RANDOM NUMBER GENERATION AND ITS BETTER TECHNIQUE

A thesis submitted in partial fulfillment of the requirements

for the award of degree of

Master of Engineering

in

Computer Science and Engineering

Submitted By

Prasada Rao Gurubilli

(Roll No: 800832004)

Under the supervision of

Dr. Deepak Garg

Assistant Professor

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT

THAPAR UNIVERSITY

PATIALA – 147004

June 2010

Certificate

I hereby certify that the work which is being presented in thesis entitled **"Random number generation and its better technique"**, in partial fulfillment of the requirements for the award of degree of Master of Engineering in Computer Science and Engineering Submitted in Computer Science and Engineering Department of Thapar University, Patiala, is an authentic record of my own work carried out under supervision of Dr. Deepak Garg and refers other researchers' works which are duly listed in reference section.

The matter presented in this thesis has not been submitted for the award of any other degree of this or any other university.

V. Prasada Rao

(Prasada Rao Gurubilli)

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.

Deepak Garg

(Dr. Deepak Garg)
Assistant professor
Computer Science and Engineering Department
Thapar University
Patiala.

Countersigned By

15/07/10

(Dr. Rajesh Bhatia)
Head
Computer Science and Engineering Dept,
Thapar University,
Patiala.

(Dr.R.K. Sharma) 16.7.10
Dean (Academic Affairs)
Thapar University
Patiala.

Acknowledgment

No volume of words is enough to express my gratitude towards my guide, Dr. Deepak Garg, Assistant Professor in Computer Science and Engineering Department, Thapar University, who has been very concerned and have aided for all the material essential for the preparation of this thesis report. They have helped me to explore this vast topic in an organized manner and provided me with all the ideas on how to work towards a research oriented venture.

I am also thankful to Dr. Rajesh Bhatia, Head of Department, CSED, and Dr. Inderveer Channa, P.G. Coordinator, for the motivation and inspiration that triggered me for the thesis work.

I would also like to thank the staff members and my colleagues who were always there in the need of the hour and provided with all the help and facilities, which I required, for the completion of my thesis.

I am also grateful to my friends and fellow students, who gave me their support in and outside the laboratory. They are so many and from so different countries that I cannot mention all of them. However, they know that I will always appreciate their friendship and help.

Most importantly, I would like to thank my Parents and the Almighty for showing me the right direction out of the blue, to help me stay calm in the oddest of the times and keep moving even at times when there was no hope.

G. Prasada Rao

(Prasada Rao Gurubilli)

800832004

Abstract

Random number generators based on linear recurrences modulo 2 are among the fastest long-period generators currently available. The uniformity and independence of the points they produce, over their entire period length, can be measured by theoretical figures of merit that are easy to compute, and those having good values for these figures of merit are statistically reliable in general. Some of these generators can also provide disjoint streams and substreams efficiently. In this paper, we review the most interesting construction methods for these generators, examine their theoretical and empirical properties, and make comparisons.

Random number generation is the art and science of deterministically generating a sequence of numbers that is difficult to distinguish from a true random sequence. This thesis introduces the field of random number generation, and studies three types of random number generators in depth. It also includes mathematical techniques for transforming the output of generators to arbitrary distributions, and methods of evaluating and comparing random number generators. It concludes with a summary and historical perspective on the field of random number generation. The mathematics in this thesis is drawn mainly from number theory, with a few fundamental ideas taken from probability and statistics.

Table of Contents

List of Figures

Figures:

List of Tables

Tables:

List of Abbreviations

TRNGs	True random Number Generators
QRNGs	Quasi-random Number Generators
PRNGs	Pseudorandom Number Generators
LCG	Linear Congruential Generators
FG	Fibonacci Generators
CMRG	Combined Multiple Recursive Generator
MRG	Multiple Recursive Generator
ANG	Additive Number Generator
FSRG	Feedback Shift Register Generators
FP	Floating Point
AF	Approximate Factoring
P2D	Powers-of-2 Decomposition

Chapter1

Introduction

1.1 What is a random number generator?

Most random number generators generate a sequence of integers by the following recurrence:

$$X_0 = \text{given}, \qquad X_{n+1} = a\,X_n + b \pmod N \quad n = 0,1,2,... \qquad \text{(i)}$$

The notation mod N means that the expression on the right of the equation is divided by N, and then replaced with the remainder.

To understand the mechanics consider the following simple Example. (Choose Example on the applet to study this example further.)

$$X_0 = 79,\ N = 100,\ a = 263, \text{ and } b = 71$$

Then

$$X_1 = 79*263 + 71 \pmod{100} = 20848 \pmod{100} = 48,$$

$$X_2 = 48*263 + 71 \pmod{100} = 12695 \pmod{100} = 95,$$

$$X_3 = 95*263 + 71 \pmod{100} = 25056 \pmod{100} = 56,$$

$$X_4 = 56*263 + 71 \pmod{100} = 14799 \pmod{100} = 99,$$

Subsequent numbers are: 8, 75, 96, 68, 36, 39, 28, 35, 76, 59, 88, 15, 16, 79, 48. The sequence then repeats. (This indicates a weakness of our example generator: If the random numbers are between 0 and 99 then one would like every number between 0 and 99 to be a possible member of the sequence. The parameters a, b and N determine the characteristics of the random number generator, and the choice of x_0 (the seed) determines the particular sequence of random numbers that is generated. If the generator is run with the same values of the parameters, and the same seed, it will generate a sequence that's identical to the previous one. In that sense the numbers generated certainly are not random. They are therefore sometimes referred to as pseudo random numbers.

1.2 Transformation of the original sequence

Of course one may want random numbers not as integers in a given range, but for example as uniformly distributed real numbers in a certain interval, or perhaps as real numbers of (almost) arbitrary size, but clustered around the origin. Distributions of that sort can be obtained by suitably transforming the original random numbers. For example, to transform a sequence defined as above into an evenly distributed set of real numbers in the interval from 0 to 1 simply divide each of the original numbers by N. In the remainder of this page, though, we just consider the sequence defined by (i) itself.

1.3 What makes a good random number generator?

That's a good question! Several answers are possible, for example:

- The sequence generated by (i) isn't random at all, so there is no good random number generator of that form.
- A sequence (i) is good if it passes several well established statistical tests.
- Or, it's good if it gives good results in particular applications (where of course the meaning of "good results" is heavily dependent upon the context).

The applet on this page takes a different tag. It plots a certain number of points (X_i, X_{i+k}) for certain values of k = 1,2,3,... . Intuitively, for a random sequence, one should obtain a set of points distributed "evenly", "randomly" or "uniformly" over a square. It is not easy to make these concepts precise, but it is sometimes glaringly apparent when a set of points is not distributed in this way. Plotting 100 points with k=1 for our example generator above generates the picture nearby. (It's is shown here at half its original size.)

