Study on Random Number Generators

Mahmuda Anwar
Student Id: 011132007
Sayla Zahan Surovi
Student Id: 011132011
Rusda Islam Toma
Student Id: 011132024
Murshad Jahan Meghna
Student Id: 011132025

A thesis in the Department of Computer Science and Engineering presented
in partial fulfillment of the requirements for the Degree of
Bachelor of Science in Computer Science and Engineering

United International University
Dhaka, Bangladesh
January 2018

Declaration

We, Mahmuda Anwar, Sayla Zahan Surovi, Rusda Islam Toma, Murshad Jahan Meghna, declare that this thesis titled, Study on Random Number Generators and the work presented in it are our own. We confirm that:

- This work was done wholly or mainly while in candidature for a BSc degree at United International University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at United International University or any other institution, this has been clearly stated.
- Where we have consulted the published work of others, this is always clearly attributed.
- Where we have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely our own work.
- We have acknowledged all main sources of help.
- Where the thesis is based on work done by ourselves jointly with others, we have made clear exactly what was done by others and what we have contributed ourselves.

_____

Mahmuda Anwar, 011132007 and Computer Science and Engineering

_____

Sayla Zahan Surovi, 011132011 and Computer Science and Engineering

_____

Rusda Islam Toma, 011132024 and Computer Science and Engineering

_____

Murshad Jahan Meghna, 011132025 and Computer Science and Engineering

Certificate

I do hereby declare that the research works embodied in this thesis entitled "**Study on Random Number Generators**" is the outcome of an original work carried out by Mahmuda Anwar, Sayla Zahan Surovi, Rusda Islam Toma, Murshad Jahan Meghna under my supervision.

I further certify that the dissertation meets the requirements and the standard for the degree of BSc in Computer Science and Engineering.

_____

Dr. Mohammad Nurul Huda
Professor and MSCSE Coordinator
Department of Computer Science and Engineering
United International University, Dhaka, Bangladesh

Abstract

Study on Random Number Generators is the most useful task in simulation. It helps us to solve real life problems like science, gaming, simulation, lottery etc. We study on different types of Random Number Generators, but it is quite difficult to find out the better random number generators. If we are testing Random Number Generators it is easier to figure out the good generator. In this paper, we include comparative investigation of different types of Random Number Generators. These generators are implemented by using c programming language and as an editor we used CodeBlocks.

Acknowledgement

We would like to grateful to the Almighty Allah for giving us the ability and the strength to finish the task successfully within the scheduled time.
This thesis titled “Study on Random Number Generators” has been prepared to fulfill the requirement of BSC degree. We are very much fortunate that we have received sincere guidance, supervision and co-operation from various persons.

We would like to thank to our Supervisor Dr. Mohammad Nurul Huda, Professor and MSCSE Coordinator , Head Examiner Dr. Swakkhar Shatabda, Associate Professor and Under Graduate Coordinator and Ex-Officio Dr. Salekul Islam, Associate Professor and Departmental Head

Finally, our deepest gratitude and love to our parents for their supports, encouragement and endless love.

Table of Contents

LIST OF TABLES

Chapter 1

Introduction

This paper is a summary of the research we conducted during our study. Our main goal is to brifely study on random number generators.This study will help us to gather some knowledge about random numbers and different types of methods of it.For determining the performance of those methods we can use some different types of tests for example Empirical Test(chi-square test,serial test,runs-up test),Theoritical Test etc.

1.1 what is random number? [4]

Random numbers are a secquence of numbers which can be generated from a large set of numbers and some mathmatical algorithms.These algorithms give equal probability to all numbers occured in specific distribution.

Random numbers should follow some rules that are,the values are generated from U[0,1] distribution and there cannot be any pseudorandomness.Pseudo randomness means not to predict future values based on the past on the present ones, that's how the random numbers become independent.The frequency of the occurence of these random numbers should approximately be the same.For these reasons,it is difficult to produce a large random number.

1.2 Two principal methods are used to produce random numbers [5]

The first method is used to measure Physical Phenomenon where it should be random.As a source of example we can generate random number by measuring thermal noise, atmospheric noise and other external electromagnetic and quantum phenomena etc.

The second one is computational algorithms from which we can produce large sequences of random number.Generally we use initial seed value,supposedly as the seed value is known then entire random sequences can be reproduced.This procedure is called psudorandom number generator.

1.3 Properties of Good random number generator [6]

- Should be uniformly distributed on [0,1] and should not exhibit any correlation with each other.
- Random number generator should be fast and avoid the need for a lot of storage.
- Random number generator should be able to reproduce a given stream of random numbers.

  Purposes:

  - Debugging or verification
  - For comparing two systems
- Random number generator should have capability to produce seperate “streams” of random numbers.

  Example:(Quening model)

  - One stream of random numbers for interarrival time.
  - One stream of random numbers for service time.
- For all standard computers and compliers we expected that the generators to be portable to generate same squence of random number

Chapter 2

Background

2.1 History [1]

There has a long and interesting history of generating random number. At the beginning people were used different types of methods for generating random numbers such as throwing dice, casting lots, dealing out cards, lottery etc. in the late 1960s and early 1970s. In early 20th century statisticians invent a device which generates random numbers more quickly in late 1930s. In 1938 Kendall and Babington-Smith had discovered a table which generates 1000,000 random numbers by using spinning disk. After some times electric circuits based on randomly pulsating vacuum tubes were invented that produce up to 50 random digits per second. In 1955s RAND corporation had used an electronic device to generate a table of millions of random digits. To pick the winners in the premium savings bond lottery the British general post office had used Electronic Random Number Indicator Equipment (ERNIE) in 1959s. Later people became more interested building and testing physical random number machines then In 1983s Miyatake et al. proposed a device which was based on counting gamma rays.

When computers became more popular people started to pay more attention on the methods of random number generation for the use of computer works. Then people invented a way to hook up an electronic random-number machine, such as ERNIE with the computer, but this procedure has some several disadvantages that it could not reproduce exactly a previously generated random number stream. Research in 1940s and 1950s found *numerical* or *arithmetic* ways to produce random numbers which are sequential.

2.2 Where and why we can use Random Number? [2]

Random numbers have many applications in real life such as science, computer simulation, cryptography, statistics, gaming, gambling, art etc. Random number is also used in other areas where unpredictable result is desirable.

2.2.1 Science

Random numbers have a lot of use in physics such as electronic noise studies, engineering, operations and research. Bootstrap method of statistical analysis, Monte Carlo methods in physics and computer science require random number. Random number is also used in parapsychology.

2.2.2 Computer simulation

Computer simulations of real phenomena are used in engineering and scientific fields. Sometimes unpredictable processes affect the real phenomena such as day to day weather and radio noise are simulated by pseudo-random numbers.

2.2.3 Statistics

Many statistical theories depend on randomness via random numbers and perfect random number is needed for the actual systematic bias. Choosing a representative sample of the population being examined and disguising the protocol of a study from a participant these are included in statistical practice.

2.2.4 Cryptography

Cryptography provides security (confidentiality, authentications, authorization, electronic commerce etc.) in modern communications. Randomness is very important for cryptographic analysis. In encryption algorithm we can select the random number as the key and that is the best way. Thus it makes difficult for attacker to attack the system.

