Random number generator

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

National Tsing Hua University

September 21, 2024

Table of Contents I

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.

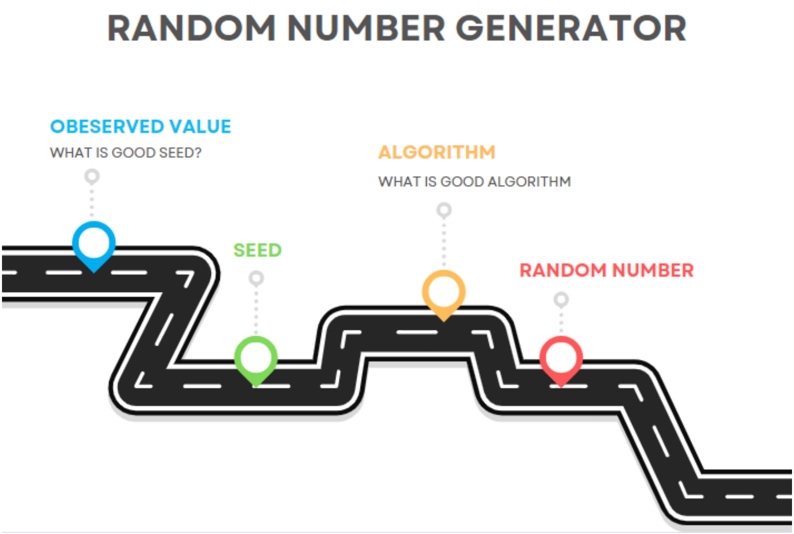

RANDOM NUMBER GENERATOR
OBESERVED VALUE
WHAT IS GOOD SEED?
ALGORITHM
WHAT IS GOOD ALGORITHM
SEED
RANDOM NUMBER

Figure: FLOW

Table of Contents

LCG Algorithm

$x_0 = seed$
$x_{n+1} = ax_n + c \ (mod\ m)$

Parameters selection:
(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

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

Easy Example

For m = 8, c = 1, a = 5

$x_{n+1} = 5 * x_n + 1 \ (mod\ 8)$

We start from seed $= x_0 = 1$.

The result is as following:

1-> 6-> 7-> 4-> 5-> 2-> 3-> 0-> 1

Observation

For the third parameter selection, we observed that if m is even, a and c is odd, then $a * (2k+1) + c \ (mod\ m)$ is even and $a * (2k+2) + c \ (mod\ m)$ is odd.

So we need some modification, for example if $m = 2^r$, let $\mathbf{X}_n = (x_r x_{r-1} ... x_1)_2$, and $\tilde{\mathbf{X}}_n = (x_k x_{k-1} ... x_{l+1} x_l)_2$,where $r \geq k \geq l$, then we will get new random integer in $[0, 2^{k-l+1})$

Useful Example

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

Finite Field

Consider the finite field (Galois field) of order 2^n(denoted $\mathbb{F}_{2^n}, GF(2^n), or\ \mathbb{Z}/2^n\mathbb{Z})$
Also represent by polynomial
$a_0 + a_1 x + ... + a_{n-1}x^{n-1}, a_i \in \{0, 1\}$.

Ex. $\mathbb{F}_{2^3}$ has the finite number of elements $\{0, 1, ..., 2^3 - 1\}$.
Take $\alpha = x^2 + 1\ in\ \mathbb{F}_{2^3}$.
α can be viewed as binary: $1 * 2^2 + 0 * 2^1 + 1 * 2^0 = 5$.

Ex.

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

$(x^2 + x) + (x^2 + 1) = 2x^2 + x + 1 = x + 1 \ (6 + 5 \equiv 3)$

$(x^2 + 1) * (x^2) = x * x^3 + x^2 = (x^2 + x) + x^2 = x \ (5 * 4 \equiv 2)$

+	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: addition

Ex.

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
$(x^2+x)+(x^2+1)=2x^2+x+1=x+1\ (6+5\equiv 3)$
$(x^2+1)*(x^2)=x*x^3+x^2=(x^2+x)+x^2=x\ (5*4\equiv 2)$

*	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: multiplication

Primitive polynomial

$F(x)$ 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** (the leading coefficient is 1) and has a root α in $\mathbb{F}_{p^n}$.
$\mathbb{F}_{p^n} = \{0, 1 = \alpha^0 = \alpha^{p^m-1}, \alpha, \alpha^2, ..., \alpha^{p^m-2}\}$.