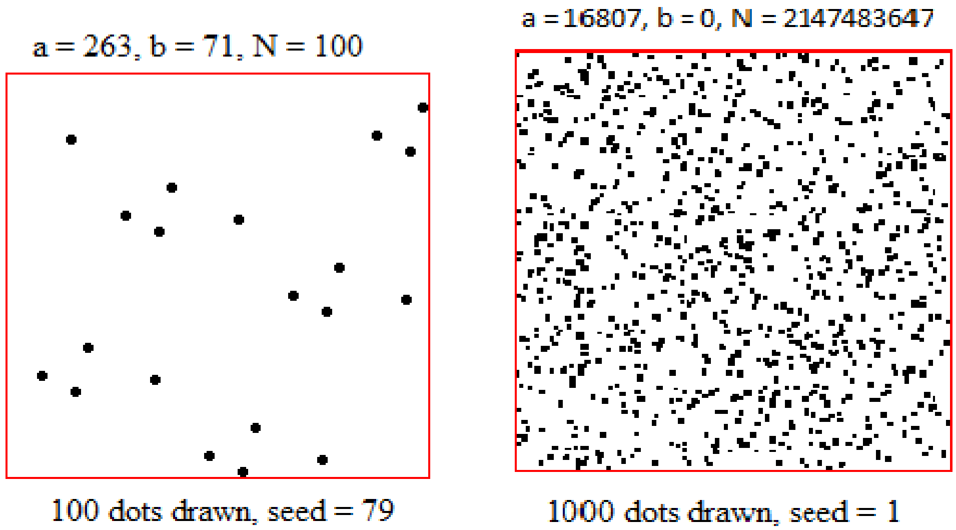

Figure 1.1: Generating the random numbers [18]

In the figure 1.1(a), the first coordinate measures horizontal distance from the left margin of the red box, and the second coordinate measures vertical distance downwards from the upper margin of the red box. Thus some of the points in this box have coordinates (79,48), (48,95), (95,56), etc. The values of the coordinates are scaled to fill the entire red box (which in this case measure 200 by 200 pixels). It's clear that there are only 20, points, and, since 100 were drawn, five lie on top of each other for each of the black dots. Moreover, the dots appear to lie along six slanted lines. As pointed out above, for a "good" random number generator there should be 100 points, and the distribution should be “random".

The figure 1.1 nearby shows a distribution of 1,000 points obtained with a widely used and well tested and analyzed random number generator using

$$a = 16807, \quad b = 0, \quad \text{and } N = 2^{31} - 1 = 2147483647.$$

This generator is described in the reference by Park and Miller given below.

One reason for the seemingly peculiar choice of N is that that particular number is the largest integer than can be represented on a Unix machine or in Java.

To illustrate the abilities of this applet consider the following Figure1.2 which shows three sequences of 100,000 points each, using the same generator, for k=1 (red), k = 2 (green), and k=3 (blue). Reassuringly, no systematic patterns are no systematic patterns are readily apparent.

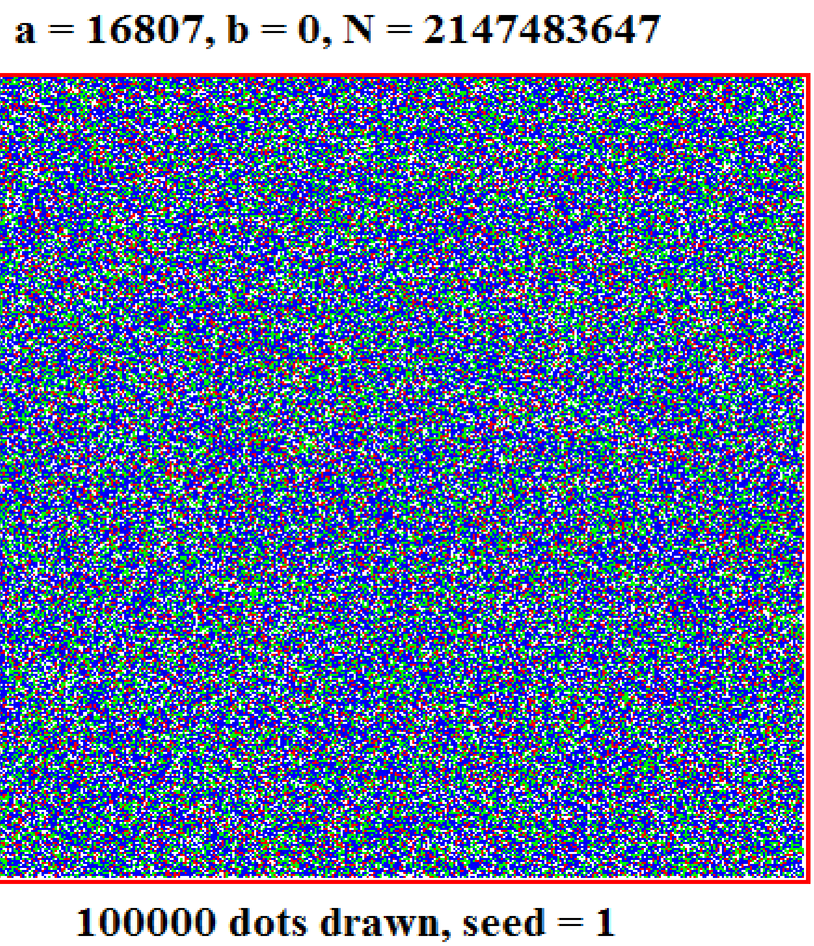

Figure 1.2: Generating 100000 random dots [18]

1.4 Types of random number generators

Random number generators can be classified into three groups, according to the source of their "randomness":

True random number generators (TRNGs): Truly random is defined as exhibiting "true" randomness, such as the time between "tics" from a Geiger counter exposed to a radioactive element. This type uses a physical source of randomness to provide truly unpredictable numbers. TRNGs are mainly used for cryptography, because they are too slow for simulation purposes. Many true random number generators are hardware solutions that you plug to a computer. The usual method is to amplify noise generated by a resistor (Johnson noise) or a semi-conductor diode and feed this to a comparator or Schmitt trigger. Once you sample the output, you get a series of bits which can be used to generate random numbers. True random number generators can be used for research, modeling, encryption, lottery prediction and parapsychological testing, among many other uses.

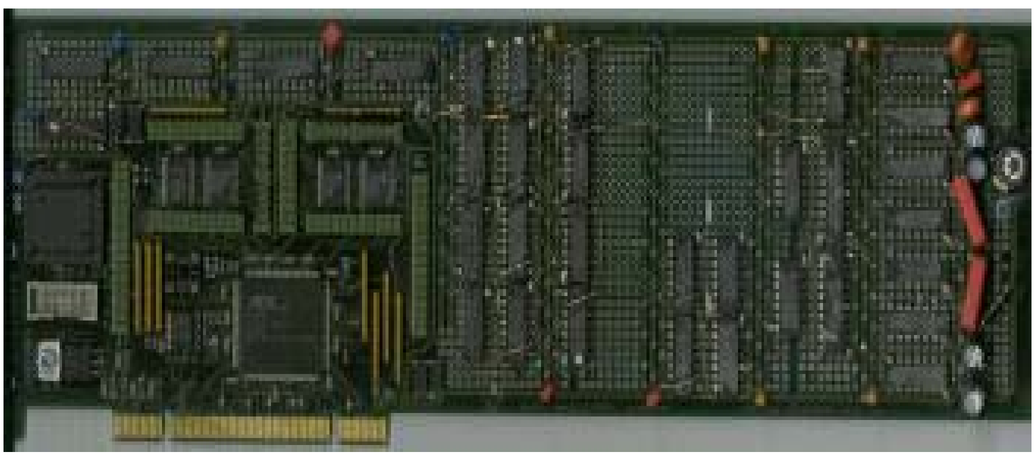

Figure 1.3: A PCI version generating true random numbers at 160 MByte/s [19]

Quasi-random number generators (QRNGs): Quasi-random is defined as filling the solution space sequentially (in fact, these sequences are not at all random - they are just comprehensive at a preset level of granularity). These generators attempt to evenly fill an n-dimensional space with points, without clustering or grouping of points. Although QRNGs are used in Monte Carlo simulations, we do not consider them in this chapter. If we change our generator so as to maintain a nearly uniform density of coverage of the domain then we have a random number generator known as quasi-random number generator. Quasi-random numbers give up serial independence of subsequently generated values in order to obtain as uniform as possible coverage of the domain. This avoids clusters and voids in the pattern of a finite set of selected points.

