^6 random numbers by seed 12345 and dividing by 2^{61} and turn them into a chart as figure 2

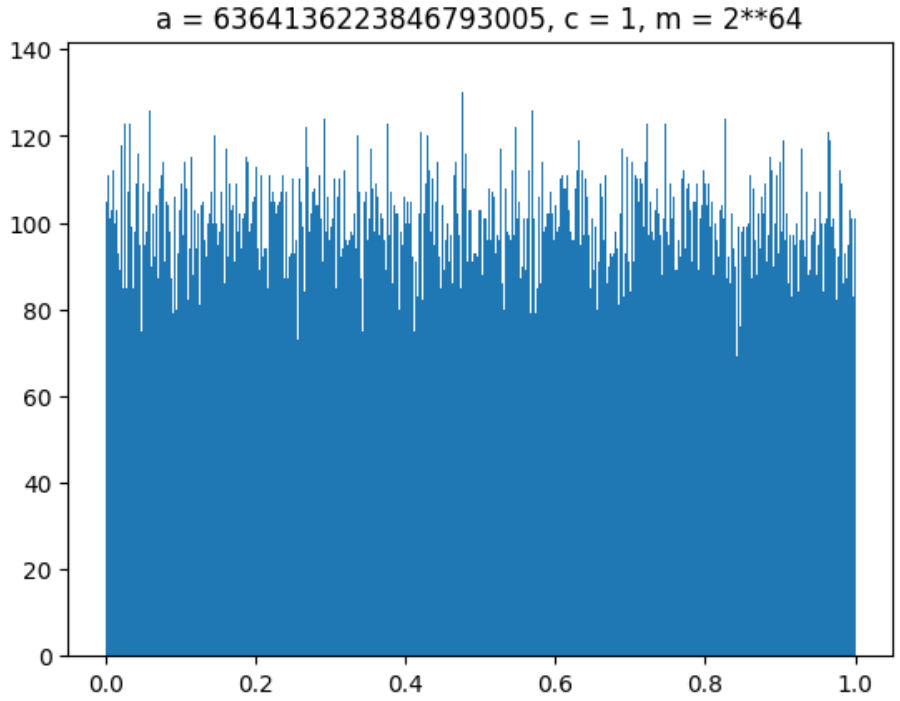


Figure 2: 10^5 random numbers generated by LCG

We can see that this distribution is very close to uniform[0,8], which is the same as our expectation

2.1.3 Advantages and Disadvantages of LCG

LCG is very fast and use less memory, but its disadvantages are also obvious, its period is short and is easy to crack when we got the "modulus" or its period(period is often the same as modulus)

2.1.4 Some observations of LCG

When we using the modulus being power of 2, then it will occur an obvious regular pattern that the two neighbor random numbers must be one even and one odd, so when we use power of 2 to be modulus, we have to filter used fragments of its binary type. for example we use the same parameter as above, and figure **??** is same numbers but adjusted by only choose the top 32 bits of its binary(i.e. $x'_k = \lfloor x_k/2^{32} \rfloor$), figure **??** is the same random numbers without adjust.

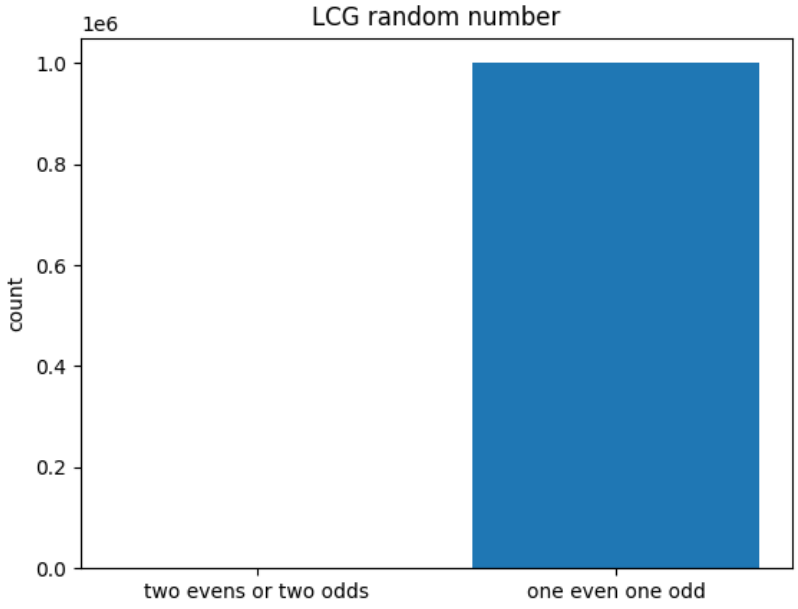


Figure 3: relation between two neighbor numbers with adjust

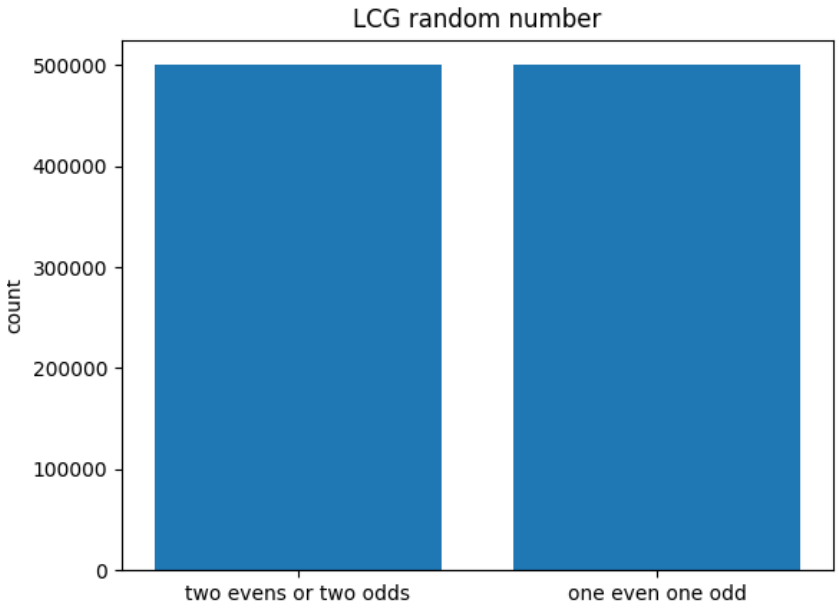


Figure 4: relation between two neighbor numbers without adjust

2.2 Linear Feedback Shift Register(LFSR)[1]

2.2.1 Observation

First, let us take a look at the finite field (Galois field) of order 2^n(denoted $\mathbb{F}_{2^n}, GF(2^n), or\ \mathbb{Z}/2^n\mathbb{Z}$) i.e. it is the set that only contains 2^n element satisfying the condition of field. Ex. $\mathbb{F}_{2^3}$ has the finite number of elements $\{0, 1, ..., 2^3-1\}$. We also can represent it as the form of polynomial: $a_0+a_1x+...+a_{n-1}x^{n-1}, a_i \in \{0, 1\}$. For example, take $\alpha = x^2 + 1\ in\ \mathbb{F}_{2^3}$. α can be viewed as binary: $1*2^2+0*2^1+1*2^0 = 5$. With such representation, we found that the addition and multiplication can be very different.

Example. Under the polynomial $x^3 + x + 1 = 0\ in\ \mathbb{F}_{2^3}$.
also denoted by $\mathbb{F}_{2^3} \cong \mathbb{F}_2[x]/ < x^3 + x + 1 >$ field extension
$x^3 = x + 1$, then
$(x^2 + x) + (x^2 + 1) = 2x^2 + x + 1 = x + 1\ (6 + 5 \equiv 3)$
$(x^2 + 1) * (x^2) = x * x^3 + x^2 = (x^2 + x) + x^2 = x\ (5 * 4 \equiv 2)$
The following tables show the operation acting on the element in $\mathbb{F}_{2^3}$ (table 1, 2)
We found that the strange relation which is different from the ordinary operating. This makes us thinking about whether we can get use of it. The answer is yes. By using a linear feedback shift register (LFSR) .

+	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	1	0	3	2	5	4	7	6
2	2	3	0	1	6	7	4	5
3	3	2	1	0	7	6	5	4
4	4	5	6	7	0	1	2	3
5	5	4	7	6	1	0	3	2
6	6	7	4	5	2	3	0	1
7	7	6	5	4	3	2	1	0

Table 1: addition

*	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7
2	0	2	4	6	3	1	7	5
3	0	3	6	5	7	4	1	2
4	0	4	3	7	6	2	5	1
5	0	5	1	4	2	7	3	6
6	0	6	7	1	5	3	2	4
7	0	7	5	2	1	6	4	3

Table 2: multiplication

Also, we should notice the primitive polynomial which is a important idea of finite field. We say that $F(x)$ is a primitive polynomial if it is the minimal polynomial of a primitive element of $\mathbb{F}_{p^n}$. That's, it is irreducible, monic and has a root α in $\mathbb{F}_{p^n}$. Get $\mathbb{F}_{p^n} = \{0, 1 = \alpha^0 = \alpha^{p^m-1}, \alpha, \alpha^2, ..., \alpha^{p^m-2}\}$. What's more, if $F(x)$ is a primitive polynomial, then x is always a primitive element of the field. ex.5

Note :

1. A polynomial is said to be irreducible if it cannot be factored into non-trivial polynomials over the same field.
2. A monic polynomial is a non-zero univariate polynomial in which the leading coefficient is equal to 1.
3. A primitive element of a finite field $\mathbb{F}_p$, is a generator of the multiplicative group of the field.
4. A primitive polynomial is the minimal polynomial of a primitive element of the finite field.
5. There's another way to find primitive polynomial more efficiently: An irreducible polynomial $F(x)$ of degree m over $\mathbb{F}_p$, where p is prime, is a primitive polynomial if the smallest positive integer n such that $F(x)$ divides x^n-1 is $n = p^m-1$.