2.2.5 Other Uses

In Jurors, military draft lotteries, computer game, hash algorithms, sorting algorithms, Queuing models, inventory models etc. are used random numbers.

Chapter 3

Different Random Number Generators

3.1 Midsquare Method [3]

In 1940s Von Neumann and Metropolis at first proposed an arithmetic generator which is called the midsquare method.

Procedure of midsquare method:

Step 1 : a four-digit positive integer , Z_0 and i=0

Step 2: Squre Z_i to obtain an integer upto eight digits.

Step 3: if Z_i^2 does not contain eight digits append zeros to the left to make it exactly eight digit.

Step 4: Z_{i+1}= select middle four digits of Z_i^2

Step 5: place a decimal point at the left of Z_{i+1} to obtain random number U_{i+1}

Step 6: i=i+1 go to step 2 if all required random numbers are not generate.

3.1.1 Example for Mid Square Method

Table 3.1: Mid square method where initial seed, Zo = 1234

i	Z_i	Z_i^2	U_i
0	1234	01522756	-
1	5227	27321529	0.5227
2	3215	10336225	0.3215
3	3362	11303044	0.3362
4	3030	09180900	0.3030
.	.	.	.
.	.	.	.

3.1.2 Code for Mid Square

```
procedure seed_generator(int seed)
        VAR square_value, length_of_number, length_of_seed, double_seed:integer
         VAR array_one[100], array_two[100], array_three[100], array_four[100]:integer
        if seed>0 then
            length_of_seed ← floor(log10(abs(seed)))+1
         else
            length_of_seed ← 1
        end if
         double_seed←length_of_seed*2

        square_value←seed*seed

        If square_value>0 then

                length_of_number←floor(log10(abs(square_value)))+1

         else

                 length_of_number←1

        end if

         i←0

         While square_value>0 do

             digit←square_value mod 10

             square_value←square_value divisible by 10

             array_one[i] ←digit

             increment i

        end while

         j←0

         For  i length_of_number-1 downto 0
```

```
        array_two[j] ←array_one[i]
        increment j
    end for
    number_of_space←double_seed-length_of_number
    if length_of_number<double_seed ||length_of_number==double_seed then
        for  i 0 to double_seed-1
                array_three[i+number_of_space] ←array_two[i]
        end for
        for i 0 to number_of_space-1
                array_three[i] ←0
        end for
  end if
        k←0
 half_of_seed←length_of_seed/2
for i half_of_seed  to length_of_seed+half_of_seed-1
        array_four[k] ←array_three[i]
         increment k
 end for
 p←0
for i 0 to length_of_seed-1
        p=10*p+array_four[i]
end for
return p
```

```
end procedure

procedure main()

        clock_t begin, end

        double time_spent

        srand(time(NULL))

        Seed, number, new_seed

        scan seed

        scan random number

        begin←clock()

        for i  1 to number

            new_seed ←seed_generator(seed)

            seed←new_seed;

            float randomNumber ← ((float)new_seed/10000)

            Print i and randomNumber

        end for

        end←clock()

        time_spent← (double)(end-begin)/CLOCKS_PER_SEC

        print time_spent

    end procedure
```

The midsquare method produce a good scrambling of one number to generate the next one and such a haphazard rule can make a fairly good way of generating random numbers but it's performance is not good at all.

3.1.3 Disadvantage of this Method [1]

- Strong tendency to degenerate fairly rapidly to zero,where it will stay forever.
  For example,
  Zo=1000, always zero will be generated.
- It is not random at all,in the sense of being unpredictable.
  For example,
  If we know one number, the next is completly determined since the rule to obtain it is fixed.

3.2 Linear Congruential Geneartors (LCG) [1]

Linear Congruential method was proposed by Lehmer in 1951. Now a days many random number generators use this formula.LCGs is a method that produce a sequence of pseudo-random numbers calculated with a discontinuous linear equation.This method is faster and comperatively easy to understand.

The method is defined by this recurrence relation:

$Z_i=(aZ_i\text{-}1+c)(\text{mod } m)$

Where, m=modulus

a=multiplier

c=increment

Z_0=seed or starting value

$Z_i=(aZ_{i-1}+c) \text{ mod } m$

$U_i=Z_i/m$

Where the integers m, a, c and Z_0 are nonnegative integers.

Form this equation we find the value of Z_i by making mod of $(aZ_{i-1}+c)$ by m where Z_i be the reminder and for getting the desired random number U_i on the interval [0,1] we will devide Zi by m.

The integers m, a, c and Z0 should satisfy 0<m, a<m, c<m and Z0<m for avoiding negetivity.

Two objections are arised againest LCGs.The first one is it is a psudo random number generator which is common to all. The value of Zi is not trully random at all because

every Zi is completely depens on the value of m, a, c and Z0. A psudo random number generator is also known as determinstic bit generator (DRBG).

The second objection of LCGs is that it can only generate random numbers of 0/m, 1/m, 2/m, ..., (m-1)/m these rational values.

Linear Congruential method have a looping behaviour. In LCGs the value of new seed(Zi) depends on the previous value of Zi. If the value of Zi which we take was previously taken exactly the same sequences of value will be generate and this cycle will repeats endlessly. The length of a cycle is called the period of a generator. The length of a period can be at most m, and if the period is m then it is called a full period geneartor. LCGs will be full period if and only if the following conditions hold :

a. m and c has no common divisor except 1
b. If q is a prime number, if q devides m then q devides a-1
c. If 4 devides m, then 4 devides a-1

Example of full period generator is given below,

3.2.1 Example for Linear Congruential Geneartor (LCGs): [1]

Let , m = 16, a = 9, c = 5, Z_0 = 7

step a:

m = 16, c = 9, only common divisor=1

step b:

If q = 2, m%q = 16%2 = 0

(a-1)%q = (9-1)%2 = 8%2 = 0

step c:

m%4 = 16%4 = 0

(a-1)%4 = (9-1) % 4 = 8%4 = 0

Therefore the generator is a full period.

Using this formula how this method works is shown below, $Z_i = (9Z_{i-1}+5)(\text{mod } 16)$

Table 3.2: The Linear Congruential generator where initial seed, $Z_0 = 7$

i	Z_i	U_i
0	7	_
1	4	0.25000
2	9	0.56250
3	6	0.37500
4	11	0.68750
5	8	0.50000
6	13	0.81250
7	10	0.62500
8	15	0.93750
9	12	0.75000
10	1	0.06250
11	14	0.87500
12	3	0.18750
13	0	0.00000
14	5	0.31250
15	2	0.12500
16	7	0.43750
17	4	0.25000
18	9	0.56250
19	6	0.37500
20	11	0.68750
21	8	0.50000
22	10	0.81250
.	.	.

There are some desireable property for a good LCGs, full period is one of them. The other desireable properties are good statistical properties, storage efficiency, reproducibility , seperate stream.

Now the question is how to obtained reproducibility and seperate stream is obtained? Obtaining reproducibility is simple, for that we must remember the initial seed value Z_0 then use this value again to get the sequence.

Seperate stram obtaining is also simple way. For example if we want a streams having legth 1000, then we must set Z_0 to some value for the first stream then use Z_{1000} as the seed for the second stream, then use Z_{2000} as the seed value of third stream and so on.