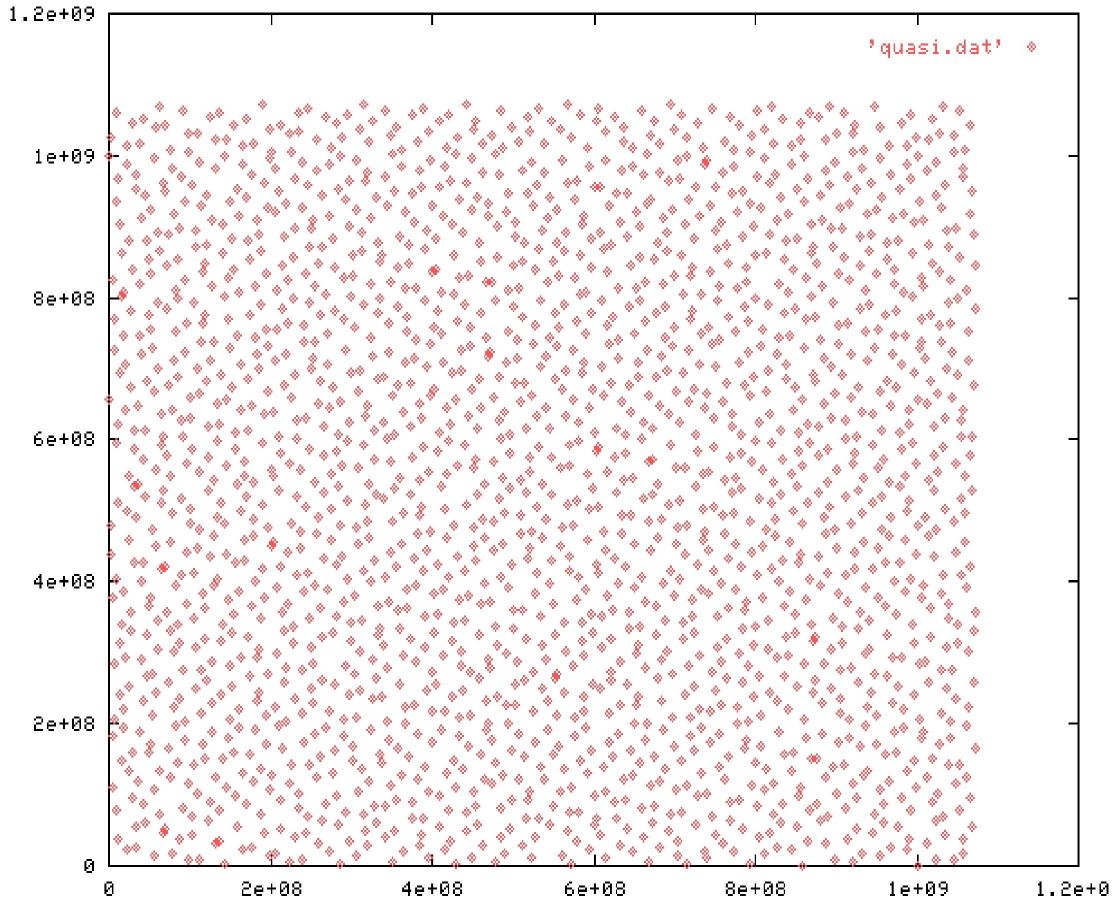

Figure 1.4: Quasi-random numbers [19]

Pseudorandom number generators (PRNGs): Pseudorandom is defined as having the appearance of randomness, but nevertheless exhibiting a specific, repeatable pattern. The most common type of random number generator, PRNGs are designed to look as random as a TRNG, but can be implemented in deterministic software because the state and transition function can be predicted completely. In this chapter, we don't consider only this type of generator.

An orthogonal classification of random number generators is organized according to the distribution of the numbers that are produced. Commonly encountered library functions, such as C's rand(), sample from the uniform distribution, meaning that within some range of numbers, each value is equally likely to occur.

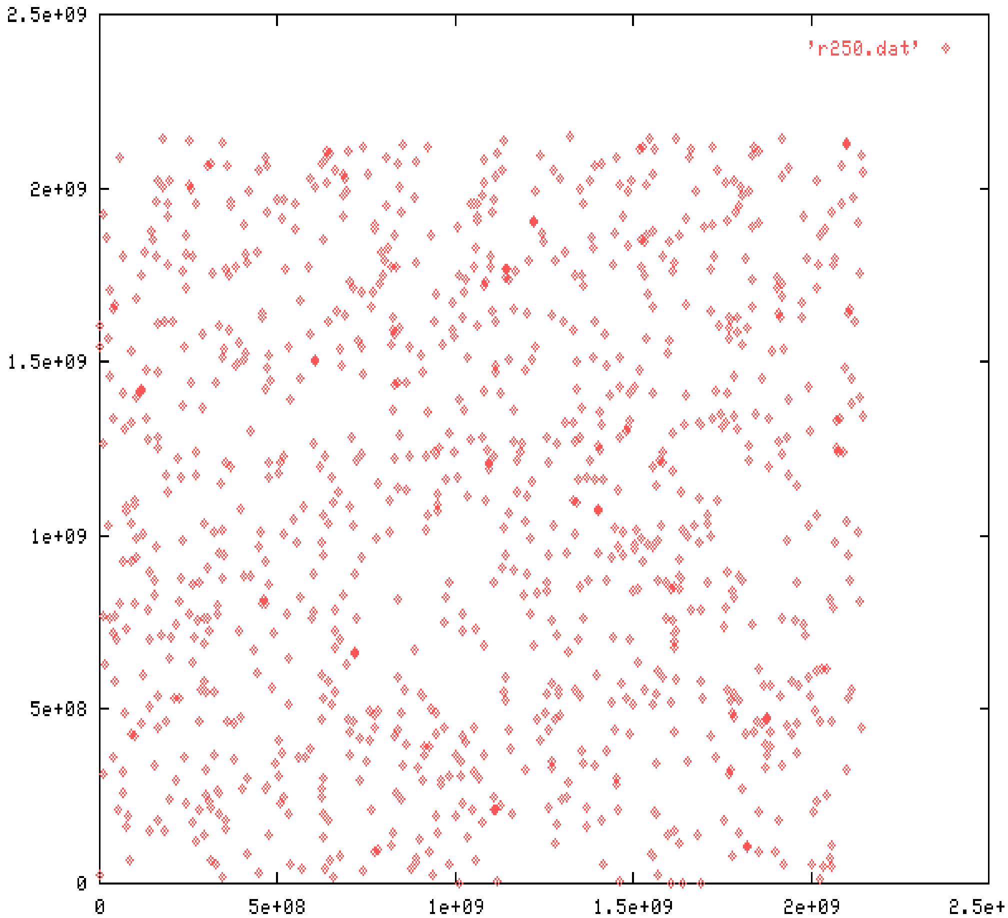

Figure 1.5: Pseudorandom numbers [19]

1.5 Applications of random numbers

Numbers that are "chosen at random" are useful in many different kinds of applications. For example:

Simulation: When a computer is being used to simulate natural phenomena, random numbers are required to make things realistic. Simulation covers many fields, from the study of nuclear physics (where particles are subject to random collisions) to operations research (where people come into, say, an airport at random intervals).