2.2.2 LFSR

Some definition:

1. Boolean algebra uses true value(False=0,True=1) as variable, logical operators like AND, OR as operators.
2. flip flops are used to store a single bit of binary data (1 or 0).

Power	Polynomial	Vector	Integer
0	0	[0, 0, 0]	0
x^0	1	[0, 0, 1]	1
x^1	x	[0, 1, 0]	2
x^2	x^2	[1, 0, 0]	4
x^3	x + 1	[0, 1, 1]	3
x^4	x^2 + x	[1, 1, 0]	6
x^5	x^2 + x + 1	[1, 1, 1]	7
x^6	x^2 + 1	[1, 0, 1]	5

Figure 5: generating $\mathbb{F}_{2^3}$ by $x(\alpha)$

3. digital circuit can be viewed as composition of many functions. In our case, the circuit is many flip flops connected one by one.

In Boolean algebra, a **linear** function is a function $f : \{0,1\}^n \to \{0,1\}$ for which there exist $a_0, a_1, ..., a_n \in \{0,1\}$ such that $f(b_0, b_1, ..., b_n) = (a_0 \wedge b_0) \oplus (a_1 \wedge b_1) \oplus ... \oplus (a_n \wedge b_n)$, where $b_0, b_1, ..., b_n \in \{0,1\}, \oplus : XOR, \wedge : AND$. The usage of binary operation XOR, AND are natural since they are same as the operation on $\mathbb{F}_2$ (Table 1,2).Thus we use XOR to be the linear function in practice directly. Note that XOR is true if and only if the inputs differ (one is true, one is false).

A	B	A XOR B
1	1	0
1	0	1
0	1	1
0	0	0

Table 3: truth table of XOR

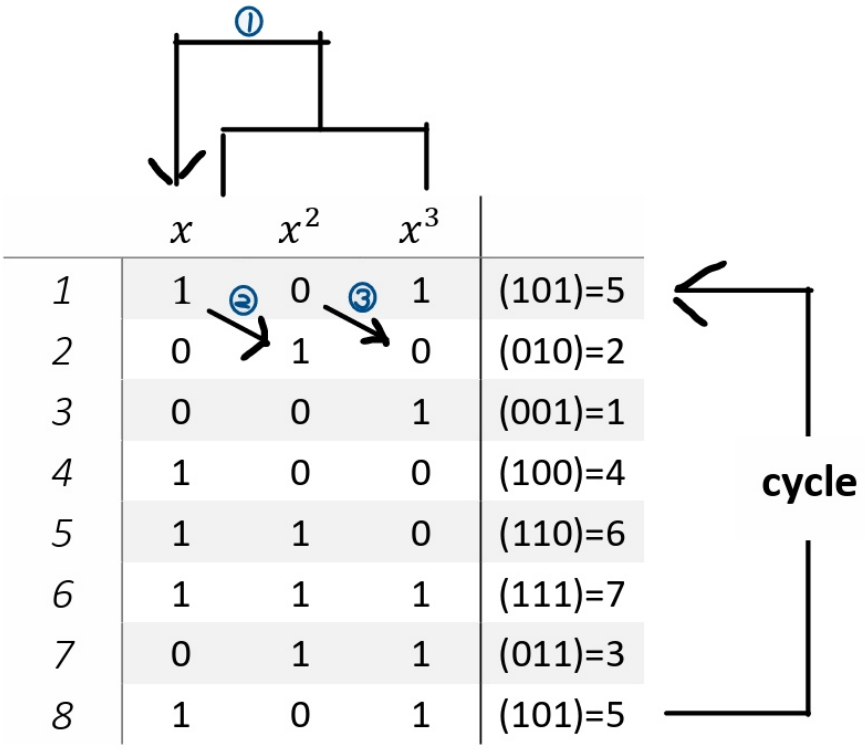
A	B	A AND B
1	1	1
1	0	0
0	1	0
0	0	0

Table 4: truth table of AND

A **shift register** is a type of digital circuit where the output of one flip-flop is connected to the input of the next. Combine above, a **linear feedback shift register (LFSR)** is a shift register whose input bit is the output of a linear function of two or more of its previous states (taps). with initial input(seed) $= (s_{n-1}, ..., s_1, s_0)$, the next contents are $(s_n, ..., s_2, s_1)$. All digits shift to the right and the leftmost bit s_{n-1} is replaced by $s_n = f(s_0, s_1, ..., s_{n-1}) = (a_0 \wedge s_0) \oplus (a_1 \wedge s_1) \oplus ... \oplus (a_{n-1} \wedge s_{n-1})$, f : linear function

$$, a_i = \begin{cases} 1 & if\ it's\ a\ tap \\ 0 & otherwise \end{cases} .$$

For example, take the taps of LFSR to be $[3, 1]$, we also denote that the feedback polynomial to be $x^3 + x + 1$ corresponding to the taps we chose. The procedure would be like figure6



	x	x^2	x^3	
1	1	0	1	(101)=5
2	0	1	0	(010)=2
3	0	0	1	(001)=1
4	1	0	0	(100)=4
5	1	1	0	(110)=6
6	1	1	1	(111)=7
7	0	1	1	(011)=3
8	1	0	1	(101)=5

Figure 6: $x^3 + x + 1$

Remark:

1. We can start from any number expect 0 to get the similar result with same length of period since 0 can only generates 0(=0+0+...+0).
2. It has the period of $7(= 2^3 - 1)$, which is the maximal-length can get from 3-digit LSFR(minus 0).
3. Although the polynomial is $x^3 + x + 1$, 1 is actually not used on LSFR.
4. Not all feedback polynomial can make the maximal-length period(ex. figure7). In fact, the LFSR is maximal-length iff the feedback polynomial is a primitive polynomial in $\mathbb{F}_2$. [3]

5. Intuitively, a shift is like to multiply x which is a primitive element as we introduced before. And the feedback is like to congruent($x^3 + x = 1$) . Make the process is like to walk through$\{0, 1, \alpha, \alpha^2, ..., \alpha^{p^m-2}\}$ but with different start and basis. Giving more sense to connect primitive polynomial with LFSR.

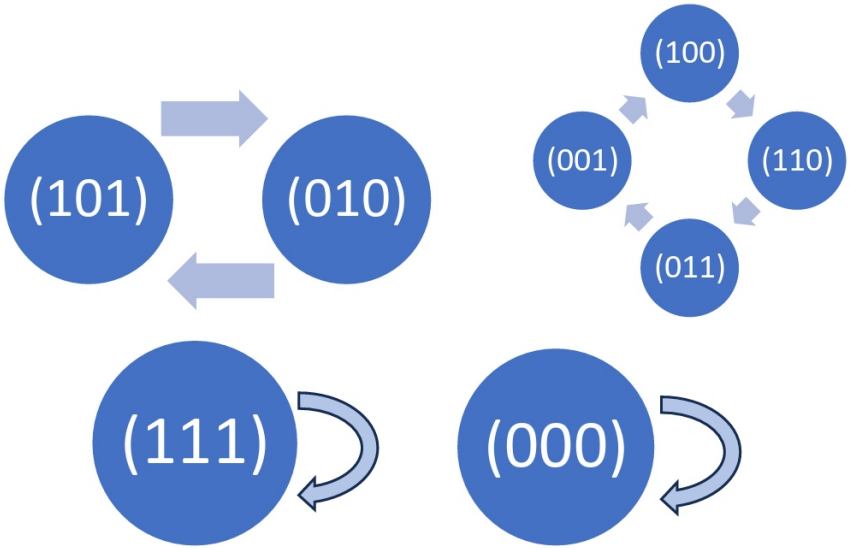


Figure 7: $x^3 + x^2 + x + 1$
maximal period=4

Remark 4 gives a reason why the polynomial should "+1" although it's useless. It's to build a relation between feedback polynomial and primitive polynomial.