3.2.2 Code for Linear Congruential Geneartor (LCGs)

```
Procedure seedGenerator (int m, int a, int c, int  z)
    VAR new_z:integer
    new_z← (a*z+c) % m
    return new_z
end procedure
procedure main()
    clock_t begin, end
    double time_spent
    srand(time(NULL))
    VAR m, a, c, z, number, new_z:integer
    scan modulus m
    scan multiplier a
    scan increment c
    scan seed z
    scan random  number
    begin←clock()
```

```
for  i 1 to number
     new_z ← seedGenerator (m,a,c,z)
     z ← new_z
     float randomNumber←((float) new_z /m)
     Print i and randomNumber
 end for
end←clock()
time_spent←(double)(end-begin)/CLOCKS_PER_SEC
Print time_spent
end procedure
```

3.2.3 Classification Of LCGs [1]

1.Mixed LCGs (c>0)

2.Multiplicative LCGs (c=0)

3.2.3.1 Mixed LCGs [1]

For obtaining a large period and high density the value of m should be large. If m is large division by m is slow airthmatic operation.To solve this problem Mixed LCGs is needed. We have to take the advantage of integer overflow

For obtaining the advantage of integer overflow we must take $m = 2^b$ which helps us to avoid explicit division by m. The choice of $m = 2^b$ says that we can get a full period if a-1 is divisible by 4 and c is odd, and Z_w can be any integer between 0 and m-1 which doesnot affects the period.

Procedure of Mixed LCGs

Step 1:

Take $m = 2^b$

Where the largest integer is 2^b-1 and word size is b bits.

Step 2:

Larger integer W

Where W contains h bits and h>b assume

Step 3:

Overflow bits = h-b

Step 4:

W (mod 2^b) = Integer discarding (h-b) left most overflow bits

Example for Mixed LCGs:

m = 16 = 2^4 , 4-bit computer, here b = 4 bit

$Z_7 = (a * Z_6 + c) (\text{mod } m)$

$=(9 * Z_6 + 5) (\text{mod } 2^4)$

$=(9 * 13 + 5) (\text{mod } 2^4)$

$= 122 \text{ mod } 2^4$

$= (1111010)_2 \text{ mod } 2^4$

$= (1010)_2$

$= 10$

choice of a:

$Z_i = (aZ_{i-1} + c)(\text{mod } m)$

if $a = 2^l + 1$

$aZ_{i-1} = (2^l + 1)Z_{i-1}$

$= 2^l * Z_{i-1} + Z_{i-1}$

if a =9 , $a = 8 + 1$

$= 2^3 + 1$

$z_6 = 13$

$z_7 = (aZ_6 + c)(\text{mod } 16)$

Here,

$aZ_6 + c = 9 * 13 + 5$

$= ((2^3 + 1) * 13) + 5$

$=(2^3 * 13 + 13) + 5$

$=(2^3 * (1101)_2 + 13) + 5$

$=((1101000)_2 + 13) + 5$

⇩

only bit shifting operation is needed(No multiplication)

Code for Mixed Linear Congruential Geneartors

```
procedure convertBinaryToDecimal(int n)
    VAR remainder:integer
    set decimalNumber←i←0
    while n ≠ 0 do
        remainder ← n mod 10
        n divisible by 10
        decimalNumber ← decimalNumber + remainder * 2^i
        increment i
    end while
    return decimalNumber
end procedure
 procedure convert(int dec)
    if dec is equal to 0 then
       return 0
    else
      return (dec %2+10*convert(dec/2))
    end if
end procedure
procedure seed_generator (int m,int a,int c,int z,int b)
    VAR incre,con_z,con_zLen,seeed_len,seeed, multiperr,multiperrr:integer
    set len←array_len← i ←j←remainder ←summ←0
    VAR sum[100],ss[100],zero_array[100],zero_arrayy[100],seed_array[100]:integer
    con_z←convert(z)
```

```
if con_z≠ 0 then
    con_zLen← 1 + (int)log10(con_z)
else
   con_zLen←1
end if
Incre←convert(c)
new_z_len←con_zLen
 i←0
While con_zLen>0 do
     digit←con_z mod 10
     con_z←con_z divisible by 10
     seed_array[i]←digit
     Increment i
     Decrease con_zLen
end while
for  i 0 to new_z_len-1
   If seed_array[i] equal to 1 then
       multiper ← a << i
          Summ←summ + multiper
    end if
end for
multiperrr ← convert(summ)
  d←0
```

```
while multiperrr ≠ 0 or incre ≠ 0
        sum[d++] ←(multiperrr % 10 + incre % 10+ remainder) % 2
        remainder ←(multiperrr % 10 + incre%10+remainder) / 2
        multiperrr ←multiperrr/ 10
        Incre←incre/10
end while
if remainder ≠ 0 then
       sum[d++] ←remainder
       Decrement d
end if
while d >= 0 do
         ss[j]←sum[d--]
         Increment j
         Increment len
end while
q←0
For k 0 to len-1
         q←10*q+ss[k]
end for
zero_array[0]←1
for i 1 to len-1
     zero_array[i]←0
      Increment array_len
```

```
end for
 p←0
For k 0 to array_len
        P←10*p+zero_array[k]
end for
 If len>b then
      seeed ← q%p
//count the length of integer
       seeed_len ← 1 + (int)log10(seeed)
        While seeed_len>b do
            zero_arrayy[0]←1
             for i 1 to seeed_len-1
                 zero_arrayy[i]←0
             end for
           p←0
             for k 0 to seeed_len-1
                 P←10*p+zero_arrayy[k]
           end for
          seeed ← seeed % p
          Decrement seeed_len
        End while
      binTOdec ←convertBinaryToDecimal(seeed)
      return binTOdec
```

```
    else
        binTOdec← convertBinaryToDecimal(q)
        return binTOdec
    end if
end procedure
procedure main()
    clock_t begin, end
    double time_spent
    srand(time(NULL))
    m,a,b,c,z,number,new_z
      scan b
      m ← 2^b
      scan multiplier a
      scan increment c
      scan seed z
     scan random  number
      begin←clock()
    if c>0 then
        for  i 1 to number
                new_z ← seed_generator (m,a,c,z,b)
                z ← new_z
                 float randomNumber←((float) new_z /m)
                 Print i and randomNumber
```

```
        end for
    else
        print "condition is not satisfied"
    end if
    end←clock()
    time_spent←(double)(end-begin)/CLOCKS_PER_SEC
    Print time_spent
end procedure
```

3.2.3.2 Multiplicative LCGs [1]

In multiplicative LCGs the addition of c is not needed. It have no full period because it couldnot satisfy the condition, m and c has no common divisor except 1.

For example, m is positive and can devides both m and c = 0, but we can obtain the period m-1 if we choose m and c carefully.

Example for Multiplicative LCGs:

$m = 2^b$

$b = 4$

$m = 2^4 = 16$

$a = 5$

$z_0 = 7$

$Z_i = (5 * Z_{i-1} + 0)(\text{mod } 16)$ with $c = 0$

Table 3.3: The Multiplicative LCGs where initial seed, $Z_0 = 7$

i	Z_i	U_i
0	7	_
1	3	0.187500
2	15	0.937500
3	11	0.687500
4	7	0.437500

5	3	0.187500
6	15	0.937500
.	.	.
.	.	.