Sampling: It is often impractical to examine all possible cases, but a random sample will provide insight into what constitutes "typical" behavior.

Numerical analysis: Ingenious techniques for solving complicated numerical problems have been devised using random numbers.

Computer programming: Random values make a good source of data for testing the effectiveness of computer algorithms.

Decision making: There are reports that many executives make their decisions by flipping a coin or by throwing darts, etc. It is also rumored that some college professors prepare their grades on such a basis. Sometimes it is important to make a completely "unbiased decision; this ability is occasionally useful in computer algorithms, for example in situations where a fixed decision made each time would cause the algorithm to run more slowly. Randomness is also an essential part of optimal strategies in the theory of games.

Recreation: Rolling dice, shuffling decks of cards, spinning roulette wheels, etc., are fascinating pastimes for just about everybody. These traditional uses of random numbers have suggested the name "Monte Carlo method," a general term used to describe any algorithm that employs random numbers.

Chapter 2
Literature Review

2.1 Desirable Attributes of Random Numbers

The following are the desirable attributes of random numbers.

1. The random numbers should be uniformly distributed.
2. They should be statically independent.
3. Though the stream of random numbers will repeat depending on the parameters used in their generation, the stream length should be sufficiently larger than the desired length for a particular application.

The generation of random numbers should be faster.

Generating uniform random numbers: In this section we shall consider methods for generating a sequence of random fractions, i.e., random real numbers U_n uniformly distributed between zero and one. Since a computer can represent a real number with only finite accuracy, we shall actually be generating integersX_n, between zero and some number m; the fraction

$$U_n = X_n/m$$

will then lie between zero and one. Usually m is the word size of the computer, so Xn may be regarded (conservatively) as the integer contents of a computer word with the radix point assumed at the extreme right, and U, may be regarded (liberally) as the contents of the same word with the radix point assumed at the extreme left.

2.2 Methods of random numbers

2.2.1 Mid Square Method

The mid square method was proposed by Von-Newmann and Metropolis in 1946. In this method of random number generation, an initial seed is assumed and that number is squared. The middle four digits of the squared value are taken as the first random number. Next, the random number which is generated most recently is again squared and the middle most four digits of this squared value are assumed as the next random number. This is to be repeated to generate the required number of random numbers.

This method is demonstrated as shown in Table 2.1 by assuming the initial seed as 8765

Serial number	n(Four digits)	n^2
1	8765	76825225
2	8252	68095504
3	0955	00912025
4	9120	83174400
5	1744	03041536
6	0415	00172225
7	1722	02965284
8	9652	93161104
9	1611	Etc

Table2.1: Sample random numbers using mid square method

Steps of mid-square method:

The following are the steps of mid square method

Step 1: Input a four digit number, n and set the value of I to 1.

Step 2: Square the four digit number n.

Step3: Store the square of n into a string variable (X).

Step4: Add necessary number of zeros to the left of X so as to have a total of eight characters in X.

Step5: Select the middle four characters in X and store these four characters in the variable n.

Treat the value of n as the I th random number.

Step6: I = I + 1

Step7: If I is less than or equal to the required number of random numbers, then go to step 2, else go to step 8.

Step8: Stop.

The pseudo-code of this algorithm is given

```
Input n                    (* Four digit number)
      N                    (* Number of random numbers required)
For I = 1 to n do
      {
        n_square = n^2
        n_string = n_square (* Convert n_square into n_string)
        count the number of characters(n1) in n_string
        if n1 < 8 then do
              {
                    add (8-n1) zeros to the left of n_string
              }
```

```
            X = middle four characters on n_string

            n = X                (* convert string into integer)

            print I, n

        }

    Stop
```

Limitations of mid square method:

- Relatively slow
- Statistically unsatisfactory
- Sample of random numbers may be too short
- There is no relationship between the initial seed and the length of the sequence of random numbers

2.2.2 The Linear Congruential Method

By far the most popular random number generators in use today are special cases of the following scheme, introduced by D. H. Lehmer in 1949. We choose four "magic numbers:

m, the modulus; $m > 0$.
a, the multiplier; $0 \leq a < m$.
c, the increment; $0 \leq c < m$.
X_0, the starting value; $0 \leq X_0 < m$.

The desired sequence of random numbers (X_n) is then obtained by setting

$$X_{n+1} = (a X_n + c) \bmod m, \qquad n \geq 0 \qquad \ldots\ldots\ldots\ldots\ldots (1)$$

This is called a linear congruential sequence.

For example, the sequence obtained when $m = 10$ and $X_0 = a = c = 7$ is

$$7,6,9,0,7,6,9,0,\ldots\ldots\ldots \qquad \ldots\ldots\ldots\ldots\ldots\ldots (2)$$

As this example shows, the sequence is not always "random" for all choices of m, a, c, and X_0. Example (2) illustrates the fact that the congruential sequences always "get into a loop"; i.e., there is ultimately a cycle of numbers that is repeated endlessly. This property is common to all sequences having the general form X_{n+1} = f(X_n); the repeating cycle is called the period; sequence (2) has a period of length 4. A useful sequence will of course have a relatively long period.

The special case c = 0 deserves explicit mention, since the number generation process is a little faster when c = 0 than it is when c # 0. Lehmer's [15] original generation method had c = 0, although he mentioned c # 0 as a possibility; the idea of taking c # 0 to obtain longer periods is due to Thomson.

When the increment c=0, it is called multiplicative congruential method. When the increment c≠0, it is called mixed congruential method. The choice of a, c, m and X_0 drastically affects the statistical properties and cycle length.

2.2.2.1 Properties of Congruential Generators

All of the pseudorandom generators examined in this thesis are congruential generators where each term is defined recursively in terms of the k immediately preceding terms. We call this type of generator a recursive congruential generator, and it is expressed mathematically as

$$x_n = \mathrm{f}(x_{n-1}, x_{n-2}, \ldots, x_{n-k}) \bmod m$$

Note that f need not be linear, and need not use all k terms. We do assume that k is as small as possible, so x n depends on x_{n-k}. For example, in an LCG, f is defined by ax_{n-1} + c, and k = 1. However, f can be an arbitrary function, and $x_n = ax_{n-1} + (x_{n-7})^2$ is a perfectly good recursive congruential generator. This theorem provides an upper bound on the period, which is formalized in the subsequent corollary.

Lemma2.2.2.1: Let x and n be integers. If $x < n$ and $x \nmid n$, then there exists a positive integer k such that $n - x < kx < n$.