What's more, if $F(x)$ is a primitive polynomial, then x is always a primitive element of the field.

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

Definition

1. Boolean algebra uses true value(False=0,True=1) as variable, logical operators like AND, XOR as operators.
2. flip flops are used to store a single bit of binary data (1 or 0).
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 $\mathbb{F}_2$ .Thus we use *XOR* to be the linear function in practice directly.

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

Table: truth table of XOR

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

Table: truth table of AND

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). All digits shift to the right and the leftmost bit s_{n-1} is replaced by s_n.

i.e.
$(s_{n-1}, ..., s_1, s_0) \to (s_n, ..., s_2, s_1)$ where
$s_n = f(s_0, s_1, ..., s_{n-1}) = (a_0 \wedge s_0) \oplus (a_1 \wedge s_1) \oplus ... \oplus (a_{n-1} \wedge s_{n-1})$,
f : linear function , $a_i = \begin{cases} 1 & \textit{if it's a tap} \\ 0 & \textit{otherwise} \end{cases}$.

LSFR[3,1]

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.

	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

cycle

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. 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.

Not all feedback polynomial can make the maximal-length period. In fact, the LFSR is maximal-length iff the feedback polynomial is a primitive polynomial in $\mathbb{F}_2$.

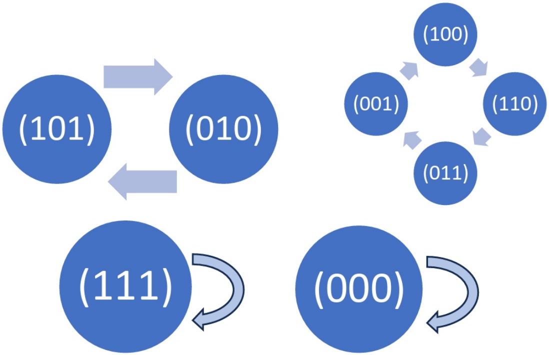


Figure: $x^3 + x^2 + x + 1$
maximal period=4

```
start_state = 1 << 2 | 1
lfsr = start_state
period = 0
while(True){
    bit = (lfsr ^ (lfsr >> 1) ) & 1
    lfsr = (lfsr >> 1) | (bit << 3)
    period += 1
    print(f"({bin(lfsr)}) {lfsr}")
    if (lfsr == start_state):
        print(f"period = {period}")
        break
}
```

```
import galois
def LS(seed=1,n=64,times=10):
    seed %= 1 << n
    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)):
            bit^= (seed>>poly[j])
        bit&=1
        seed = (seed >> 1) | (bit << (n-1))
        print(f"{i}. {seed}")
```

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

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

Example

$$Q_1(x) = x + 2, \quad p_1 = 3$$

$$a_i = 2 * a_{i-1}, \quad p_1 = 3$$

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 $p^r - 1$

MRG32K3aa

Since it's part of MR32K3a so I called it MR32K3aa
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:

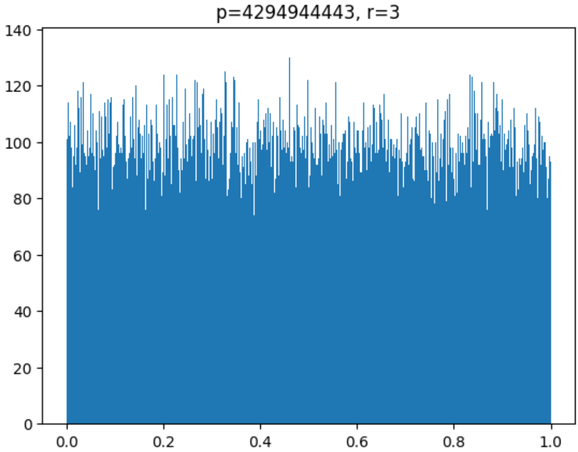


Figure: MRG32K3aa generate 100k numbers

Introduction

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.