Code for Multiplicative LCGs

```
procedure convertBinaryToDecimal(int n)
    VAR remainder:integer
    set decimalNumber← i←0
    while n ≠ 0 do
        remainder ← n mod 10
        n divisible by 10
        decimalNumber ← decimalNumber + remainder * 2^i
        increment i
    end while
    return decimalNumber
end procedure
 procedure convert(int dec)
        if dec is equal to 0 then
                return 0
        else
                return (dec %2+10*convert(dec/2))
        end if
end procedure
procedure seedgenerator (int m,int a,int c,int z,int b)
    VAR incre,con_z,con_zLen,seeed_len,seeed, multiperr,multiperrr:integer
    set len←array_len← i ← j←remainder ←summ←0
    VAR sum[100],ss[100],zero_array[100],zero_arrayy[100],seed_array[100]:integer
    con_z←convert(z)
    if con_z≠ 0 then
```

```
    con_zLen← 1 + (int)log10(con_z)
else
    zzcon_zLen←1
end if
Incre←convert(c)
new_z_len←con_zLen
 i←0
While con_zLen>0 do
     digit←con_z mod 10
     con_z←con_z divisible by 10
     seed_array[i]←digit
     Increment i
     Decrease con_zLen
end while
for  i 0 to new_z_len-1
     If seed_array[i] equal to 1 then
          multiper ← a << i
          Summ←summ + multiper
    end if
end for
multiperrr ← convert(summ)
d←0
 while multiperrr ≠  0 or  incre  ≠ 0
        sum[d++] ←(multiperrr % 10  + incre % 10+ remainder) % 2
        remainder ←(multiperrr % 10  + incre%10+remainder) / 2
       multiperrr ←multiperrr/ 10
       Incre←incre/10
end while
 if  remainder  ≠  0 then
      sum[d++] ←remainder
      Decrement d
end if
 while d >= 0 do
```

```
            ss[j]←sum[d--]
            Increment j
            Increment len
end while
 q←0
 For  k 0 to len-1
          q←10*q+ss[k]
end for
zero_array[0]←1
for i 1 to len-1
      zero_array[i]←0
       Increment array_len
end for
p←0
For k 0 to array_len
        P←10*p+zero_array[k]
end for
 If len>b then
       seeed ← q%p
//count the length of integer
        seeed_len ← 1 + (int)log10(seeed)
         While seeed_len>b do
                zero_arrayy[0]←1
                for i 1 to seeed_len-1
                     zero_arrayy[i]←0
                end for
                p←0
                for k 0 to seeed_len-1
                    P←10*p+zero_arrayy[k]
                end for
                seeed ← seeed % p
                Decrement seeed_len
           End while
```

```
            binTOdec ←convertBinaryToDecimal(seeed)
            return binTOdec
        else
            binTOdec← convertBinaryToDecimal(q)
            return binTOdec
        end if
end procedure
procedure main()
        clock_t begin, end
        double time_spent
        srand(time(NULL))
        VAR  m,a,b,c,z,number,new_z:integer
        scan b
        m ← 2^b
        scan multiplier a
        scan increment c
        scan seed z
        scan random  number
        begin←clock()
        if c==0 then
            for  i 1 to number
                    new_z ← seedgenerator (m,a,c,z,b)
                    z ← new_z
                    float randomNumber←((float) new_z /m)
                    Print i and randomNumber
            end for
        else
             print “condition is not satisfied”
        end if
            end←clock()

        time_spent←(double)(end-begin)/CLOCKS_PER_SEC

        Print time_spent
```

end procedure

3.3 Prime modulus multiplicative LCGs (PMMLCGs) [1]

Period will be m-1 if it satisfy the condition, l = m-1 where $a^l - 1$ is divisible by m.

Here Z_i can be any integer from 1 to m-1 these are known as prime modulus multiplicative LCGs.

Example of one m-1 periodic generator is given bellow,

3.3.1 Example for Prime modulus multiplicative LCGs (PMMLCGs) [1]

Let, m = 3

c = 0

l = m-1

= 3 – 1

= 2

if a = 2

$a^l - 1 = a^2 - 1$

$= 2^2 - 1$

= 3

$a^l - 1 / m = 3 \% 3 = 0$

$Z_i = (a * Z_{i-1} + c) \pmod m$

$= (2 * Z_{i-1} + 0) (\mod 3)$

Where $Z_0 = 2$

Therefore the generator have m-1 period

Table 3.4: Prime modulus multiplicative LCGs where initial seed, $Z_0 = 2$

i	Z_i	U_i
0	2	_
1	1	0.333333
2	2	0.666667
3	1	0.333333
4	2	0.666667

.	.	.
.	.	.

3.3.2 Code for Prime Modulus Multiplicative LCGs

```
procedure seedGenerator (int m, int a, int c, int  z)
    VAR  new_z:integer
    new_z← (a*z+c) % m
    return new_z
end procedure
procedure main()
     VAR clock_t begin, end:integer
     VAR time_spent: double
     srand(time(NULL))
     VAR m,a,c,z,number,new_z,l,power_of_a,power_a:integer
     scan modulus m
     scan multiplier a
     scan increment c
     scan number of random number number
     l ←m-1
     power_a←pow(a,l)
     power_of_a←pow(a,l)-1
     begin←clock()
     if((power_of_a%m)==0)
          for  i 1 to number
               new_z ← seedGenerator (m,a,0,z)
              z ← new_z
              float randomNumber←((float) new_z /m)
              Print i and randomNumber
         end for
     else
          print “condition is not satisfied”
     end if
     end←clock()
```

time_spent←(double)(end-begin)/CLOCKS_PER_SEC

Print time_spent

end procedure

To optimize runtime we can use bit shifting algorithm of Prime Modulus Multiplicative generator which decreases the runtime. We use it for avoid multiplication and division and thus we can get better performance.

Like mixed generators here it is also efficient of taking $m = 2^b$ for obtaining the adavntages of integer overflow by which we can avoid explicit division.

Example of one m-1 periodic generator is given bellow,

3.3.3 Example for Prime Modulus Multiplicative LCGs with bit shifting

$m = 2^b - q$

To get, $Z_i = (aZ_{i-1})(\text{mod } 2^b - q)$

Let $Z_i' = (aZ_{i-1})(\text{mod } 2^b)$

If k is largest integer less than or equal to $aZ_{i-1}/2^b$

$Z_i = Z_i' + kq$ if $Z_i' + kq < 2^b - q$ .....(i)

Let $a = 2$, $m = 3 = 2^2 - 1 = 2^b - q$

Therefore $q = 2$, $b = 2$

$i = 1$ $Z_1' = (2 * Z_0)(\text{mod } 2^2)$

$= (2 * 2) \text{ mod } 2^2$

$= 4 \text{ mod } 2^2$

$= (100)_2 \text{ mod } 2^2$

$= (00)_2$

$= 0$

$k = (aZ_{i-1})/2^b$

$=(2*Z_0)/2^2$

$=(2 * 2)/2^2 = (100)_2/2^2 = (001)_2$

$=1$

Here we use bit shifting instead of multiplecation and division.