2.2.3 practice

1. code for figure6

```
# 設定初始值
start_state = 1 << 2 | 1
lfsr = start_state
period = 0
while(True){
    # 將 taps 移至第一位做 XOR，再用 AND 將算好的 bit 取出
    bit = (lfsr ^ (lfsr >> 1) ) & 1
    # 右移一格 新 bit 放到最左邊
    lfsr = (lfsr >> 1) | (bit << 3)
```

```
    period += 1
    print(f"({bin(lfsr)}) {lfsr}")
    if (lfsr == start_state):
        print(f"period = {period}")
        break
}
```

2. for general case Using the python package "galois" to find the primitive polynomial of $\mathbb{F}_{2^n}$

```
# %pip install -U galois
import galois
def LS(seed=1,n=64,times=10):
    '''
        seed : (int) the start number.
        n : (int) the orde of primitive polynomial.  period = 2^n-1
        times : (int) print how many numbers
    '''
    # 設定初始值不超過 2^n
    seed %= 1 << n
    # 取得 primitive_polynomial 再轉換成 taps 的形式
    f = galois.primitive_poly(2, n, method="min")
    temp=f._coeffs
    poly=[]
    for i in range(1,len(temp)):
        if temp[i]==1:
            poly.append(n-i)

    print(f"0. {seed}")
    for i in range(1,1+times):
        bit=0
        for j in range(len(poly)):
            # 將 taps 移至第一位做 XOR，再用 AND 將算好的 bit 取出
            bit^= (seed>>poly[j])
        bit&=1
        # 右移一格 新 bit 放到最左邊
        seed = (seed >> 1) | (bit << (n-1))
        print(f"{i}. {seed}")

LS(2,3,8) # Example
```

2.3 $\mathbb{F}_p$-Linear Generator

2.3.1 Definition

$\mathbb{F}_p$-Linear Generator is the generator in finite field $\mathbb{F}_p$ (is the simply set {0,1,.....,p-1} together with addition and multiplication modulo p) with the

following form:

$$a_n = q_0 a_{n-r} + q_1 a_{n-r+1} + ... + q_{r-1} a_{n-1} (mod\ p)$$

where $q_0 ... q_{r-1}$ and initial conditions $a_0 ... a_{r-1}$ are integers in $0, 1, ..., p$

A multiple recursive generator is defined by the linear recurrence

$$a_n = q_0 a_{n-r} + q_1 a_{n-r+1} + ... + q_{r-1} a_{n-1} (mod\ p)$$
$$u_n = \frac{a_n}{p}$$

For example :

(1)The first example is from linear shift generator When p=2 ,$\mathbb{F}_2$ linear generator is simply a linear shift register, where A,B are matrices over $\mathbb{F}_p$ with A of size $r * r$ transform x_n to x_{n+1}, B of size $k * r$ transform vector x_n of length r to y_n of length k where k<r. The final step is transform y_n into a number in range $[0, p]$ by consider the entries of y_n as the coefficient of u_n.

(2)

$$Q_1(x) = x + 2, \quad p_1 = 3$$
$$a_i = 2 * a_{i-1}, \quad p_1 = 3$$

and let $a_0 = 1$

$$\{\frac{a_n}{p_1}\} = \{\frac{1}{3}, \frac{2}{3}, ...\} = \{0.\overline{01}, 0.\overline{10}, ...\}_2$$

Theorem

If p is a prime, and $q_0, ..., q_{r-1} \in \{0, 1, ..., p\}$ such that

$$a^r = q_0 + q_1 a_1 + ... + q_{r-1} a^{r-1} \ \ is\ \ primitive\ \ over\ \ \mathbb{F}_p$$

Then the $\mathbb{F}_p$ linear generator generates a sequence with period $p^r - 1$

MR232K3aa: Since it's part of MR232K3a so I called it MR232K3aa Let:

$$x_n = q_{11} x_{n-1} + q_{12} x_{n-2} + q_{13} x_{n-3} (mod m_1)$$
$$q_{11} = 527612,\ q_{12} = 0,\ q_{13} = -1370589$$
$$m_1 = 2^{32} - 22853$$

Since m_1 is prime, period is $2^{38} - 46$(By previous theorem).
It's the result:

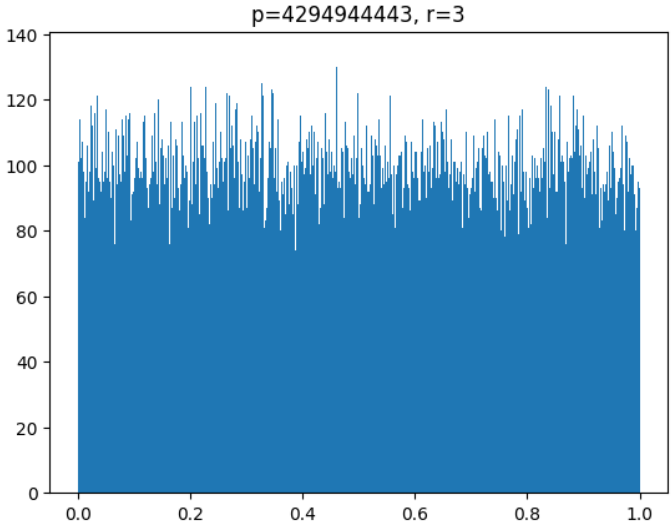


Figure 8: MRG32K3aa generate 100k numbers

2.4 Combined Multiple Recursive Generator(CMRG)

Restricting ourselves to $\mathbb{F}_p$-linear generators whose polynomial $Q(x)$ have exactly two nonzero coefficients, q_0 and q_s, with $0 < s \leq r-1$, greatly simplifies calculations.However, the generated sequences do not behave very well from a statistical point of view. In order to mitigate this deficiency we combine several such generators, operating with respect to distinct prime numbers p and distinct polynomials $Q(x)$ of the same degree.

2.4.1 Definition

We consider m linear recurrences

$$a_{n,j} = q_{0,j}a_{n-r,j} + q_{1,j}a_{n-r+1,j} + ... + q_{r-1,j}a_{n-1,j} \quad (mod\ p_j),\ j = 1, ..., m,$$

satisfying that p_j is prime and $q_{0,j}, ..., q_{r-1,j} \in \{0, 1, ...p-1\}$ are chosen such that the polynomial

$$Q_j(x) = x^r - q_{r-1,j}x^{(r-1)} - ... - q_{1,j}x - q_{0,j}$$

is primitive over $\mathbb{F}_p$.

Then we combine these recurrences as:

$$u_n = \left\{ \sum_{j=1}^{m} \frac{\delta_j a_{n,j}}{p_j} \right\}$$

where the δ_j are arbitrarily chosen integers s.t. each δ_j is relatively prime to p_j. And x represents the fractional part of a real number x defined by

$$\{x\} = x - [x]$$

where $[x]$ is the integer part of the number x.where $[x]$ is the integer part of the number x. (This means that we consider the $a_{n,j}$ both as elements of $\mathbb{F}_{p_j}$ and as real numbers!) A random-number generator of this form is called a combined multiple recursive generator.

Remark. In the literature we also find the notation x (mod 1) instead of x. Even if x and x are not integers, this definition is similar to the classic definition, where two numbers a and b are congruent modulo an integer n if their difference $a - b$ may be written in the form mn for an integer $m \in \mathbb{Z}$.

2.4.2 Example

Consider a CMRG with $r = 3, m = 2, p_1 = 3, p_2 = 2, and \delta_1 = \delta_2 = 1$. and $Q_1(x) = x^3 - x - 2$, $Q_2(x) = x^3 - x - 1$ Note that $Q_1(x)$ is primitive over $\mathbb{F}_3$ and $Q_2(x)$ is primitive over $\mathbb{F}_2$. That is, $q_{0,1} = -2, q_{1,1} = -1, q_{2,1} = 0$ and $q_{0,2} = -1, q_{1,2} = -1, q_{2,2} = 0$. In this example, m=2 means that there are only two generators we want to combine, and hence we could use simpler notation. That is,
1st linear recurrence equation is $a_n = 0 + a_{n-2} + 2a_{n-3}$
2nd linear recurrence equation is $b_n = 0 + b_{n-2} + b_{n-3}$
Combined these recurrences as $u_n = \frac{a_n}{3} + \frac{b_n}{2}$
Starting with initial condition 001 (i.e. $a_0 = b_0 = 0, a_1 = b_1 = 0, a_2 = b_2 = 0$)
Using the code CMRG3-4-2Ex.c, we get the result of above recurrences, 1st linear recurrence generator(LRG) give us the following sequence of period 26:

00101211201110020212210222

and the 2nd LRG give us the following sequence of period 7:

0010111

Hence, we could figure out that combined recurrence u_n gives us a sequence of period $26 \times 7 = 182$, and we could compare u_n with a_n and b_n , we would get the following table(only show the first 42 data):

0	0	1	0	1	2	1	1	2	0	1	1	1	0	0	2	0	2	1	2	2
0	0	1	0	1	1	1	0	0	1	0	1	1	1	0	0	1	0	1	1	1
0	0	$\frac{5}{6}$	0	$\frac{5}{6}$	$\frac{1}{6}$	$\frac{5}{6}$	$\frac{2}{6}$	$\frac{4}{6}$	$\frac{3}{6}$	$\frac{2}{6}$	$\frac{5}{6}$	$\frac{5}{6}$	$\frac{3}{6}$	0	$\frac{4}{6}$	$\frac{3}{6}$	$\frac{4}{6}$	$\frac{5}{6}$	$\frac{1}{6}$	$\frac{1}{6}$
1	0	2	2	2	0	0	1	0	1	2	1	1	2	0	1	1	1	0	0	2
0	0	1	0	1	1	1	0	0	1	0	1	1	1	0	0	1	0	1	1	1
$\frac{2}{6}$	0	$\frac{1}{6}$	$\frac{4}{6}$	$\frac{1}{6}$	$\frac{3}{6}$	$\frac{3}{6}$	$\frac{2}{6}$	0	$\frac{5}{6}$	$\frac{4}{6}$	$\frac{5}{6}$	$\frac{5}{6}$	$\frac{1}{6}$	0	$\frac{2}{6}$	$\frac{5}{6}$	$\frac{2}{6}$	$\frac{3}{6}$	$\frac{3}{6}$	$\frac{1}{6}$

Note that 1st row and 4th row are a_1 to a_{42}, 2nd and 5th row are b_1 to b_{42}, 3rd

row and 6th row are u_1 to u_{42}, and I use the straight dashed line to separate these data withe period of b_n, just to compare these data easily. With the first 42 data, we found that u_n or its subsequence is less regular than a_n and b_n.