Definition

We consider m linear recurrences

$a_{n,j} = q_{0,j}a_{n-r,j} + q_{1,j}a_{n-r+1,j} + \ldots + q_{r-1,j}a_{n-1,j} \pmod{p_j}\ j = 1, \ldots, m$

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

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

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 such that 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. (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.

Example

Consider a CMRG with these two recurrences:

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 $u_n = \dfrac{a_n}{3} + \dfrac{b_n}{2}$

Starting with initial condition 001
(i.e. $a_0 = b_0 = 0, a_1 = b_1 = 0, a_2 = b_2 = 0$)
We get the result of above recurrences, 1st linear recurrence generator (LRG) gives us the following sequence of period 26:

$$00101211201110020212210222$$

and the 2nd LRG gives 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 28 data):

$$\begin{array}{ccccccc|ccccccc|}
0 & 0 & 1 & 0 & 1 & 2 & 1 & 1 & 2 & 0 & 1 & 1 & 1 & 0 \\
0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\
0 & 0 & \frac{5}{6} & 0 & \frac{5}{6} & \frac{1}{6} & \frac{5}{6} & \frac{2}{6} & \frac{4}{6} & \frac{3}{6} & \frac{2}{6} & \frac{5}{6} & \frac{5}{6} & \frac{3}{6} \\
\\
0 & 2 & 0 & 2 & 1 & 2 & 2 & 1 & 0 & 2 & 2 & 2 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\
0 & \frac{4}{6} & \frac{3}{6} & \frac{4}{6} & \frac{5}{6} & \frac{1}{6} & \frac{1}{6} & \frac{2}{6} & 0 & \frac{1}{6} & \frac{4}{6} & \frac{1}{6} & \frac{3}{6} & \frac{3}{6}
\end{array}$$

MRG32k3a

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

$$a_n = 1403580 \cdot a_{n-2} - 810728 \cdot a_{n-3} \pmod{2^{32} - 209}$$

$$b_n = 527612 \cdot b_{n-1} - 1370589 \cdot b_{n-3} \pmod{2^{32} - 22853}$$

$$c_n = a_n - b_n$$

$$d_n = \frac{c_n}{2^{32} - 209}$$

Although MRG32k3a only combined 3 generators and each generator only has 2 components, it gives a wonderful random number output with a period of 2^{191}. We 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 8.

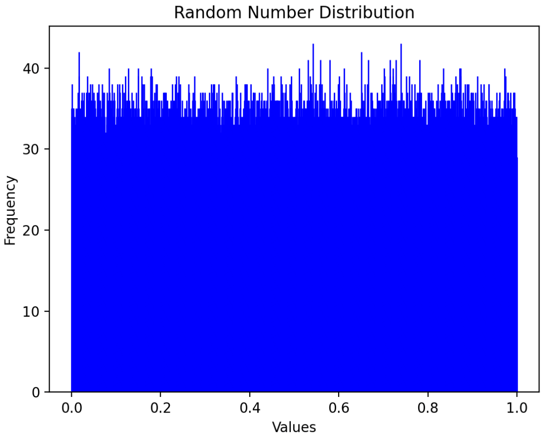


Figure: 10M Datas of MRG32k3a

Mersenne Twister

Makoto Matsumoto and Takuji Nishimura (1997) develope the MT algorithm provides a super astronomical period of $2^{19937} - 1$ and 623-dimensional equidistribution up to 32-bit accuracy.
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

Twist

Twist is defined as:

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

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:

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

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

Tempering

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

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

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.

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.

Coefficients in general

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

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

and the coefficients for std::mt19937_64 are:

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

Pseudo Code

```
##MersenneTwister
class MersenneTwister:
    def __init__():
        ##Initialize the generator from a seed

    def __twister(self):
        ##Generate the next n values from the series X
        for i in range(self.__n):
            ##Get (X_i (upper)|X_i+1 (lower))
            ##i.e. concatenating the upper w - r bits

            ##Compute XA
            ##Compute X_k+n (= X_k+m XOR XA)
        return
```

```
def __temper(self):
    ##Extract a tempered value based on MT[index]
    ##Calling twist() every n numbers

    ##Tempering

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

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

Table of Contents

Parameters

Linear Congruential Generator:
$a = 6364136223846793005,\ c = 1,\ m = 2^{64}$

Linear Feedback Shift Register:
primitive polynomial of $n = 64$ degree

$\mathbb{F}_p$ Linear Generator:
primitive polynomial of $r = 3$ degree,
$p = 2^{32} - 22853$

Combined Multiple Recursive Generator:
$n = 2$ primitive polynomials,
primitive polynomial of $r = 3$ degree,
$p_1 = 2^{32} - 209,\ p_2 = 2^{32} - 22853$

Parameters

Mersenne Twister(mt19937):

$$
\begin{aligned}
(w, n, m, r) &= (32, 624, 397, 31) \\
\boldsymbol{a} &= \mathrm{9908B0DF}_{16} \\
(u, s, t, l) &= (11, 7, 15, 18) \\
(\boldsymbol{b}, \boldsymbol{c}) &= (\mathrm{9D2C5680}_{16}, \mathrm{EFC60000}_{16}) \\
f &= \mathrm{6C078965}_{16}
\end{aligned}
$$

Function

Linear Congruential Generator: $1 \to 1$

$x_{i+1} = f(x_i)$

Linear Feedback Shift Register: $1 \to 1$

$x_{i+1} = f(x_i)$

$\mathbb{F}_p$ Linear Generator: $r \to 1$

$x_{i+1} = f(x_i, x_{i-1}, \ldots, x_{i-(r-2)}, x_{i-(r-1)})$

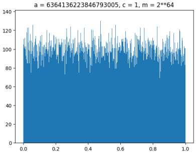
Combined Multiple Recursive Generator: $r \times j \to 1$

$x_{i+1} = f(x_{i,1}, x_{i,2}, \ldots, x_{i,j}, \ldots, x_{i-(r-1),1}, x_{i-(r-1),2}, \ldots, x_{i-(r-1),j})$

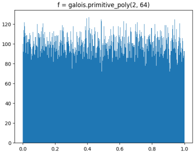
Mersenne Twister(mt19937): $n \to n$

$(x_i, x_{i+1}, \ldots, x_{i+(n-2)}, x_{i+(n-1)}) = f(x_{i-1}, x_{i-2}, \ldots, x_{i-(n-1)}, x_{i-n})$

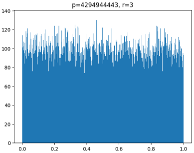
Graphs



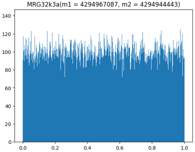
(a) LCG



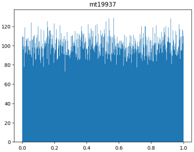
(b) LFSR



(c) Fp



(a) CMRG



(b) mt19937

Figure: all generate 100k

Period

Linear Congruential Generator: $km(modulus)$, $k = 1, \frac{1}{4}$

Linear Feedback Shift Register: $2^{n(bit\ length)} - 1$

$\mathbb{F}_p$ Linear Generator: $p^{r(deg\ of\ primitive\ polynomial)} - 1$

Combined Multiple Recursive Generator: $\prod_{j=1}^{n(recurrences)} (p_j^{r_j} - 1)$

Mersenne Twister(mt19937): $2^{19937} - 1 \approx 2 \times 10^{6001}$

Table of Contents

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 $F^{-1}(y)$

Example

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

Note that the inverse cdf of exponential distribution is

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

Now put the random numbers into $F^{-1}(y)$. (Here theta=5.)

Graph

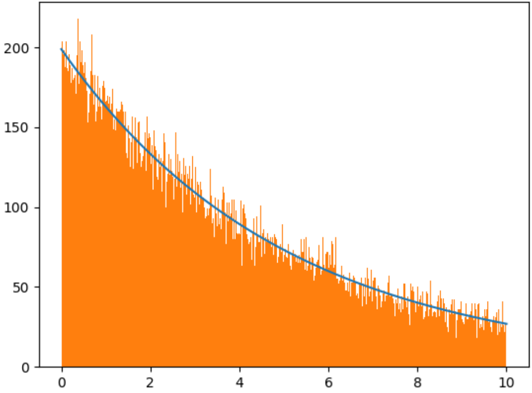


Figure: The random numbers after ITS

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

$$\begin{aligned} Z_1 &= R\cos(\theta), \quad \text{where } R = \sqrt{-2\ln U_1} \\ Z_2 &= R\sin(\theta), \quad \theta = 2\pi U_2 \end{aligned}$$

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**.

Example

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

$$Z_1 = \sqrt{-2\ln U_1}\cos(2\pi U_2), \tag{5}$$

$$Z_2 = \sqrt{-2\ln U_1}\sin(2\pi U_2) \tag{6}$$

Now put the random numbers into the transformations.

Graph

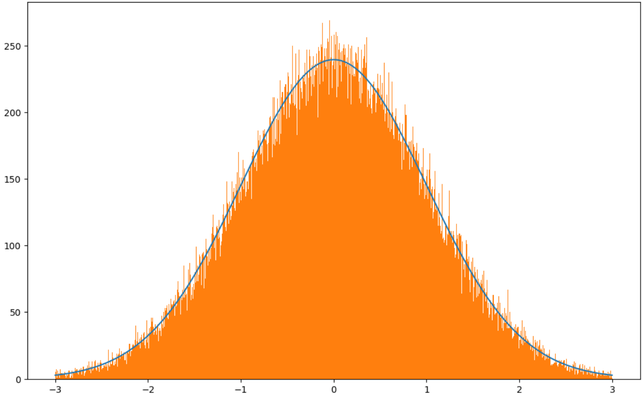


Figure: The random number after Box-muller transform

Table of Contents

Algorithm