Theorem2.2.2.1: Suppose a recursive congruential generator produces the sequence (x_n), so x_n = f(x_{n-1}, x_{n-2} ,... , x_{n-k}) mod m for constants k and m. If a k-tuple repeats, or (x_a, x_{a+1},. .., x_{a+k}) = (x_{a+c}, x_{a+c+1},... , x_{a+c+1}) for some a and c, then x_i = x_{i+c} for all i ≥ a. Furthermore, the period λ divides c.

Corollary2.2.2.1: Any recursive congruential generator defined in terms of the k preceding terms has maximal period m^k.

Note that this theorem only states that a recursive congruential generator cannot have a period longer than m^k; it does not imply that a generator can achieve period m^k. The preceding theorems enable us to place an upper bound on equidistribution of a congruential sequence, as formalized in the following theorem.

Theorem2: Any recursive congruential generator defined in terms of the k preceding terms can be equidistributed in no more than k dimensions.

Generating Real Values: Congruential generators produce integers in the discrete set $\mathbb{Z}_m$. Many applications, however, assume random sequences of numbers drawn from a real interval. A computer has finite memory, so it cannot represent all real numbers with infinite precision, but it can approximate them up to the precision of its numerical representation by using a finite set of closely spaced rational numbers. Most generators are congruential and thus integer-valued, so a method is needed to transform their output to an approximation of real numbers in [0, 1). Typically, this transformation is done by dividing each term in the sequence by the modulus. The new sequence is then distributed over the set { $\frac{n}{m}$: n ∈ $\mathbb{Z}_m$ }, which approximates the real interval [0, 1) well when m is large. However, the distance between any two terms in the sequence can be no smaller than $\frac{1}{m}$. Whether a given value of m leads to a sufficiently accurate approximation of the real interval depends on the requirements of the application. Most congruential generators strive to generate numbers uniformly in $\mathbb{Z}_m$, so when they are transformed to real values they approximate the uniform probability distribution on the interval [0,1), which we denote U[0, 1).

2.2.2.2 LCG Full period: m is c ≠ 0, m − 1 otherwise (if c = 0, 0 is a fixed point for the recurrence). Considere the case c ≠0.

Theorem (Period): The LCG has full period if and only if the following three conditions hold:

1. The only positive integer that (exactly) divides both m and c is 1;
2. If q is a prime number that divides m, then q divides a- 1;
3. If 4 divide m, then 4 divide a-1.

A popular LCG is the"standard minimal", as known from the terminology introduced by Park and Miller in 1988:

$$X_{n+1} = 16807X_n \bmod 2147483647.$$

Observe that $2147483647 = 2^{31} - 1$; on 3 2-bit architectures, the largest representable (signed) integer is 2^{31}.

The Standard Minimal Generator

```
#define a 16807

#define m 2147483647

#define AM_MIL (1.0/m)

#define IQ_MIL 127773

#define IR_MIL 2836

double ran_standardminimal_get_val(long * state)

{

 long k;

double ans; k=( * state)/IQ_MIL;

idum=a * ( * state-k * IQ_MIL)-IR_MIL * k;
```

```
if ( * state < 0) * state += m;

ans=AM_MIL * ( * state); return ans;

}
```

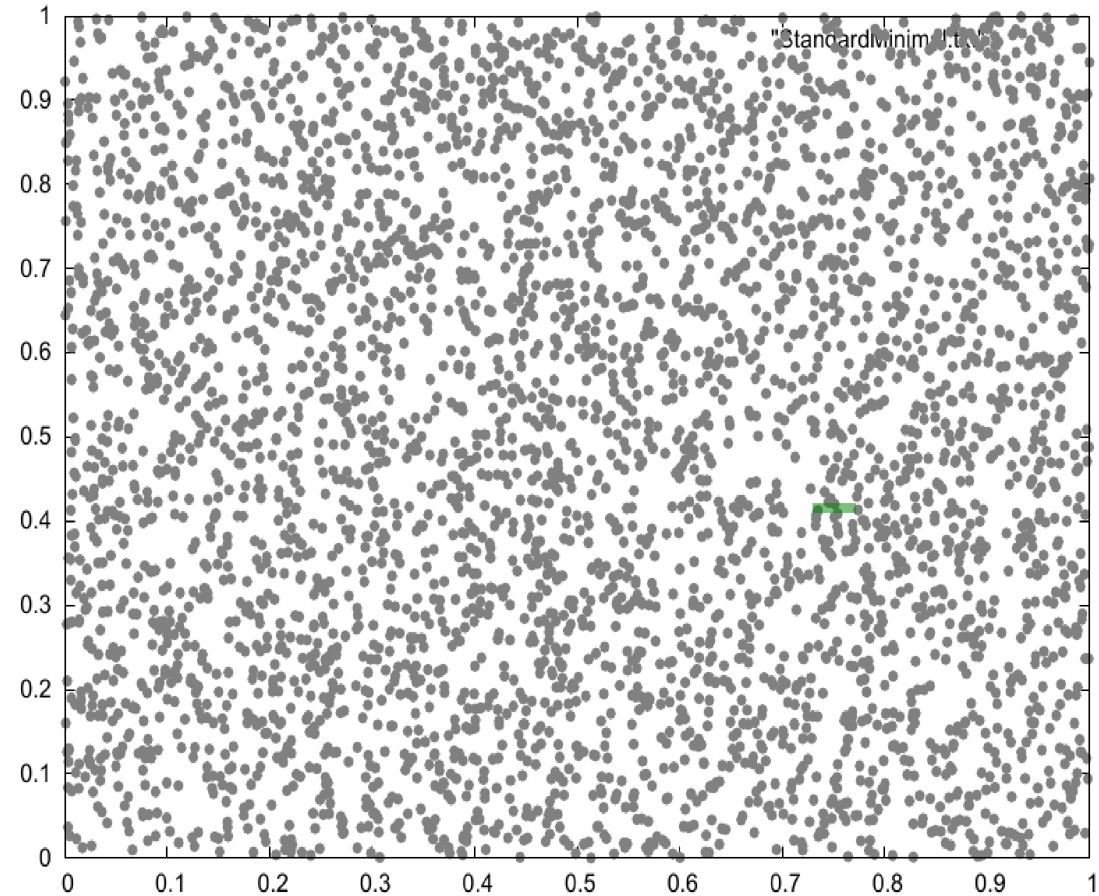

Figure 2.1: 4096 generated points on the unit square

2.2.2.3 Multiplicative congruential Method

The multiplicative congruential method is an arithmetic procedure to generate a finite sequence of uniformly distributed random numbers. Two integers P and Q are congruent if their difference is an integral multiple of m. This is represent as shown below

$$P \equiv Q \pmod{m}$$

This means that" P is congruent to Q modulo m" and further the following are true

1. (P-Q) is divisible by m.

2. P and Q, when divided by m, leave identical remainders.

Some examples of congruent relationships are shown below.

$$53738 \equiv 38 \pmod{100}$$
$$97 \equiv 67 \pmod{11}$$

Let X_i be the ith uniformly distributed random number.

Then (i+1)th random number is given by the following relation.

$$X_{i+1} = a\, X_i \pmod{m}$$

where a and m are nonnegative integers.

That is, X_{i+1} = Remainder of [(a X_i)/m]

The range of the random number that will be generated using this relation is from 0 to 1.