2.4.3 MRG32k3a

MRG32k3a is one of good parameters and implementations for CMRG which is found by Pierre L'Ecuyer [4], it is a 32-bit combined multiple recursive generator with 2 components of order 3 and the following equations is its algorithm:

$a_n = 1403580 \cdot a_{n-2}$ - $810728 \cdot a_{n-3}$ ($mod\ 2^{32} - 209$)
$b_n = 527612 \cdot b_{n-1}$ -$1370589 \cdot b_{n-3}$ ($mod\ 2^{32} - 22853$)
$c_n = a_n$ - b_n
$d_n = c_n/(2^{32} - 209)$

Although MRG32k3a only combined 3 generators and each generator only have 2 component, it gives a wonderful random number output with period 2^{191} . The following is the implemented code of this random generator, and the coding idea is provide by Pierre L'Ecuyer[4]. It use norm to control the number generated between 0 and 1, and let it run 10 million times and draw the 10 million data generated as Figure 7.

```
import time
import matplotlib.pyplot as plt
# parameter from MRGK32a
norm = 2.328306549295728e-10
m1 = 4294967087.0
m2 = 4294944443.0
a12 = 1403580.0
a13n = 810728.0
a21 = 527612.0
a23n = 1370589.0
# Get current time to use as seed
y = int(time.time())
SEED = y
s10 = SEED
s11 = SEED
s12 = SEED
s20 = SEED
s21 = SEED
s22 = SEED
results = []
for i in range(10000000):
    # Component 1 (1st recursive equation)
    p1 = a12 * s11 - a13n * s10
    k = p1 // m1
```

```
        p1 -= k * m1
        if p1 < 0.0:
            p1 += m1
        s10 = s11
        s11 = s12
        s12 = p1

        # Component 2 (2nd recursive equation)
        p2 = a21 * s22 - a23n * s20
        k = p2 // m2
        p2 -= k * m2
        if p2 < 0.0:
            p2 += m2
        s20 = s21
        s21 = s22
        s22 = p2
        # Combination
        if p1 <= p2:
            print(f"{(p1 - p2 + m1) * norm} ", end='')
            results.append((p1 - p2 + m1) * norm)
        else:
            print(f"{(p1 - p2) * norm} ", end='')
            results.append((p1 - p2) * norm)
plt.hist(results, bins=500000, edgecolor='blue')
plt.title('Random Number Distribution')
plt.xlabel('Values')
plt.ylabel('Frequency')
plt.show()
```

2.5 The Mersenne Twister algorithm(MT)

Makoto Matsumoto and Takuji Nishimura (1997) [5] develope the MT algorithm provides a super astronomical period of $2^{19937} - 1$ and 623-dimensional equidistribution up to 32-bit accuracy.Mersenne Twister is used as a default PRNG by many software, such as standard C++ library (since C++11), Python, Dyalog APL, IDL, PHP, R, Ruby ... etc., and the Mersenne Twister is stated to be "more reliable". The MT algorithm is based on two following steps:Twist and Tempering

2.5.1 Twist

Twist is defined as:

$$\boldsymbol{X}_{k+n} = \boldsymbol{X}_{k+m} \oplus (\boldsymbol{X}_k^{upper} | \boldsymbol{X}_{k+1}^{lower}) A, \ (k = 0, 1, ...)$$

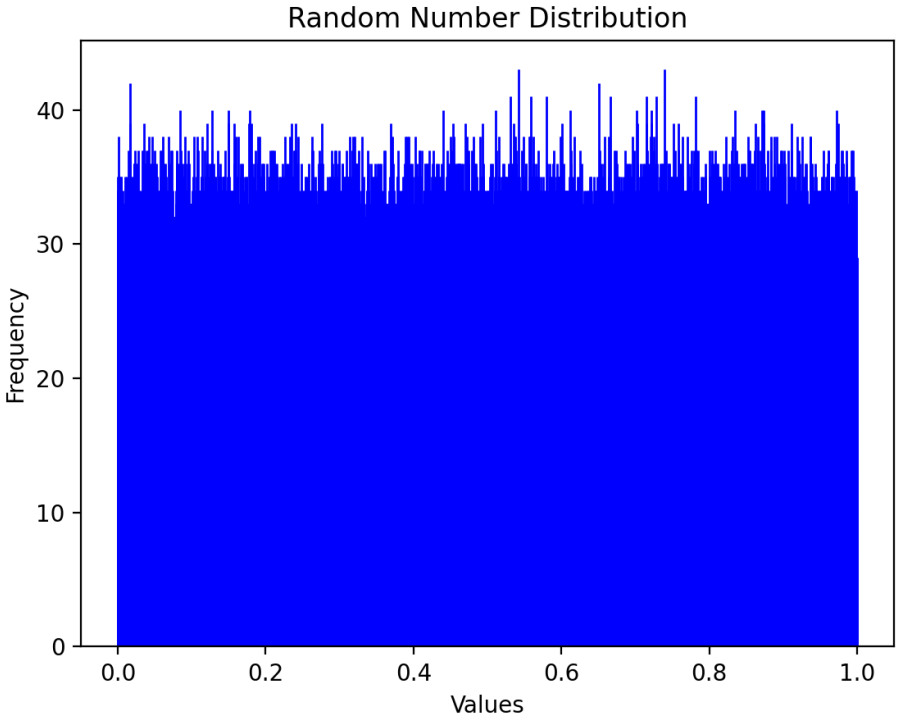


Figure 9: 10M datas of MRG32k3a

where n is the degree of the recurrence, $1 \leq m \leq n$, A is a constant $w \times w$ matrix with entries in $\mathbb{F}_2$, an interger r (hidden in the definition of $\boldsymbol{X}_k^u$ and $\boldsymbol{X}_{k+1}^l$), $0 \leq r \leq w-1$, the upper $w-r$ bits $\boldsymbol{X}^{upper} = (x_{w-1}, ..., x_r, 0, ..., 0)$, the lower r bits $\boldsymbol{X}^{lower} = (0, ..., 0, x_{r-1}, ..., x_0)$; namely,$(\boldsymbol{X}_k^{upper} | \boldsymbol{X}_{k+1}^{lower})$is just concatenating the upper $w-r$ bits of $\boldsymbol{X}_k$ and the lower r bits of $\boldsymbol{X}_{k+1}$, '$\oplus$' is bitwise XOR operation (bitwise addition modulo 2). The parameters n and r are selected so that the characteristic polynomial is primitive or $nw-r = 19937$ which is a Mersenne exponent.

We define the form of the matrix A:

$$A = \begin{pmatrix} 0 & 1 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 1 & & 0 & 0 \\ \vdots & \vdots & & \ddots & & \vdots \\ 0 & 0 & 0 & & 1 & 0 \\ 0 & 0 & 0 & \cdots & 0 & 1 \\ a_{w-1} & a_{w-2} & a_{w-3} & \cdots & a_1 & a_0 \end{pmatrix} = \begin{pmatrix} 0 & \boldsymbol{I}_{w-1} \\ a_{w-1} & (a_{w-2}, \cdots, a_0) \end{pmatrix}$$

then

$$\boldsymbol{X}A = \begin{cases} \boldsymbol{X} >> 1 & \oplus & 0 & \text{if} & x_0 = 0 \\ \boldsymbol{X} >> 1 & \oplus & \boldsymbol{a} & \text{if} & x_0 = 1 \end{cases}$$

where $<<$ and $>>$ are the bitwise left and right shifts. And the value of the $\boldsymbol{a}$ is chosen randomly

We note that if $r = 0$,then this recurrence reduces to the Twisted Generalized Feedback Shift Register (TGFSR) proposed in Matsumoto and Kurita [1992; 1994] [6][7] ,and if $r = 0$ and $A = I$, it reduces to Generalized Feedback Shift Register (GFSR) [Lewis and Payne 1973][8].

2.5.2 Tempering

As with A, we choose a tempering transform to be easily computable. The tempering is defined in the case as:

$$\begin{aligned} \boldsymbol{Y} &:= \boldsymbol{X} \oplus ((\boldsymbol{X} >> u) \& \text{ maxbits}) && (1) \\ \boldsymbol{Y} &:= \boldsymbol{Y} \oplus ((\boldsymbol{Y} << s) \ \& \ \boldsymbol{b}) && (2) \\ \boldsymbol{Y} &:= \boldsymbol{Y} \oplus ((\boldsymbol{Y} << t) \ \& \ \boldsymbol{c}) && (3) \\ \boldsymbol{Z} &:= \boldsymbol{Y} \oplus (\boldsymbol{Y} >> l) && (4) \end{aligned}$$

where u, s, t, and l called Tempering shift parameters are integers, maxbits is $2^w - 1$, $\boldsymbol{b}$ and $\boldsymbol{c}$ called Tempering bitmask parameters are suitable bitmasks of word size, '&' is bitwise AND operation. The tempering parameters must be chosen to satisfy the k-distribution test.