```
def f(time,domain_size):
    return (time-int(time))*domain_size
```

Algorithm

STEP1

We consider m linear recurrences

$$a_{n,j} = q_{0,j}a_{n-r_j,j} + q_{1,j}a_{n-r_j+1,j} + \dots + q_{r_j-1,j}a_{n-1,j} \pmod{p_j},\ j = 1, \dots,$$

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

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

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

then we get

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

Algorithm

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

Simple Example

primitive polynomials:

$$Q_1(x) = x + 2, \quad p_1 = 3$$

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

let seed $= 1$, then we have

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

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

combine: $\{b_n\} =$

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

Useful Example

primitive polynomials:

$Q_1(x) = x^3 + 527612x^2 - 1370589,\ p_1 = 4294944443$

$Q_2(x) = x^5 + 1154721x^3 + 1739991x - 1108499,\ p_2 = 4294949027$

in this case the period is

$4294944443^2 * 4294949027^3 \approx 1.4615 * 10^{46}$

pseudo code

```
# Q1
def RNG1(times):
    return random_numbers_generated_by_Q1

# Q2
def RNG2(times):
    return random_numbers_generated_by_Q2

# decimal to binary
def dec_to_bin(x, k=10):
    return last_k_digits_of_x_in_in_binary_type

# binary to decimal
def bin_to_dec(L):
    return L_in_decimal
```

pseudo code

```
# 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) # return C(a,b)
# 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
```

Algorithm

STEP1

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

$$a_{n,j+1} = q_{0,j}a_{n-r_j,j} + q_{1,j}a_{n-r_j+1,j} + \dots + q_{r_j-1,j}a_{n-1,j} \pmod{p_j},$$

$$j = 1, \dots, m$$

which give number to the next one sequence.

Algorithm

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

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

Pseudo Code

```
## Component k = 1, ..., m
def Component(k)(self):
    ##linear recurrence equation
    ##a[n-1][k] use number from Component(k-1)
    ##then give number to Component(k+1)

##Compute next random number
def next_rnd(self):
    ##Alternately taken
    ##from Component(1) to Component(m)
```

Random number generator

Pseudo random number generator(PRNG)
Linear Congruential Generator(LCG)
Linear Feedback Shift Register
$\mathbb{F}_p$-Linear Generator
Combined Multiple Recursive Generator
The Mersenne Twister algorithm
Discussion
Statistic
Inverse Transform Sampling
Box-Muller Transform
The PRNG we make
Seed function
The first RNG
The second RNG
Conclusion

Table of Contents

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.

1. Statistics and Probability: In statistical analysis, machine learning, and probability models, random numbers are used to generate samples or simulate uncertainty.
2. Security Verification: In testing for security vulnerabilities or conducting security audits, using random numbers can simulate the behavior of attackers.
3. Network Protocols: In certain network protocols, random numbers are used to generate challenging data to verify identity or ensure the security of communication