if i=1, $Z_i' + kq = Z_1' + kq$

$= 0 + 1 * 1$

$= 1$

$Z_1 = 1$ Since $1 < 2^b - q$

3.3.4 Code for Prime Modulus Multiplicative LCGs with bit shifting

```
procedure convertBinaryToDecimal(int n)
    VAR remainder:integer
    set decimalNumber← i←0
    while n ≠ 0 do
        remainder ← n mod 10
        n divisible by 10
        decimalNumber ← decimalNumber + remainder * 2^i
        increment i
    end while
    return decimalNumber
end procedure
procedure convert(int dec)
    if dec is equal to 0 then
       return 0
    else
      return (dec %2+10*convert(dec/2))
    end if
end procedure
procedure newSeedGenerator (int new_z,int k,int q,int b)
    VAR z_i:integer

    z_i← new_z +(k*q)

    VAR check:integer

    pow(2,b)-q

    if(z_i<check)

          return z_i
```

```
    else
        print "condition is not satisfied"
    end if
    end procedure
procedure  zprimeGenerator (int a,int c,int z,int b)
    VAR incre,con_z,con_zLen,seeed_len,seeed, multiperr,multiperrr:integer
    set len←array_len←i← j←remainder ←summ←0
    VAR  sum[100],ss[100],zero_array[100],zero_arrayy[100],seed_array[100]:integer
    con_z←convert(z)
    if con_z≠ 0 then
        con_zLen← 1 + (int)log10(con_z)
    else
        con_zLen←1
    end if
    Incre←convert(c)
    new_z_len←con_zLen
    i←0
    While con_zLen>0 do
        digit←con_z mod 10
        con_z←con_z divisible by 10
        seed_array[i]←digit
        Increment i
        Decrease con_zLen
    end while
    for  i 0 to new_z_len-1
        If seed_array[i] equal to 1 then
            multiper ← a << i
            Summ←summ + multiper
        end if
    end for
    multiperrr ← convert(summ)
```

```
d←0
 while multiperrr ≠ 0 or  incre ≠ 0
        sum[d++] ←(multiperrr % 10  + incre % 10+ remainder) % 2
        remainder ←(multiperrr % 10  + incre%10+remainder) / 2
        multiperrr ←multiperrr/ 10
       Incre←incre/10
end while
 if  remainder ≠ 0 then
      sum[d++] ←remainder
      Decrement d
end if
 while d >= 0 do
         ss[j]←sum[d--]
        Increment j
       Increment len
  end while
 q←0
 For  k 0 to len-1
      q←10*q+ss[k]
end for
zero_array[0]←1
 for i 1 to len-1
     zero_array[i]←0
      Increment array_len
end for
     p←0
For k 0 to array_len
       P←10*p+zero_array[k]
end for
 If len>b then
      seeed ← q%p
//count the length of integer
       seeed_len ← 1 + (int)log10(seeed)
```

```
        While seeed_len>b do
            zero_arrayy[0]←1
             for i 1 to seeed_len-1
                 zero_arrayy[i]←0
             end for
            p←0
             for k 0 to seeed_len-1
                 P←10*p+zero_arrayy[k]
             end for
             seeed ← seeed % p
             Decrement seeed_len
        end while
        binTOdec ←convertBinaryToDecimal(seeed)
        return binTOdec
    else
       binTOdec← convertBinaryToDecimal(q)
        return binTOdec
     end if
end procedure
procedure  k_Generator (int a,int z,int b)
    VAR multi_a,con_z,con_zLen,multiperr,multiperrr:integer
    set i ← j←summ←0
    VAR  seed_array[100]:integer
    con_z←convert(z)
    if con_z≠ 0 then
      con_zLen← 1 + (int)log10(con_z)
    else
      con_zLen←1
    end if
```

```
    multi_a←convert(a)
    new_z_len←con_zLen
    new_z_len←con_zLen
    Initialize i
    While con_zLen>0 do
       digit←con_z mod 10
        con_z←con_z divisible by 10
        seed_array[i]←digit
       Increment i
       Decrease con_zLen
    end while
    for i 0 to new_z_len-1
        if seed_array [i]equal to1
              Declare multiper← a << i
              summ←summ + multiper
       end if
    end for
    division_result_k ← summ >> b
     return division_result_k
End procedure
procedure main()
     clock_t begin, end
     VAR time_spent: double
     srand(time(NULL))
```

```
 VAR m,a,b,q,c,z,number,new_z,l,power_of_a,new_z1,kgenerator:integer
scan b
m ← 2^b
q←1
m ←pow(2,b)-q
l ←m-1
scan multiplier a
scan seed z
scan number of random  number number
power_of_a←pow(a,l)-1
begin←clock()
    if power_of_a%m==0 then
      for  i 1 to number
          kkgenerator←k_Generator(a,z,b)
          new_z ← zprimeGenerator(a,0,z,b)
          new_z1←newSeedGenerator(new_z,kkgenerator,q,b)
          z ← new_z1
          random_number← ((float)new_z1/m)
          Print i and randomNumber
      end for
   else
       print ”condition is not satisfied”
   end if
end←clock()
time_spent←(double)(end-begin)/CLOCKS_PER_SEC
Print time_spent
end procedure
```

3.4 Quadratic Congruential Generator [1]

Quadratic congruential generator is one of the alternatives to linear congruential pseudorandom number generator.

As a special case of generators LCGs can be written by

$$Z_i = g(Z_{i-1}, Z_{i-2},\ldots\ldots)(\text{mod } m) \ldots\ldots\ldots (i)$$

Here g is a fixed deterministic function of previous Z_j.

Another apparent generalization of LCGs is

$$g(Z_{i-1}, Z_{i-2},\ldots\ldots) = a'Z_{i-1}^2 + aZ_{i-1} + c \ldots\ldots\ldots (ii)$$

this produces a quadratic congruential generator.

Special case $a_1 = a = 1$

$C = 0$

m = power of 2

From (i) and (ii)

$Z_i = (a_1Z_{i-1}^2 + aZ_{i-1} + c) \pmod m$ is generated as a new formula. We can generate a sequence of $Z_1, Z_2,\ldots..$ using this formula.

Let, $a_1 = a = 1, c = 0, m = 2^4$

Table 3.5: Quadratic congruential generator where initial seed, $Z_0 = 5$

i	Z_i	$U_i = Z_i/m$
0	5	-
1	14	0.875
2	2	0.125
3	6	0.375
4	10	0.625
.	.	.

3.4.1 Code for Quadratic Congruential Generator

```
procedure  seed_generator(int m,int a1,int a,int c,int z)
        new_z=((a1*z*z)+(a*z)+c)  mod  m

        return new_z

end procedure

procedure main()
       clock_t begin, end
       VAR time_spent: double
       srand(time(NULL))
       VAR m,a1,a,b,c,z,number,new_z:integer
       a1← a← 1
```

```
        scan b
        m ← 2^b
        scan seed z
        scan random  number
        begin←clock()
        for  i 1 to number
             new_z← seed_generator(m,a1,a,0,z,b)
              z ←new_z
            float randomNumber←((float) new_z /m)
            Print i and randomNumber
        end for
       end←clock()
        time_spent←(double)(end-begin)/CLOCKS_PER_SEC
        Print time_spent
end procedure
```

Although quadratic congruential generator is similar to Midsquare Method, it has better statistical properties. Because Z_i depends only on Z_{i-1}, not on earlier Z_j's. Every seed value generates between 0 and m-1 and the period of quadratic congruential generator is at most m.