2.5.3 Initialization

If the initial state has too many zeros then the generated sequence may also contain many zeros for more than 10000 generations. So, The state needed for

a MT implementation is an array of n values of w bits each. To initialize the array, a w-bit seed value is used to supply $\boldsymbol{X}_0$ through $\boldsymbol{X}_{n-1}$ by setting $\boldsymbol{X}_0$ to the seed value and thereafter setting:

$$\boldsymbol{X}_k = f \times (\boldsymbol{X}_{k-1} \oplus (\boldsymbol{X}_{k-1} >> (w-2)) + k, (k = 0, 1, ..., n-1)$$

The constant f forms another parameter to the generator, though not part of the algorithm proper.

2.5.4 Coefficients in general

In C++ library[9],the coefficients for std::mt19937 are:

$$\begin{aligned} (w, n, m, r) &= (32, 624, 397, 31) \\ \boldsymbol{a} &= 9908\text{B}0\text{DF}_{16} \\ (u, s, t, l) &= (11, 7, 15, 18) \\ (\boldsymbol{b}, \boldsymbol{c}) &= (9\text{D}2\text{C}5680_{16}, \text{EFC}60000_{16}\ ) \\ f &= 6\text{C}078965_{16} \end{aligned}$$

and the coefficients for std::mt19937_64 are:

$$\begin{aligned} (w, n, m, r) &= (64, 312, 156, 31) \\ \boldsymbol{a} &= \text{B}5026\text{F}5\text{AA}96619\text{E}9_{16} \\ (u, s, t, l) &= (29, 17, 37, 43) \\ (\boldsymbol{b}, \boldsymbol{c}) &= (71\text{D}67\text{FFFEDA}60000_{16}, \text{FFF}7\text{EEE}000000000_{16}\ ) \\ f &= 5851\text{F}42\text{D}4\text{C}957\text{F}2\text{D}_{16} \end{aligned}$$

```
#MersenneTwister
class MersenneTwister:
    #Initialize the generator from a seed
    def __init__(self, seed = 0,w= 32,n= 624,m= 397,r = 31,\
                 a= 0x9908B0DF,\
                 u = 11,s = 7,t = 15,l = 18,\
                 b = 0x9D2C5680,c= 0xEFC60000,\
                 initialization_multiplier= 0x6C078965):
        #Word size
        self.__w = w
        #State size
        self.__n = n
        #Shift size
        self.__m = m
        #Mask bits
        self.__r = r
```

```
        #Xor Mask
        self.__a = a
        #Tempering shift parameters
        self.__u = u
        self.__s = s
        self.__t = t
        self.__l = l
        #Tempering bitmask parameters
        self.__b = b
        self.__c = c
        self.__max_bits = (1 << self.__w) - 1

        #f
        self.__initialization_multiplier = initialization_multiplier

        self.__lower_mask = (1 << self.__r) - 1
        self.__upper_mask = ( self.__max_bits | (~self.__lower_mask))

        self.__index = self.__n
        self.__MT = [0] * self.__n
        self.__MT[0] = seed & self.__max_bits

        #Initialization
        for i in range(1, self.__n):
            self.__MT[i]= (self.__initialization_multiplier * \
                           (self.__MT[i - 1] ^ (self.__MT[i - 1] >> (self.__w-2))) + i) & \
                            self.__max_bits
        return

    #Generate the next n values from the series X_i
    def __twister(self):
        for i in range(self.__n):
            #Get (X_i (upper)|X_i+1 (lower))
            #i.e. concatenating the upper w - r bits of X_i and the lower r bits of X_i+1
            x = (self.__MT[i] & self.__upper_mask) | \
                (self.__MT[(i + 1) % self.__n] & self.__lower_mask)
            #Compute XA
            xA = (x >> 1)
            if (x & 1):
                xA ^= self.__a
            #Compute X_k+n (= X_k+m XOR XA)
            self.__MT[i] = self.__MT[(i + self.__m) % self.__n] ^ xA
        self.__index = 0
        return

    # Extract a tempered value based on MT[index]
```

```
    # Calling twist() every n numbers
    def __temper(self):
        if self.__index == self.__n:
            self.__twister()

        x = self.__MT[self.__index]
        y = x ^ (x >> self.__u) & self.__max_bits
        y = y ^ (y << self.__s) & self.__b
        y = y ^ (y << self.__t) & self.__c
        z = y ^ (y >> self.__l)

        self.__index = (self.__index + 1) % self.__n

        #return lowest w bits of z
        return z & ((self.__max_bits<<1)|1)

    #call function
    def __call__(self):
        return self.__temper()
```

3 Generate a given distribution

The algorithms above help us to generate a sequence of random numbers which may have equal probability. Now we want to generate a sequence of numbers that follow a specific distribution. To achieve this goal, we need some basic probability knowledge. Throughout this section, for convenience, we just discuss the cases of the random variables with continuous type.

3.1 Basic Probability Knowledge

This subsection will help us recall the probability knowledge we need in this section.

Cumulative Distribution Function

Given a random variable X. The function defined by

$$F(x) = Pr(X \leq x) \ , \qquad -\infty < x < \infty \ ,$$

is called the **cumulative distribution function** of X and abbreviate it as **cdf**.

Uniform Distribution

A **uniform distribution** is the random variable X which has the pdf with the

form

$$f(x) = \frac{1}{b-a}\ , \quad a \le x \le b\ , \qquad \textit{where } [a,b] \textit{ is the support of } X$$

And we usually denote it by $X \sim U(a,b)$

Remark. The cdf of $U(a,b)$ is

$$F(x) = \begin{cases} 0, & x < a \\ \dfrac{x-a}{b-a}\ , & a \le x < b \\ 1, & b \le x \end{cases}$$

Exponential Distribution

A **exponential distribution** is the random variable X which has the pdf with the form

$$f(x) = \lambda \cdot e^{-\lambda x}, \quad 0 \le x < \infty\ ,$$

where $\lambda = \frac{1}{\theta}$ and θ is the parameter that $\theta > 0$, and we usually denote it by $X \sim exp(\theta)$

Remark. The cdf of $exp(\theta)$ is

$$F(x) = \begin{cases} 0, & -\infty < x < 0 \\ 1 - e^{\frac{x}{\theta}}, & 0 \le x < \infty \end{cases}$$

Normal Distribution

A **normal distribution** is the random variable X which has the pdf with the form

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \cdot e^{\frac{-(x-\mu)^2}{2\sigma^2}}\ , \qquad -\infty < x < \infty\ ,$$

where μ and σ are parameters satisfying $-\infty < \mu < \infty$ and $0 < \sigma < \infty$, and we usually denote it by $X \sim N(\mu, \sigma^2)$

Remark. The **standard normal distribution** is a normal distribution with $\mu = 0$ and $\sigma = 1$. i.e. $N(0,1)$. Therefore the pdf of a standard normal

distribution is

$$f(x) = \frac{1}{\sqrt{2\pi}} \cdot e^{\frac{-x^2}{2}} \ , \qquad -\infty < x < \infty \ ,$$

Change-of-Variables Technique

Let X_1, X_2 be two random variables with joint pdf $f(x_1, x_2)$, and let Y_1, Y_2 be some function of X_1, X_2 i.e. let

$$Y_1 = u_1(X_1, X_2) \ , \ Y_2 = u_2(X_1, X_2)$$

with the single-valued inverse given by

$$X_1 = v_1(Y_1, Y_2) \ , \ X_2 = v_2(Y_1, Y_2)$$

Then the joint pdf of Y_1, Y_2, denoted by $g(y_1, y_2)$, is

$$g(y_1, y_2) = f(v_1(y_1, y_2), v_2(y_1, y_2)) \cdot |J| \ , \ where \ J = \begin{vmatrix} \frac{\partial v_1}{\partial y_1} & \frac{\partial v_2}{\partial y_1} \\ \frac{\partial v_1}{\partial y_2} & \frac{\partial v_2}{\partial y_2} \end{vmatrix}$$

Above all, we recall the **cdf**, some **distributions**, and a **technique** we would use later.

3.2 Inverse Transform Sampling

We've just covered the probability knowledge we need in this section. Now we are going to introduce what is **Inverse Transform Sampling**. Note that, in this section, we abbreviate it as **ITS**. At the end, there would be an example that show how us apply this method practically.

Through previous subsection, we recall what is a cdf of a random variable X. Note that cdf is a continuous function with range $[0, 1]$. The thought of **ITS** is to find the inverse cdf function of a given distribution . So that we can map $[0, 1]$ into the support of X. Note that, in fact, we've done the mission that draw a real number randomly from an interval. Together with **ITS**, we can map these random numbers into a support of X. Hence those numbers would be with a specific probability that fit in the given distribution.