The value of m is given by the following formula

$$m = 2^r$$

Where r is the number of bits in the computer word.

The value of a is given by the following formula

$$a = 8t \pm 3$$

where t is any positive integer.

The initial value of X_i (That is X_0) is any positive old integer.

The steps of the multiplicative congruential method for generation of uniformly distributed random numbers are presented as follows:

Step 1: Input the following:

1. Choose any number less than nine digits and assign it to X_0.
2. Assign at least five digits value for a.
3. Value for m.

Step 2: Set i = 1

$$X = X_0$$

Step 3: Find the product of a and X

$$Y = a \text{ x } X$$

Step 4: Divide Y by m and do the following.

Step 4.1: Store the remainder as the ith random number.

R_i = Remainder of [Y/m] = Y – Int[Y/m] x m

Step 4.2: Store the quotient as X.

X = Int[y/m]

Step 5: Store or print or use the ith random number (R_i).

Step 6: i = i +1

Step 7: If I is less than or equal to the required number of random numbers, then go to step3, else go to step 8.

Step 8: Stop.

The pseudo-code of this algorithm is shown

```
Input   X0          (* 9 digits number)
        a           (* at least 5 digits number )
        m
        N           (* number of required random numbers)
Initialize X = X0
for i = 1 to N do
        {
           Y = a*X
           Z = Y/m
           R(i) = Y –(int (Z)*m)
           X = int (Z)      (* integer of Z)
           Print R(i)
        }
stop.
```

2.2.2.4 Merits and Demerits of LCG

- LCGs are fast and require minimal memory (typically 32 or 64 bits) to retain state. This makes them valuable for simulating multiple independent streams.
- Linear congruential random number generators are widely used in simulation and Monte Carlo calculations. Because they are very fast, and because they have

minimal state space, they remain attractive for use in parallel computing environments.

- LCGs should not be used for applications where high-quality randomness is critical. For example, it is not suitable for a Monte Carlo simulation because of the serial correlation (among other things). They should also not be used for cryptographic applications.
- LCGs tend to exhibit some severe defects. For instance, if an LCG is used to choose points in an n-dimensional space, the points will lie on, at most, $m^{1/n}$ hyper planes (Marsaglia's Theorem, developed by George Marsaglia). This is due to serial correlation between successive values of the sequence X_n. The spectral test, which is a simple test of an LCG's quality, is based on this fact.
- A further problem of LCGs is that the lower-order bits of the generated sequence have a far shorter period than the sequence as a whole if m is set to a power of 2. In general, the nth least significant digit in the base b representation of the output sequence, where $b^k = m$ for some integer k, repeats with at most period b^n.
- Nevertheless, LCGs may be a good option. For instance, in an embedded system, the amount of memory available is often very severely limited. Similarly, in an environment such as a video game console taking a small number of high-order bits of an LCG may well suffice. The low-order bits of LCGs when m is a power of 2 should never be relied on for any degree of randomness whatsoever. Indeed, simply substituting 2^n for the modulus term reveals that the low order bits go through very short cycles. In particular, any full-cycle LCG when m is a power of 2 will produce alternately odd and even results.

2.2.3 Quadratic Congruential Method

It is proposed by R.R.Coveyou. It is used to get more random numbers.

$$X_{n+1} = (dX_n^2 + aX_n + c) \bmod m \qquad \text{……………………} \ (3)$$

The conditions on d, a and c required for the maximum period (which matches the modulus) are:

1. c must be relatively prime to the modulus.

2. a is equal to 1 modulo every odd prime factor of the modulus.
3. d is equal to 0 modulo every odd prime factor of the modulus.
4. If the modulus is divisible by 4, then a is equal to 1 modulo 2, and d is equal to a-1 modulo 4
5. If the modulus is divisible by 2, either d is even and a is odd, which is the only possibility if the modulus is divisible also by 4, or d is odd and a is even
6. If the modulus is divisible by 9, either a is equal to 0 modulo 9, or a is equal to 1 modulo 9, and d times c is equal to 6 modulo 9.

An interesting quadratic method has been proposed by R.R. Coveyou when m is a power of two.

$$X_0 \bmod 4 = 2 \qquad X_{n+1} = X_n(X_n + 1) \bmod 2^e \qquad \text{………………(4)}$$

This sequence can be computed with about the same efficiency as (1), without any worries of overflow.

2.2.4 Fibonacci Generator

The simplest sequence in which X_{n+1} depends on more than one of the preceding values is the Fibonacci sequence,

$$X_{n+1} = (X_n + X_{n-1}) \bmod m. \qquad \text{……………………………. (5)}$$

This generator was considered in the early 1950s, and it usually gives a period length greater than m; but tests have shown that the numbers produced by the Fibonacci recurrence (5) are definitely not satisfactorily random.

Problems with Fibonacci Generator (FG):

- The initialization of FGs is a very complex problem. The output of FGs is very sensitive to initial conditions, and statistical defects may appear initially but also periodically in the output sequence unless extreme care is taken.
- Another potential problem with FGs is that the mathematical theory behind them is incomplete, making it necessary to rely on statistical tests rather than theoretical performance

2.2.5 Combined Multiple Recursive Generator (CMRG)

An approach is to combine two or more multiplicative congruential generators to have good statistical properties and longer periods.

$$X_n = (X_n + X_{n-k}) \bmod m \qquad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots.. (6)$$

When k is a comparatively large value.

When $k \leq 15$, the sequence fails to gap test. The gap test means counts the number of digits that appear between repetitions of a particular digit and then uses the Kolmogorov-Smirnov test to compare with the expected number of gaps.

Consider two (or more) MRG's working in parallel:

$X_{1,n} = (a_{1,1}X_{1,n-1} + \cdots + a_{1,k}X_{1,n-k}) \bmod m_1$,

$X_{2,n} = (a_{2,1}X_{2,n-1} + \cdots + a_{2,k}X_{2,n-k}) \bmod m_2$.

We define the two combinations

$Z_n := (X_{1,n} - X_{2,n}) \bmod m_1$; $U_n := Z_n / m_1$;

$W_n := (X_{1,n} / m_1 - X_{2,n} / m_2) \bmod 1$.

The sequence $\{W_n , n \geq 0\}$ is the output of another MRG, of module $m = m_1 m_2$, and $\{U_n , n \geq 0\}$ is nearly the same sequence if m_1 and m_2 are close. We can achieve the period $(m_1^k - 1)(m_2^k - 1)/2$.

MRG32k3a: The following combined MRG was proposed by L'Ecuyer, and is amongst the most popular and efficient known generators. It combines 2 MRG's.

$k = 3$,

$m_1 = 2^{32} - 209$, $a_{11} = 0$, $a_{12} = 1403580$, $a_{13} = -810728$,

$m_2 = 2^{32} - 22853$, $a_{22} = 527612$, $a_{22} = 0$, $a_{23} = -1370589$.

Combination: $Z_n = (X_{1,n} - X_{2,n}) \bmod m_1$.

Corresponding MRG: $k = 3$, $m = m_1 m_2 = 18446645023178547541$,

a_1 = 18169668471252892557,

a_2 = 3186860506199273833,

a_3 = 8738613264398222622.