To optimize runtime we can use bit shifting algorithm of quadratic congruential generator which decreases the runtime. We use it for avoid multiplication and division and thus we can get better performance.

3.4.2 Code for Quadratic Congruential Generator Bit Shifting Algorithm

```
procedure convertBinaryToDecimal(int n)
    VAR remainder:integer
    set decimalNumber← i←0
    while n ≠ 0 do
        remainder ← n mod 10
        n divisible by 10
        decimalNumber ← decimalNumber + remainder * 2^i
        increment i
```

```
    end while
    return decimalNumber
end procedure
 procedure convert(int dec)
     if dec is equal to 0 then
         return 0
    else
          return (dec %2+10*convert(dec/2))
    end if
end procedure
procedure seed_generator (int m,int a1,int a,int c,int z,int b)
     VAR incre,con_z,con_z1,con_zLen,con_zLen1,seeed_len,seeed,multiperr,multiperrr,
      multiperr1, multiperrr1:integer
     set len←array_len←summ←summ1←i ←j←remainder←0
     VAR sum[100],ss[100],zero_array[100],zero_arrayy[100],
     seed_array[100],seed_array1[100]:integer
     con_z1←convert(z*z)
     if con_z1≠ 0 then
         con_zLen1← 1 + (int)log10(con_z1)
     else
          con_zLen1←1
     end if
     con_z←convert(z)
     if con_z≠ 0 then
          con_zLen← 1 + (int)log10(con_z)
     else
          con_zLen←1
     end if
     Incre←convert(c)
     new_z_len1←con_zLen1
     i←0
     While con_zLen1>0 do
          digit←con_z 1mod 10
```

```
        con_z1←con_z 1divisible by 10
        seed_array1[i]←digit
        Increment i
        Decrease con_zLen1
end while
for  i 0 to new_z_len1-1
     If seed_array1[i] equal to 1 then
          multiper 1← a << i
         Summ1←summ 1+ multiper1
   end if
end for
multiperrr1 ← convert(summ1)
new_z_len←con_zLen
i←0
While con_zLen>0 do
    digit←con_z mod 10
    con_z←con_z divisible by 10
    seed_array[i]←digit
    Increment i
    Decrease con_zLen
end while
for  i 0 to new_z_len-1
      If seed_array[i] equal to 1 then
           multiper ← a << i
           Summ←summ + multiper
       end if
end for
multiperrr ← convert(summ)
d←0
 while multiperrr1≠0 || multiperrr≠ 0 || incre≠0
     sum[d++]  ← (multiperrr1 % 10  + multiperrr % 10  + incre % 10+ remainder)
%2
     remainder  ← (multiperrr1 % 10  + multiperrr % 10  + incre%10+remainder) /2
```

```
        multiperrr1  ← multiperrr1/ 10
        multiperrr  ←  multiperrr/ 10
        incre ← incre/10
  end while
   if  remainder  ≠  0 then
         sum[d++] ←remainder
         Decrement d
 end if
 while d >= 0 do
        ss[j]←sum[d--]
        Increment j
         Increment len
end while
 q←0
 For  k 0 to len-1
      q←10*q+ss[k]
 end for
 zero_array[0]←1
 for i 1 to len-1
       zero_array[i]←0
        Increment array_len
 end for
  p←0
  For k 0 to array_len
         P←10*p+zero_array[k]
  end for

   If len>b then

        seeed ← q%p

  //count the length of integer
        seeed_len ← 1 + (int)log10(seeed)

         While seeed_len>b do
```

```
                zero_arrayy[0]←1
                for i 1 to seeed_len-1
                    zero_arrayy[i]←0
                end for
                p←0
                for k 0 to seeed_len-1
                    P←10*p+zero_arrayy[k]
                end for
                seeed ← seeed % p
                Decrement seeed_len
            End while
            binTOdec ←convertBinaryToDecimal(seeed)
            return binTOdec
        else
            binTOdec← convertBinaryToDecimal(q)
            return binTOdec
        end if
end procedure
procedure main()
        clock_t begin, end
        VAR time_spent: double
        srand(time(NULL))
        VAR m,a1,a,b,c,z,number,new_z:integer
        a1← a← 1
        scan b
```

```
m ← 2^b
scan seed z
scan random  number
begin←clock()
for  i 1 to number
     new_z← seed_generator(m,a1,a,0,z,b)
      z ←new_z
     float randomNumber←((float) new_z /m)
     Print i and randomNumber
end for
end←clock()
time_spent←(double)(end-begin)/CLOCKS_PER_SEC
Print time_spent
end procedure
```

3.5 Fibonacci Generator [1]

As a special case of generators LCGs can be written by

$$Z_i = g(Z_{i-1}, Z_{i-2},\ldots\ldots)(\text{mod } m) \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\text{(i)}$$

Multiple recursive generators are defined by

$$g(Z_{i-1}, Z_{i-2},\ldots\ldots) = a_1Z_{i-1} + a_2Z_{i-2} + \ldots\ldots\ldots + a_qZ_{i-q} \quad \ldots\ldots\ldots\ldots\text{(ii)}$$

Focused on g's in equation (i) and (ii) we can define as $Z_{i-1} + Z_{i-q}$ which have contained Fibonacci Generator as follows

$$Z_i = (Z_{i-1} + Z_{i-2})(\text{mod } m) \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\text{(iii)}$$

Let, $Z_{-1} = 5$, m=16

Table 3.6: Fibonacci Generator where initial seed, $Z_0 = 7$

i	Z_{i-1}	Z_i	$U_i = Z_i/m$
0	5	7	-
1	7	12	0.75
2	12	3	0.1875

3	3	15	0.937
4	15	2	0.125
.	.	.	.
.	.	.	.

The period of fibonacci generator is excess of m,but it is completely unacceptable from a statistical standpoint

3.5.1 Code for Fibonacci Generator

```
procedure seed_generator(int m,int z1,int z2)
      new_z ←( (z1+z2) mod m

     return new_z

     end procedure

procedure main()
     clock_t begin, end
     VAR time_spent: double
      srand(time(NULL))
     VAR m,z1,z2,number,new_z:integer
     scan modulus m
     scan seed1 z1
     scan seed2 z2
     scan random  number
     begin←clock()
      for  i 1 to number
            new_z ←seed_generator(m,z1,z2)
             z1←z2
             z2 ← new_z
            float randomNumber←((float) new_z /m)
            Print i and randomNumber
      end for
```

End←clock()

time_spent←(double)(end-begin)/CLOCKS_PER_SEC

Print time_spent

end procedure

3.6 Composite generators [1]

Composite generators come from the combination of separate two or more generators. The composite generators give better statistical behavior and longer period than earlier separate generators. But composite generators have some limitations that the cost of obtaining U_i is high. On the other side, simple generator takes less cost of obtaining each U_i.

MacLaren and Marsaglia in 1965 developed the earliest composite generators that shuffle the result of first LCG and used in second LCG. Later this process is extended by Marsaglia and Bray in 1968, Grosenbaugh in 1969, Nance and Overstreet in 1975.