Inverse Transform Sampling

Given a random variable X, let $f(x)$ and $F(x)$ be the pdf and cdf of X respectively. the **algorithm** for **ITS** is

1. Generate $U(0,1)$

2. Calculate the inverse function $x = F^{-1}(y)$

3. Send $[0,1]$ into $F^{-1}(y)$

Proof of Inverse Transform Sampling

Let U be a continuous random variable having a standard uniform distribution i.e. $U \sim U(0,1)$. Let X be a random variable such that follows the given distribution with cdf $F_X^{-1}(x)$. Define the random variable Y by

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

Then we calculate the cdf of Y,

$$\begin{aligned} Pr[Y \leq y] &= Pr[F_X^{-1}(U) \leq y] \\ &= Pr[U \leq F_X(y)] \\ &= F_X(y) \end{aligned}$$

provided that the cdf of $U(0,1)$ is $F_U(u) = u$.

Above shows that Y has the cdf $F_X(x)$. Hence Y and X follows the same distribution. i.e. $Y = X$

Now we take **exponential distribution** for example and apply **ITS** on it.

Use ITS to generate an exponential distribution

We do the case $X \sim exp(\theta)$ where $\theta = 5$.

First we generate 10^5 random numbers from $[0,1]$.

(See Figure 10)

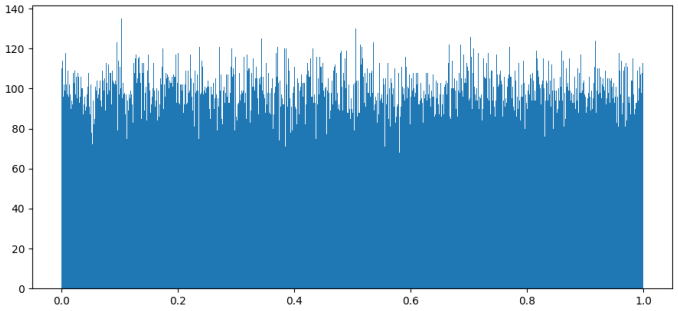


Figure 10: Random numbers from (0,1)

Note that the inverse cdf of exponential distribution is

$$x = F^{-1}(y) = -\theta \ln(1-y), \quad 0 \le y \le 1 \ ,$$

Now put the random numbers into $F^{-1}(y)$, we get the figure below. (See Figure 11)

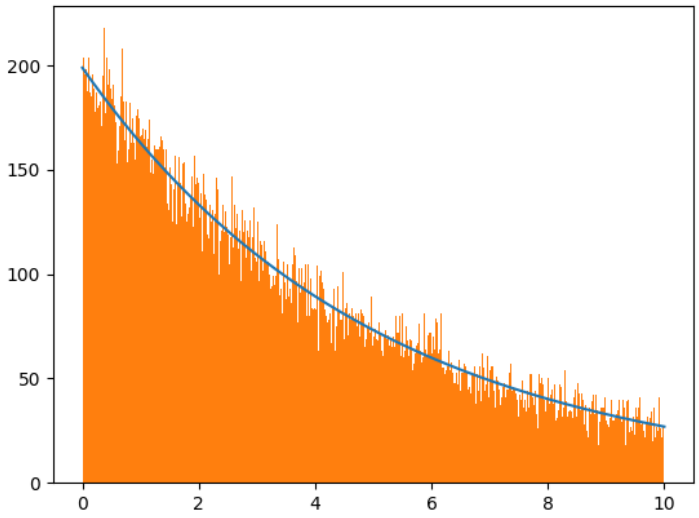


Figure 11: The random numbers after ITS

Note that the curve is the **pdf** of $exp(\theta = 5)$, it seems like the histogram fit well in with the curve. But now here comes a problem : It is **NOT** always can we find the **close-form expression** of an inverse cdf function easily. The subsection later will show how we deal with this problem on a **normal distribution**.

Code

```
import matplotlib.pyplot as plt
import time
import math


# 定義 exp(5) 的 pdf
def exp_theta(x, theta=5):
    return math.exp(-x / theta) / theta


# 定義 exp(5)cdf 的反函數
def inverse_exp_theta(y, theta=5):
    return (-1) * theta * math.log(1 - y)


# LCG 演算法
def LCG(x0):
    # 定義 LCG 所需參數
    a = 6364136223846793005
    c = 1
    m = 2 ** 64

    while (True):
        x0 = (a * x0 + c) % m
        yield x0


def RNG(tm):
    # tm 為取隨機數的次數
    t = time.time()  # 取得時間
    seed = int((t - int(t)) * (2 ** 64))  # 取 seed 為時間取小數後 *2^64
    G = LCG(seed)

    result = []  # 創建 list

    for i in range(tm):
        n = next(G)  # n=xi
```

```
        n = (n // 2 ** 32) / 2 ** 32  # 將 n 送進 [0,1]
        result.append(inverse_exp_theta(n, theta=5))  # 將 n 丟進 exp(5)cdf 的反函數
    return result  # 將結果儲存至 list


prob_a1 = 1 - math.exp(-10 / 1000)
print(prob_a1)
L = RNG(100000)
X = [x / 5000 for x in range(0, 10000 * 5, 1)]
Y = [exp_theta(x, theta=5) * prob_a1 * 100000 for x in X]
plt.plot(X, Y)  # 將 exp(5) 的 pdf* 累積次數 * 相對機率後畫出
plt.hist(L, bins=1000, range=(0, 10))  # 將 (0,10) 切成 1000 等份後將亂數記數並且把直方圖畫出
plt.show()
```

3.3 Box-Muller Transform

In previous subsection we've seen how **ITS** worked. But how can we do when we want to generate a normal distribution which its inverse cdf function can **NOT** be expressed in the close-form? To deal with this problem, we are now going to introduce the **Box-Muller Transform**.

Roughly, the **Box–Muller Transform**, by George Edward Pelham Box and Mervin Edgar Muller, is a **random number sampling** method for generating pairs of independent **standard normal distribution**, given a source of **uniformly distributed** random numbers.

Basic form of Box-Muller Transform

Assume U_1, U_2 are two random variables identically independent distributed (abbreviate it as **i.i.d**) form $U(0,1)$. Now define Z_1, Z_2 by

$$Z_1 = Rcos(\theta)\ ,\ Z_2 = Rsin(\theta)\ ,\ where\ R = \sqrt{-2\ln U_1}\ ,\ \theta = 2\pi U_2$$

then Z_1, Z_2 would be two random variables **i.i.d** form $N(0,1)$, i.e. we would get two independent **standard normal distributions**.

Proof of Box-Muller Transform

Let $U_1, U_2 \sim U(0,1)$ be **i.i.d** from standard uniform distribution.
Define Z_1, Z_2 by

$$Z_1 = Rcos(\theta)\ ,\ Z_2 = Rsin(\theta)\ ,\ where\ R = \sqrt{-2\ln U_1}\ ,\ \theta = 2\pi U_2$$

Then we can write U_1, U_2 as the functions of Z_1, Z_2.

$$u_1(z_1, z_2) = e^{\frac{-(z_1^2+z_2^2)}{2}}$$

$$u_2(z_1, z_2) = \frac{1}{2\pi} \tan^{-1}\left(\frac{z_2}{z_1}\right)$$

Let $f(u_1, u_2)$ be the joint pdf of U_1, U_2, then the joint pdf of Z_1, Z_2, denoted by $g(z_1, z_2)$, could be expressed as

$$\begin{aligned}
g(z_1, z_2) &= f(u_1(z_1, z_2), u_2(z_1, z_2)) \cdot |J| \\
&= f(u_1(z_1, z_2), u_2(z_1, z_2)) \cdot \left|\begin{vmatrix} \frac{\partial u_1}{\partial z_1} & \frac{\partial u_2}{\partial z_1} \\ \frac{\partial u_1}{\partial z_2} & \frac{\partial u_2}{\partial z_2} \end{vmatrix}\right| \\
&= f(u_1(z_1, z_2), u_2(z_1, z_2)) \cdot \left|\begin{vmatrix} -z_1 \cdot e^{\frac{-(z_1^2+z_2^2)}{2}} & -z_2 \cdot e^{\frac{-(z_1^2+z_2^2)}{2}} \\ \frac{-1}{2\pi} \cdot \frac{z_2}{z_1^2+z_2^2} & \frac{1}{2\pi} \cdot \frac{z_1}{z_1^2+z_2^2} \end{vmatrix}\right| \\
&= 1 \cdot \left|\frac{-1}{2\pi} \cdot e^{\frac{-(z_1^2+z_2^2)}{2}}\right| \\
&= \frac{1}{2\pi} \cdot e^{\frac{-(z_1^2+z_2^2)}{2}} = \left[\frac{1}{\sqrt{2\pi}} \cdot e^{\frac{-z_1^2}{2}}\right] \cdot \left[\frac{1}{\sqrt{2\pi}} \cdot e^{\frac{-z_2^2}{2}}\right]
\end{aligned}$$

Finally we can see that it is a product of two pdfs come from independent standard normal distribution. Therefore $Z_1, Z_2 \sim N(0, 1)$.

Apply Box-Muller Transform to get a normal distribution

First we generate two groups of 10^5 random numbers from $[0, 1]$. Denoted the groups by u_1, u_2 respectively.