Period ρ = $(m_1^3 - 1)(m_2^3 - 1)/2 \approx 2^{191}$

MRG32k3a Implementation:

```
#define norm 2.328306549295728e-10 / * 1/(m1+1) * /

#define m1 4294967087.0

#define m2 4294944443.0

 #define a12 1403580.0

#define a13n 810728.0

#define a21 527612.0

#define a23n 1370589.0

double s10, s11, s12, s20, s21, s22;

double MRG32k3a ()

{

long k;

double p1, p2;

/* Component 1 */

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 */

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)

return ((p1 - p2 + m1) * norm);

else return ((p1 - p2) * norm);

}
```

The generator ”L’Ecuyer” presentin AMLET is a combination of two LCG’s. It is a bit less efficient than MRG32K3a (which will replace it in a future version), but still very good.

Merits of CMRG:

Combining parallel multiple recursive sequences provides an efficient way of implementing random number generators with long periods and good structural properties. Such generators are statistically more robust than simple linear congruential generators that fit into a computer word

2.2.6 Additive number generator (ANG)

It was devised by G.J.Mitchell and D.P. Moore.

$$X_n = (X_{n-24} + X_{n-55}) \bmod m, \qquad n \geq 55 \qquad \ldots\ldots\ldots\ldots\ldots\ldots\ldots (7)$$

where m is even and $X_0, \ldots\ldots\ldots.., X_{54}$ are arbitrary integers not all even. The constants 24 and 55 in this definition were not chosen at random; they are special values that happen to have the property that the least significant bits ($X_n \bmod 2$) will have a period of length 2^{55} - 1. Therefore the sequence (X_n) must have a period at least this long.

Algorithm (Additive number generator):

Memory cells Y [I], Y [2],....., Y [55] are initially set to the valuesX_{54} , X_{53}, . . . , X_0, respectively; j is initially equal to 24 and k is 55. Successive performances of this algorithm will produce the numbers X_{55}, X_{56} . . . as output.

Al. [Add.] (If we are about to output X_n at this point, Y[j] now equals X_{n-24} and Y[k] equals X_{n-55}.) Set Y[k] ← (Y[k] + Y[j]) mod 2^e, and output Y[k].

A2. [Advance.] Decrease j and k by 1. If now j = 0, set j ← 55; otherwise if k = 0, set k ← 55.

This generator is usually faster than the previous methods we have been discussing, since it does not require any multiplication. Besides its speed, it has the longest period we have seen yet; and it has consistently produced reliable results, in extensive tests since its invention in 1958. Furthermore, as Richard Brent has observed, it can be made to work correctly with floating point numbers, avoiding the need to convert between integers and fractions. Therefore it may well prove to be the very best source of random numbers for practical purposes. The only reason it is difficult to recommend sequence (7) wholeheartedly is that there is still very little theory to prove that it does or does not have desirable randomness properties; essentially all we know for sure is that the period is very long, and this is not enough. John Reiser (Ph. D. thesis, Stanford Univ., 1977) has shown, however, that an additive sequence like (7) will be well distributed in high dimensions, provided that a certain plausible conjecture is true.

The fact that the special numbers (24, 55) in (7) work so well follows from theoretical results developed in some of the exercises below. Table 1 lists all pairs (I, k) for which the sequence $X_n = (X_{n-l} + X_{n-k})$ mod 2 has period length 2^k - 1, when k < 100. The pairs (1, k) for small as well as larger k are shown, for the sake of completeness; the pair (1, 2) corresponds to the Fibonacci sequence mod 2, whose period has length 3. However, only pairs (1, k) for relatively large k should be used to generate random numbers in practice.

(1, 2)	(1, 15)	(5, 23)	(7, 31)	(5, 47)	(21, 52)	(18, 65)	(28, 73)	(2, 93)
(1, 3)	(4, 15)	(9, 23)	(13, 31)	(14, 47)	(24, 55)	(32, 65)	(31, 73)	(21, 94)
(1, 4)	(7, 15)	(3, 25)	(13, 33)	(20, 47)	(7, 57)	(9, 68)	(9, 79)	(11, 95)
(2, 5)	(3, 17)	(7, 25)	(2, 35)	(21, 47)	(22, 57)	(33, 68)	(19, 79)	(17, 95)
(1, 6)	(5, 17)	(3, 28)	(11, 36)	(9, 49)	(19, 58)	(6, 71)	(4, 81)	(6, 97)
(1, 7)	(6, 17)	(9, 28)	(4, 39)	(12, 49)	(1, 60)	(9, 71)	(16, 81)	(12, 97)
(3, 7)	(7, 18)	(13, 28)	(8, 39)	(15, 49)	(11, 60)	(18, 71)	(35, 81)	(33, 97)
(4, 9)	(3, 20)	(2, 29)	(14, 39)	(22, 49)	(1, 63)	(20, 71)	(13, 84)	(34, 97)
(3, 10)	(2, 21)	(3, 31)	(3, 41)	(3, 52)	(5, 63)	(35, 71)	(13, 87)	(11, 98)
(2, 11)	(1, 22)	(6, 31)	(20, 41)	(19, 52)	(31, 63)	(25, 73)	(38, 89)	(27, 98)

For each pair (l, k), the pair $(k-l, k)$ is also valid (see exercise 24), hence only values of $l \leq k/2$ are listed here. For extensions of this table, see N. Zierler and J. Brillhart, *Information and Control* **13** (1968), 541–554; **14** (1969), 566–569; **15** (1969), 67–69.

Table 2.2: Subscript pairs yielding long period mod 2[14]

Merits:

It is faster than previous methods since it does not require any multiplication. It can be made to work correctly with floating point numbers, avoiding the need to convert between integers and fraction.

2.2.7 Combination of random number generator

Suppose we have two sequences X_0, X_1, ….. and Y_0, Y_1, …….. of random numbers between 0 and m-1, preferably generated by two unrelated methods.

$$Z_n = (X_n + Y_n) \bmod m \qquad \ldots\ldots\ldots\ldots\ldots\ldots\ldots(8)$$

In this case, the period will be quite long if the period lengths of (X_n) and (Y_n) are relatively prime to each other.

Algorithm M (Randomizing by shuffling):

Given methods for generating two sequences (X_n) and (Y_n), this algorithm will successively output the terms of a "considerably more random" sequence. We use an auxiliary table V[0], V[l], . . . , V[k-1], where k is some number chosen for convenience, usually in the neighborhood of 100. Initially, the V-table is filled with the first k values of the X-sequence. MI. [Generate X,Y.] Set X and Y equal to the next members of the sequences (X_n) and (Y_n), respectively.

M2. [Extract j.] Set j $\leftarrow$ [kY/m], where m is the modulus used in the sequence (Y_n); that is, j is a random value, $0 \leq j < k$, determined by Y.

M3. [Exchange.] Output V[j] and then set V[j] $\leftarrow$ X.

As an example, assume that Algorithm M is applied to the following two sequences, with k = 64:

$X_0 = 5772156649$, $X_{n+1} = (3141592653X_n + 2718281829) \bmod 2^{35}$;

$Y_0 = 1781072418$, $Y_{n+1} = (2718281829Y_n + 3141592653) \bmod 2^{35}$.