When we want to extend period of a generator by breaking up any correlation, shuffling has this type of natural inherent appeal. MacLaren and Marsaglia obtained shuffling generators which behave like a better statistical generator than the two individual LCGs. In 1978 Nance and Overstreet discovered that if we shuffling two bad LCGs with one another and extend the period it may result in a good composite generator. Despite of all these advantages one cannot jump shuffled generators output sequence into an arbitrary point, all the intermediate values should be generated. But in LCGs sequences can be broken.

Let, the m be the modulus and the sequence is…..

$\{Z_{1,i}\},\{Z_{2,i}\},\ldots\ldots\ldots,\{Z_{j,i}\}$

So the final equation is,

$Y_i = (\delta_1 Z_{1,i}+\delta_2 Z_{2,i}+\ldots\ldots\ldots+\delta_j Z_{j,i})(\text{mod } m)$

$U_i = Y_i / m$

3.6.1 Example for Composite generators

Let, modulus m = 16

Seed1=5, Seed2=7, Seed3=3, Seed4=4

Delta1 = 5000041, Delta2 = 5018467, Delta3 = 5006334, Delta4 = 5026500

$Z_1 = (\delta_1 Z_{1,i-2} - \delta_2 Z_{1,i-3}) (\text{mod } m)$

$= (5000041 * 5 – 5018467 * 7) (\text{mod } 2^4) = 8$

$Z_2 = (\delta_3 Z_{1,i-1} – \delta_4 Z_{1,i-3}) (\text{mod } m)$

$= (5006334 * 3 – 5016500 * 4)(\text{mod } 2^4) = 6$

Seed value is,

$Y_1 = (Z_1 - Z_2)(\text{mod } m) = 2$

The random number is,

$U_i = Y_i/m = 2/16 = 0.125000$

In 1982 Wichmann and Hill proposed an idea to combining three generators and it have the behavior of long period, speed, portability and usability on small computers.

3.6.2 Code for Composite generators

```
procedure random_number(int m,int z1,int z2)
        VAR delta1,delta2:integer
        delta1 ← rand() % 10000000 + 5000000
        delta2 ← rand() % 10000000 + 5000000
        new_z←abs(delta1*z1-delta2*z2) % m
        return new_z
end procedure
procedure random_number_prime(int m,int z3,int z4)
        VAR delta3,delta4:integer
        delta3 ← rand() % 10000000 + 5000000
        delta4 ← rand() % 10000000 + 5000000
        new_z_prime←abs(delta3*z3-delta4*z4) % m
        return new_z_prime
end procedure
procedure main()
        clock_t begin, end
```

```
VAR time_spent: double
 srand(time(NULL))
VAR m,z1,z2,z3,z4,number,new_z,y,u,new_z_prime:integer
scan  modulus m
scan seed1 z1
scan seed2 z2
 scan seed3 z3
scan  seed4 z4
 scan  random number
 begin←clock()
for  i 1 to number
          new_z ← random_number(m,z1,z2)
          z1←z2
          z2 ← new_z
          new_z_prime ← random_number_prime(m,z3,z4)
         z3←z4
          z4 ← new_z_prime
          y←abs(new_z-new_z_prime)
          float random_number←((float)y/m)
          print i and random_number
end for
end←clock()
time_spent←(double)(end-begin)/CLOCKS_PER_SEC
```

Print time_spent

end procedure

If i^{th} random numbers are U_{1i},U_{2i} and U_{3i} produced by three different generators.

The equation is,

$U_i = U_{1i}+U_{2i}+U_{3i}$

Later it found by McLeod in 1985 that in some computer architectures these generators may have numerical difficulties. In 1986 Zeisel showed this generator is similar to multiplicative LCGs with very large modulus and multiplier.

3.7 Tausworthe and Related Generators [1]

In 1965 based on a paper by Tausworthe various kinds of interesting generators have been developed that are related to cryptographic methods. To form random number the cryptographic methods are work without deviation on bits.

Let a sequence of binary digits $b_1, b_2,$…….

$b_i = (c_1b_{i-1} + c_2b_{i-2} +$……$+ c_qb_{i-q}) \pmod 2$

where $c_1 , c_2,$……., c_q are constant either 0 or 1 and $c_q =1$. The maximum period is 2^q-1.

Application of Tausworthe Generators,

$b_i = (b_{i-r} + b_{i-q}) \pmod 2$ where $0<r<q$

using X-OR,

$b_i = 0$ if $b_{i-r} = b_{i-q}$ and

$b_i = 1$ if $b_{i-r} \neq b_{i-q}$

Which is written by $b_i = b_{i-r} \oplus b_{i-q}$ and this equation is use to initialize the sequence of b_i.

3.7.1 Example for Tausworthe and Related Generators

Let, r = 5, q = 7 and $b_1 = b_2 =$…….$= b_5 = 1$

The first 45 period bit sequence is give below,

111111100000110001111011001011110010011101 11

The maximum period 2^7-1 = 127 so let l=6.

Then the l bit binary strings are-

$(111111)_2 = (63)_{10}$ $U_1 = 63/2^6 = 63/64 =0.984375$

$(100000)_2 = (32)_{10}$ $U_1 = 32/2^6 = 32/64 = 0.5000$

$(110001)_2 = (49)_{10}$ $U_1 = 49/2^6 = 49/64 = 0.765625$

$(111011)_2 = (59)_{10}$ $U_1 = 59/2^6 = 59/64 = 0.921875$

$(001011)_2 = (11)_{10}$ $U_1 = 11/2^6 = 11/64 = 0.171875$

$(110010)_2 = (50)_{10}$ $U_1 = 50/2^6 = 50/64 = 0.781250$

$(011101)_2 = (29)_{10}$ $U_1 = 29/2^6 = 29/64 = 0.453125$

From the observation it indicate the obtaining b_i's are being used as a source of U(0,1) random numbers.

3.7.2 Code for Tausworthe and Related Generators

```
procedure convertBinaryToDecimal(int n)
    VAR remainder:integer
    set decimalNumber← i←0
    while n≠0 do
        remainder ← n%10
        n ← n/ 10
        decimalNumber ← decimalNumber +remainder*pow(2,i)
         increment  i
    end while
    return decimalNumber
end procedure
procedure main()
    clock_t begin, end
    VAR time_spent: double
    srand(time(NULL))
    VAR  i,r,q,l, n:integer
```

```
VAR b[1000],a[1000]:integer
i←6
 b[1]←b[2]←b[3]←b[4]←b[5]←1
 scan  r,q,l,n
begin←clock()
 for i 6 to n
     b[i]←(b[i-r]+b[i-q])% 2
end for
for i 1 to n
     print b[i]
end for
j←1
 k←l
s←1
p←0
 if l<=q then
     for  x 1 to n/l
               for i j to k*s
                         a[i] ←b[i]
               end for
               for i j to k*s
                         p←10*p+a[i]
               end for
```

```
                des ← convertBinaryToDecimal(p)
               float random_Number←(float)des/pow(2,l)
                print x and random_Number
                p←0
                j←(k*s)+1
                  s←s+1
        end for
    else
          Print "condition is not satisfied"
     end if
     end←clock()
      time_spent←(double)(end-begin)/CLOCKS_PER_SEC
      print time_spent
end procedure
```

The recurrence of Tausworth can be used to implement on a binary computer by using a switching circuit, that circuit is called a linear feedback shift register (LFSR). The Tausworth equation and linear feedback shift register use the same recurrence so we can call the Tausworth generator as LFSR generators.