Now do the transformations, let

$$Z_1 = R\cos(\theta)\ ,\ Z_2 = R\sin(\theta)\ ,\ where\ R = \sqrt{-2\ln U_1}\ ,\ \theta = 2\pi U_2$$

Now put the random numbers into the transformations, we get the figure below.

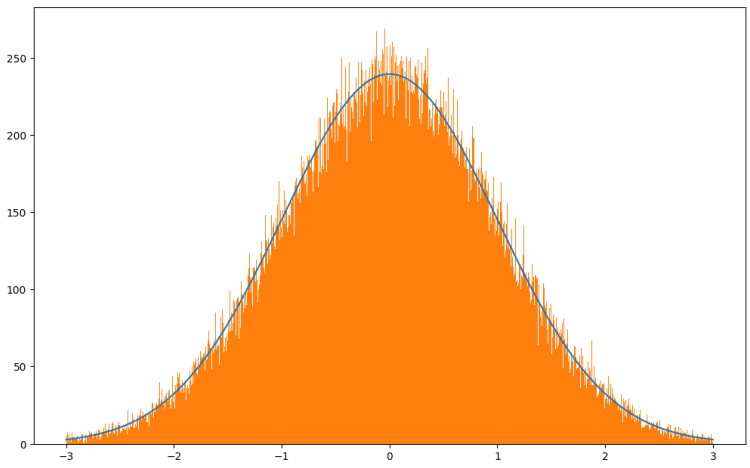


Figure 12: The random number z_1 after Box-muller transform

Code

```
import matplotlib.pyplot as plt
import time
import math
from scipy.stats import norm

# 定義 normal(0,1) 的 cdf
def normal_cdf(x):
    return norm.cdf(x)

# 定義 normal(0,1) 的 pdf
def normal01(x):
    return (math.exp(-(x**2)/2))/((2*math.pi)**0.5)

# Define R by ((-2)*ln(x)^0.5
def R_fun(x):
    return (-2*math.log(x)) ** 0.5

# Define theta by 2(pi)x
def theta_fun(x):
    return 2*math.pi*x

# Define z1 by R*cos(theta)
```

```
def z1(r, theta):
    return r*math.cos(theta)

# Define z2 by Rsin(theta)
def z2(r, theta):
    return r*math.sin(theta)

# LCG 演算法
def LCG(x0):
    # 定義 LCG 所需參數
    a = 6364136223846793005
    c = 1
    m = 2**64

    # 重複迭代 x
    while (True):
        x0 = (a*x0 + c) % m
        yield x0


def RNG(tm):
    # tm 為取隨機數的次數
    # return type: list[int]
    # 初始化
    t = time.time() # 取得時間
    seed = int((t-int(t))*(2**64)) # 取 seed 為時間取小數後 *2^64
    G = LCG(seed)

    # 創建 list
    result_z1 = []
    result_z2 = []
    result = []

    for i in range(tm//2):
        n = next(G) # n=x_(2i+1)
        n = (n // 2**32)/2**32 # 將 n 送進 [0,1]
        r = R_fun(n) # 轉換成 R
        n = next(G)# n=x_(2i+2)
        n = (n // 2**32)/2**32 # 將 n 送進 [0,1]
        theta = theta_fun(n) # 轉換成 theta
        # 把 R,theta 轉換成 normal 並丟進 result 內
        result_z1.append(z1(r, theta))
        result_z2.append(z2(r, theta))
    return result_z1, result_z2
```

```
Z1, Z2 = RNG(100000)
Z = Z1+Z2 # z1 跟 z2 都是 normal(0,1), 所以可以合併在一起
X = [x/1000 for x in range(-3000, 3000, 1)]
prob_ai = [normal_cdf(x+(6/1000)) - normal_cdf(x) for x in X]
Y = [(x[1])*100000 for x in zip(X, prob_ai)]
plt.plot(X, Y) # 將 normal(0,1) 的 pdf* 累積次數 * 相對機率後畫出
plt.hist(Z, bins=1000, range=(-3, 3))# 將 (0,10) 切成 1000 等份後將亂數記數並且把直方圖畫出，
plt.show()
```

4 The PRNG we make

These method is inspired by CMRG

4.1 First PRNG

4.1.1 Algorithm

STEP1
We consider m linear recurrences

$$a_{n,j} = q_{0,j}a_{n-r_j,j} + q_{1,j}a_{n-r_j+1,j} + ... + q_{r_j-1,j}a_{n-1,j} \quad (mod\ p_j),\ j = 1, ..., m,$$

satisfying that p_j is prime and $q_{0,j}, ..., q_{r_j-1,j} \in \{0, 1, ...p-1\}$ are chosen such that the polynomial

$$Q_j(x) = x^{r_j} - q_{r_j-1,j}x^{(r_j-1)} - ... - q_{1,j}x - q_{0,j}$$

is primitive over $\mathbb{F}_{p_j}$.
then we get

$$\frac{a_{n,j}}{p_j} \in [0, 1), j = 1, ..., m$$

STEP2
turn them into binary by following function

$$T_k(\frac{a_{n,j}}{p_j}) = \tilde{a}_{n,j} = (0.b_{n,j,1}b_{n,j,2}...b_{n,j,k})_2, j = 1, ..., m$$

where k is how many digits we need
STEP3
combine $\tilde{a}_{n,1}, \tilde{a}_{n,2}...\tilde{a}_{n,m}$ by the following function

$$C(\tilde{a}_{n,1}, \tilde{a}_{n,2}...\tilde{a}_{n,m}) = (0.b_{n,1,1}b_{n,2,1}...b_{n,m,1}b_{n,1,2}...b_{n,m,k})_2 = b_n$$

then b_n is the final output, period of this method is $\prod_{j=1}^{m}({p_j}^{r_j} - 1)$ if p_j are related prime

4.1.2 Example

Simple Example

$$Q_1(x) = x + 2, \quad p_1 = 3$$
$$Q_2(x) = x^2 + x + 1, \quad p_2 = 2$$

and seed $= 1$, then we have

$$\{\frac{a_{n,1}}{p_1}\} = \{\frac{1}{3}, \frac{2}{3}, ...\} = \{0.\overline{01}, 0.\overline{10}, ...\}_2$$

$$\{\frac{a_{n,2}}{p_2}\} = \{\frac{1}{2}, \frac{1}{2}, 0, ...\} = \{0.1, 0.1, 0, ...\}_2$$

combine

$$\{b_n\} = \{0.01\overline{1000}, 0.11\overline{0010}, 0.\overline{0010}, 0.11\overline{0010}, 0.01\overline{1000}, 0.\overline{1000}, ...\}_2$$

Useful Example

```
# Q1
def RNG1(times):
    t = time.time()
    m1 = 4294944443.0
    q11 = 527612.0
    q13 = 1370589.0
    SEED = int((t - int(t)) * (m1))
    s10, s11, s12 = SEED, SEED, SEED
    random_numbers = []

    for _ in range(times):
        p1 = q11 * s12 - q13 * s10
        k = p1 // m1
        p1 -= k * m1
        if p1 < 0.0:
            p1 += m1
        s10, s11, s12 = s11, s12, p1
        random_numbers.append(p1 / m1)
    return random_numbers

# Q2
def RNG2(times):
    t = time.time()
    m1 = 4294949027.0
    a12 = 1154721.0
    a14 = 1739991.0
    a15n = 1108499.0
```

```
    SEED = int((t - int(t)) * (m1))
    s10, s11, s12, s13, s14 = SEED+12345, SEED+1234, SEED+123, SEED+12, SEED+1
    random_numbers = []
    for _ in range(times):
        p1 = a12 * s13 - a15n * s10
        if (p1 > 0.0):
            p1 -= a14 * m1
        p1 += a14 * s11
        k = int(p1 / m1)
        p1 -= k * m1
        if (p1 < 0.0):
            p1 += m1
        s10 = s11
        s11 = s12
        s12 = s13
        s13 = s14
        s14 = p1
        random_numbers.append(p1/m1)
    return random_numbers

# decimal to binary
def dec_to_bin(x, k=10):
    L = []
    for i in range(k):
        n = -1-i
        if x > 2**n:
            L.append(1)
            x -= 2**n
        else:
            L.append(0)
    return L

# binary to decimal
def bin_to_dec(L):
    return sum([x*2**(-1-i) for i, x in enumerate(L)])

# combine
def combine(a, b):
    LA = dec_to_bin(a)
    LB = dec_to_bin(b)
    L = []
    for i in range(len(LA)):
        L.append(LA[i])
        L.append(LB[i])
    return bin_to_dec(L)
```

```python
# main function
def RNG(times):
    L1 = RNG1(times)
    L2 = RNG2(times)
    print(L2)
    L = [combine(a, b) for a, b in zip(L1, L2)]
    return L
```

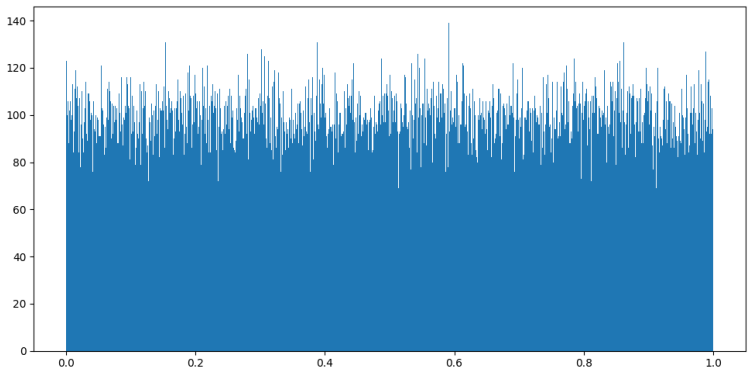


Figure 13: RNR_WeMake

4.2 Second PRNG

4.2.1 Algorithm

STEP1
We consider m linear recurrences similar to the First PRNG, but we set

$$a_{n,j+1} = q_{0,j}a_{n-r_j,j} + q_{1,j}a_{n-r_j+1,j} + ... + q_{r_j-1,j}a_{n-1,j} \quad (mod\ p_j),\ j = 1, ..., m,$$

which give number to the next one sequence.
STEP2
Get the number from the sequences individually.

$$\frac{a_{n,j}}{p_j} \in [0, 1), j = 1, ..., m$$

that is, the new sequence

$$S = \{\frac{a_{n,1}}{p_1}, \frac{a_{n,2}}{p_2}, \cdots, \frac{a_{n,m}}{p_m}\}_{n=max\{r_j\}}^{\infty}$$

4.2.2 Example

```
#CustomPRNG2 base on MRG32k3a,MRG32k5a
class CustomPRNG2():
    def __init__(self,seed=0):
        self.__seed=seed
        self.__x1=0
        self.__x2=0
        self.__x3=0
        self.__x4=0
        self.__index=1
        self.__components=4
        self.__rn=0

        #here we use parameters of MRG32k3a,MRG32k5a
        self.__m1 = 4294967087.0
        self.__a12 = 1403580.0
        self.__a13 = -810728.0
        self.__s10 = ((self.__seed+0b1111)%self.__m1)
        self.__s11 = ((self.__seed+0b11111)%self.__m1)
        self.__s12 = ((self.__seed+0b111111)%self.__m1)

        self.__m2 = 4294949027.0
        self.__a22 = 1154721.0
        self.__a24 = 1739991.0
        self.__a25 = -1108499.0
        self.__s20 = ((self.__seed+0b1111)%self.__m2)
        self.__s21 = ((self.__seed+0b11111)%self.__m2)
        self.__s22 = ((self.__seed+0b111111)%self.__m2)
        self.__s23 = ((self.__seed+0b1111111)%self.__m2)
        self.__s24 = ((self.__seed+0b11111111)%self.__m2)

        self.__m3 = 4294944443.0
        self.__a31 = 527612.0
        self.__a33 = -1370589.0
        self.__s30 = ((self.__seed+0b1111)%self.__m3)
        self.__s31 = ((self.__seed+0b11111)%self.__m3)
        self.__s32 = ((self.__seed+0b111111)%self.__m3)

        self.__m4 = 4294934327.0
        self.__a41 = 1776413.0
        self.__a43 = 865203.0
        self.__a45 = -1641052.0
        self.__s40 = ((self.__seed+0b1111)%self.__m4)
        self.__s41 = ((self.__seed+0b11111)%self.__m4)
        self.__s42 = ((self.__seed+0b111111)%self.__m4)
```

```
        self.__s43 = ((self.__seed+0b1111111)%self.__m4)
        self.__s44 = ((self.__seed+0b11111111)%self.__m4)

    # Component 1
    def Component1(self):
        #linear recurrence equation
        p1 = (self.__a12 * self.__s11)%self.__m1 \
                + (self.__a13 * self.__s10)%self.__m1
        p1 %=self.__m1
        if (p1 < 0.0):
            p1 += self.__m1

        self.__s10 = self.__s11
        self.__s11 = self.__s12
        #here we use number generated by Component4
        self.__s12 = (self.__x4%self.__m1)
        #here we give number to Component2
        self.__x1=p1
        self.__rn=self.__x1 / self.__m1
        return

    # Component 2
    def Component2(self):
        #linear recurrence equation
        p2 = (self.__a22 * self.__s23)%self.__m2 \
                + (self.__a24 * self.__s21)%self.__m2 \
                + (self.__a25 * self.__s20)%self.__m2

        p2 %=self.__m2
        if (p2 < 0.0):
            p2 += self.__m2

        self.__s20 = self.__s21
        self.__s21 = self.__s22
        self.__s22 = self.__s23
        self.__s23 = self.__s24
        #here we use number generated by Component2
        self.__s24 = (self.__x1%self.__m2)
        #here we give number to Component4
        self.__x2=p2
        self.__rn=self.__x2 / self.__m2
        return

    # Component 3
    def Component3(self):
        #linear recurrence equation
```

```
        p3 = (self.__a31 * self.__s32)%self.__m3 \
                + (self.__a33 * self.__s30)%self.__m3
        p3 %=self.__m3
        if (p3 < 0.0):
            p3 += self.__m3

        self.__s30 = self.__s31
        self.__s31 = self.__s32
        #here we use number generated by Component1
        self.__s32 = (self.__x2%self.__m3)
        #here we give number to Component3
        self.__x3=p3
        self.__rn=self.__x3 / self.__m3
        return

    # Component 4
    def Component4(self):
        #linear recurrence equation
        p4 = (self.__a41 * self.__s44)%self.__m4 \
                + (self.__a43 * self.__s42)%self.__m4 \
                + (self.__a45 * self.__s40)%self.__m4

        p4 %=self.__m4
        if (p4 < 0.0):
            p4 += self.__m4

        self.__s40 = self.__s41
        self.__s41 = self.__s42
        self.__s42 = self.__s43
        self.__s43 = self.__s44
        #here we use number generated by Component3
        self.__s44 = (self.__x3%self.__m4)
        #here we give number to Component1
        self.__x4=p4
        self.__rn=self.__x4 / self.__m4
        return

    #compute next random number
    def next_rnd(self):
        if self.__index==1:
            self.Component1()
        if self.__index==2:
            self.Component2()
        if self.__index==3:
            self.Component3()
        if self.__index==0:
```

```
            self.Component4()
        self.__index+=1
        self.__index%=self.__components
        return

    #get next random number (between 0 and 1)
    def random(self):
        self.next_rnd()
        return self.__rn

    #get random number between low and high
    def rnd_in_I(self,low=1, high=10):
        self.next_rnd()
        return  self.__rn*(high-low)+low
```

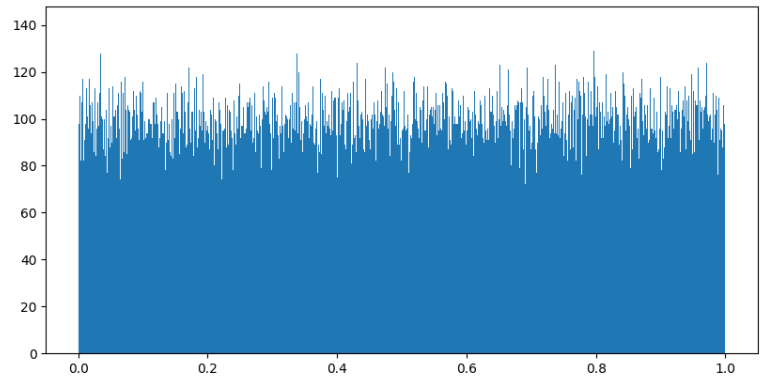


Figure 14: Second PRNG generates 100k numbers

5 Conclusion

When will we need random number?

1. Encryption: In encryption systems, random numbers can be used to generate keys, making them difficult to guess or crack.
2. Simulation and Testing: In scientific research or engineering, random numbers are needed to simulate complex systems, representing uncertainty in the real world.
3. Games and Applications: Random numbers can be used for random events in games, generating random maps, or in applications for random recommendation features.

4. Statistics and Probability: In statistical analysis, machine learning, and probability models, random numbers are used to generate samples or simulate uncertainty.

5. Security Verification: In testing for security vulnerabilities or conducting security audits, using random numbers can simulate the behavior of attackers.

6. Network Protocols: In certain network protocols, random numbers are used to generate challenging data to verify identity or ensure the security of communication

References

[1] NTNU. Finite field. https://web.ntnu.edu.tw/~algo/FiniteField.html.

[2] A. R. Hull, T. E.; Dobell. Random number generators. *Industrial and Applied Mathematics*, 4(3):230–254, 1962.

[3] Primitive polynomials in lfsrs. https://math.stackexchange.com/questions/819259/primitive-polynomials-in-lfsrs.

[4] Pierre L'ecuyer. Good parameters and implementations for combined multiple recursive random number generators. *Operations Research*, 47(1):159–164, 1999.

[5] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):4–30, 1997.

[6] M. MATSUMOTO and Y. KURITA. Twisted gfsr generators. *ACM Trans. Model. Comput. Simul.*, 2(3):179–194, 1992.

[7] M. MATSUMOTO and Y. KURITA. Twisted gfsr generators ii. *ACM Trans. Model. Comput. Simul.*, 4(3):254–266, 1994.

[8] T. G. LEWIS and W. H. PAYNE. Generalized feedback shift register pseudorandom number algorithm. *Journal of the ACM*, 20(3):456–468, 1973.

[9] C++ Reference. std::mersenne_twister_engine. https://en.cppreference.com/w/cpp/numeric/random/mersenne_twister_engine.