Matsumoto and Kurita invented in 1996 and later in 1995 Tezuka invented that LFSR generators have statistical deficiencies. L'Ecuyer discovered a combined LFSR generator which gives better statistical properties and larger period.

The modified version of LFSR had invented in 1973 by Lewis and Payne that is called generalized feedback shift register (GFSR) generator.

3.8 Blum, Blum and Shub [1]

In Tausworthe generators the main element is a sequence of bits generate. Using bits generator a different method with applications to cryptography was proposed by Blum, Blum and Shub in 1986.

Let p and q prime numbers where (p-3) and (q-3) each divisible by 4.

The modulus is m = pq

From quadratic congruential recurrence we know,

$X_i = X^2_{i-1} \pmod m$

Where we get the sequence of integers $\{X_i\}$

The bit sequence of b_i means parity of rightmost bit of X_i . When X_i is even $b_i = 0$ and otherwise $b_i = 1$.

Blum, Blum and Shub in 1986 found that invention of nonrandomness of bit sequence is basically unpredictable. So it is computationally equivalent to factoring m into pq.

3.8.1 Example for Blum, Blum and Shub

Let, p = 7, q = 11

m = p*q = 77

$X_o = 9$

So, $X_1 = 9^2 \mod 77 = 4 = (100)_2$

$b_1 = 0$ since X_1 = even

$X_2 = 4^2 \mod 77 = 16 = (10000)_2$

$b_2 = 0$ since X_2 = even

$X_3 = 16^2 \mod 77 = 25 = (11001)_2$

$b_3 = 1$ since X_3 = odd

the obtaining b's are,

001100110…….

Here m = 77 as we can represent 77 in binary 1001101 has 7 digits so we take 7 bits then the random number is,

$(0011001)_2 = (25)_{10}$

$U_i = 25/m = 25/77 = 0.32467$

3.8.2 Code for Blum, Blum and Shub

```
procedure convertBinaryToDecimal(int n)
        set decimalNumber← i←0
        VAR remainder:integer
        while n ≠ 0 do
                remainder ← n%10
                n ←n/ 10
                decimalNumber ← decimalNumber +remainder*2^i
                increment  i
        end while
        return decimalNumber
  end procedure
  procedure convert(int dec)
        if dec equal to 0 then
           return 0
        else
            return (dec % 2 + 10 * convert(dec/2))
        end if
  end procedure
  procedure  x_generator(int x,int m)
        new_x←(x*x) % m
        return new_x
 end procedure
  procedure main()
```

```
clock_t begin, end

VAR time_spent: double

srand(time(NULL))

VAR i,p,q, m,x,new_x,n,len_bin_new_x,new_len_bin_new_x:integer

c ←e←0

VAR  a[10000],new_lsb[10000],  new_a[10000],lsb_array[10000]:integer

s←1

scan p,q,x,n

begin←clock()

 for i 1 to p

        if p mod i equal to 0 then

                increment c

        end if

end for

 for i 1 to q

        if q mod i equal to 0 then

                increment e

        end if

  end for

 if c==2 && e==2 && (p-3)%4==0 && (q-3)%4==0 then

        m←p*q

        for i 1 to n

             new_x ← x_generator(x,m)
```

```
            x ← new_x
            bin_new_x←convert(new_x)
             len_bin_new_x← 1 + (int)log10(bin_new_x)
             new_len_bin_new_x←len_bin_new_x
             initialize j
             while len_bin_new_x>0 do
                digit←bin_new_x%10
                 bin_new_x← bin_new_x/10
                a[j]←digit
                increment j
                 decrement  len_bin_new_x
        end while
        k←0
        for i new_len_bin_new_x-1 downto 0
                 new_a[k]←a[i]
                increment k
        end for
       for j to new_len_bin_new_x-1 to new_len_bin_new_x-1
                lsb_array[s]←new_a[j]
                Increment  s
        end for
end for
m_bin←convert(m)
```

```
len_m_bin←floor(log10(abs(m_bin)))+1
l←len_m_bin
 p←0
 j←1
 s←1
 k←l
  if n%l==0 then
        for  x 1 to n/l
              for i j to k*s
                    new_lsb[i] ← lsb_array[i]
              end for
              for i j to k*s
                    p←10*p+new_lsb[i]
               end for
               des ← convertBinaryToDecimal(p)
               float random_Number←(float)des/m
               print x and random_Number
               p←0
              j←(k*s)+1
              s←s+1
       end for
   else
       print “Condition is not satisfied”
```

```
            end if
else
        print " Condition is not satisfied"
end if
end←clock()
time_spent←(double)(end-begin)/CLOCKS_PER_SEC
    print time_spent
end procedure
```

Chapter 4

Analysis and Experimental Results

By analyzing these algorithms we gather some information which helps us to differentiate the efficiency of different types of generators.

Table 4.1: Time complexity of different random number generators

Number	Name of Algorithms	Periods	Runtime
1	Midsqure Method	Short	0.515
2	Linear Congruential Generator(LCGs)	Full	0.468
3	Mixed LCGs	Full	0.460
4	Multiplicative LCGs	m-1	0.453
5	Prime Modulus Multiplicative LCGs	m-1	0.531
6	Prime Modulus Multiplicative LCGs (With bit shifting)	m-1	0.453
7	Quardetic Congruential Generator	at most m	0.593
8	Quardetic Congruential Generator (with bit shifting)	at most m	0.484
9	Fibonacci Generator	Long	0.499
10	Composite Generator	Long	0.421
11	Tausworthe and Related Generator	Long	0.686
12	Blum,Blum and Shub Method	Long	0.780

From the table we can reach in a result that we can not deffirentiate the efficiency of algorithms only based on runtime, periods.Some other test is also needed as,emperical test and theoritical test. For test Auto correlation, Independence. uniformity we use Chi-

square test. Theoritical test is sophisticated and mathematically complex so we prefer Chi-square test.

Chapter 5

Conclusion and Future works

By studying on random numbers and different types of random number generators we have learnt how to get large period, high density, optimized runtime, better statistical properties, reproducibility and portability.

We tried to fulfill these properties and implement algorithms. We also used bit shifting operations in replace of multiplications and divisions for generating more efficient algorithms.

In future we want to generate more efficient algorithms that give large period, less runtime, good statistical properties, separate streams.

We tried to give clear concepts of the algorithms that can help anyone to have a good concept on random number generators.

Chapter 6

References

1. Law A.M. , “Simulation Modeling and Analysis”, McGraw Hill, Edition 4, 2008
2. Applications of randomness: https://en.wikipedia.org/wiki/Applications_of_randomness
3. Mid square method: https://www.slideshare.net/armanhossain1/midsquare-method
4. Definition of random numbers: http://whatis.techtarget.com/definition/random-numbers
5. What does random numbers means: https://www.techopedia.com/definition/31706/random-number
6. Random numbers generations: https://en.wikipedia.org/wiki/Random_number_generation