On intuitive grounds it appears safe to predict that the sequence obtained by applying Algorithm M will satisfy virtually anyone's requirements for randomness in a computer-generated sequence, because the relationship between nearby terms of the output has been almost entirely obliterated. Furthermore, the time required to generate this sequence is only slightly more than twice as long as it takes to generate the sequence (X_n) alone.

However, there is an even better way to shuffle the elements of a sequence, discovered by Carter Bays and S. D. Durham [ACM Trans. Math. Software 2 (1976), 59-64]. Their approach, although it appears to be superficially similar to Algorithm M, can give surprisingly better performance even though it requires only one input sequence (X_n) instead of two:

Algorithm B (Randomizing by shuffling):

Given a method for generating a sequence (X_n), this algorithm will successively output the terms of a "considerably more random" sequence, using an auxiliary table V[0], V[l], . . . …, V[k – 1] as in Algorithm M. Initially the V-table is filled with the first k values of the X-sequence, and an auxiliary variable Y is set equal to the (k + 1)st value.

B1. [Extract j.] Set j $\leftarrow$ [kY/rn], where m is the modulus used in the sequence (X_n); that is, j is a random value, $0 \leq j < k$, determined by Y.

B2. [Exchange.] Set Y $\leftarrow$V[j], output Y, and then set V[j] to the next member of the sequence (X_n).

Chapter 3
Problem Statement

Random number generation is the fundamental problem applicable in many real life applications. The demand for random numbers in scientific applications is increasing. However, the most widely used multiplicative, congruential random-number generators with modulus $2^{31} - 1$ have a cycle length of about 2.1 x 10^9. Moreover, developing portable and efficient generators with a larger modulus such as $2^{61} - 1$ is more difficult than those with modulus $2^{31} - 1$. Linear congruential random-number generators with Mersenne prime modulus and multipliers of the form a = ± 2^q $\pm 2^r$ have been developed. Their main advantage is the availability of a simple and fast implementation algorithm for such multipliers. Disadvantage is generalizes the algorithm points out statistical weaknesses of these multipliers when used in a straightforward manner as seen in previous chapters.

After studying and comparing the existing algorithms some implements will be suggested for combined multiple recursive random number generation as follows:

A class of combined multiple recursive random number generators constructed in a way that each component runs fast and is easy to implement, while the combination enjoys excellent structural properties as measured by the spectral test. Each component is a linear recurrence of order k> 1, modulo a large prime number, and the coefficients are either 0 or are of the form a = ± 2^q or a =± 2^q ± 2^r. This allows a simple and very fast implementation, because each modular multiplication by a power of 2 can be implemented via a shift, plus a few additional operations for the modular reduction. Select the parameters in terms of the performance of the combined generator in the spectral test to provide a specific implementation.

Chapter 4
Design and Implementation

4.1 Multiple Recursive Generators

The multiple recursive generator (MRG) generalizes the multiplicative linear congruential generator from a multiple of the previous term to a linear combination of the previous k terms. A multiple recursive generator (MRG) is defined by the linear recurrence

$$x_n = (a_1 x_{n-1} + \cdots + a_k x_{n-k}) \bmod m; \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \text{(9)}$$

$$u_n = x_n / m. \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \text{(10)}$$

The modulus m and the order k are positive integers, the coefficients a_i belong to $\mathbb{Z}_m = \{0, 1, \ldots, m-1\}$, and the state at step n is the vector $x_n = (x_{n-k+1}, \ldots, x_n)$. The maximal period length of the recurrence (1) is $\rho = m^k - 1$ and this length is attained if and only if m is prime and the characteristic polynomial of the recurrence,

$$P(z) = z^k - a_1 z^{k-1} - \cdots - a_k,$$

is a primitive polynomial modulo m. Primitive polynomials [32][33] can be found by random search. To obtain a primitive polynomial, one needs at least 2 non- zero coefficients a_i's. This minimal number of non-zero coefficients yields the following economical version of (9):

$$x_n = (a_r x_{n-r} + a_k x_{n-k}) \bmod m. \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \text{(11)}$$

The classical linear congruential generator (LCG) is obtained with k = 1. Further details about MRGs can be found, e.g., in Knuth (1998), Niederreiter (1992), L'Ecuyer (1994), L'Ecuyer (1996), and the references therein. Besides period length, two important issues in the design of MRGs are the statistical quality and the efficiency of the implementation. The statistical quality is traditionally assessed by measuring the uniformity of the set of all vectors of successive output values $(u_n, \ldots, u_{n+t-1})$, from all initial states, as we explain in Section 4.4. For the implementation, the key issue is how to compute efficiently the products a_ix mod m when m is large.

The mathematical properties of the multiple recursive generator resemble those of the inversive congruential and feedback shift register generators, which we discuss after it. An in-depth exploration of these properties is beyond the scope of this thesis. The computation of a sequence using this generator takes about k times as long as with a simple LCG, and requires about k times as much storage space, since at any point the previous k terms must be stored. Thus we see that although a larger k value increases the period by a factor of m, it also increases computation time and storage space linearly. One way to increase computational efficiency is to set all multipliers to ±1 or 0, as suggested by Watson in 1962 [28]. A particularly good generator of this form is $x_n = x_{n-24} + x_{n-55}$ mod m, with the restriction that the initial values must not all be even (Knuth 1997 [14]). The least significant bit in the generated sequence has period $2^{55} - 1$, so the generator will have at least this period for any value of m. Furthermore, it has a period of $2^{k-1}(2^{55} - 1)$ if $m = 2^k$. Since this generator doesn't require any multiplication, it is faster even than the simple LCG, and has a much longer period. Deng and Lin in 2000 [2] proposed a slightly different way to achieve efficiency with the MRG. They recommend a so-called Fast MRG of the form $x_n = bx_{n-k} - x_{n-1}$ mod m for some k, where the multiplier b is chosen such that the generator has full period. This generator eliminates the tradeoff of high computation time, but maintains the long period afforded by keeping track of k previous values.

4.1.1 Feedback Shift Register Generators

The feedback shift register generator (FSRG), which is a special form of the multiple recursive generator with modulus 2, was introduced by Tausworthe in 1965 [27]. It has the form

$$x_n = (a_1 x_{n-1} + \cdots + a_k x_{n-k}) \bmod 2.$$

It generates individual bits, 0 or 1, based on the preceding k bits. The multipliers a_i are either 0 or 1, since any other value is equivalent to one of these mod 2. To begin generation, k seed bits are required. As shown in Theorem 2.2.2.1 and Corollary 2.2.2.1, the upper bound on the period of an FSRG is 2^k because the entire sequence repeats as soon as a k-tuple repeats. However, in an FSRG, the k-tuple consisting of all zeros results in the remainder of the sequence being all zeroes, so a usable FSRG must not produce this k-tuple. Therefore the maximal period for an FSRG is actually 2^{k-1} (Gentle 2003

[11]). Addition mod 2 is equivalent to the exclusive-or operation, which is very efficient on a computer. To increase efficiency further, it is common to set all coefficients to zero except for two, as with MRGs. The generator then takes the form $x_n = x_{n-p} + x_{n-k}$ mod 2, and is called a two-tap generator. In general, a generator with N non-zero coefficients is called an N-tap generator. The generator is named for the fact that it can be implemented efficiently using a single register and the shift operation, as illustrated in the following diagram. In each time step, the bits shift down the register according to the arrows, and a new bit is calculated and pushed onto the end.

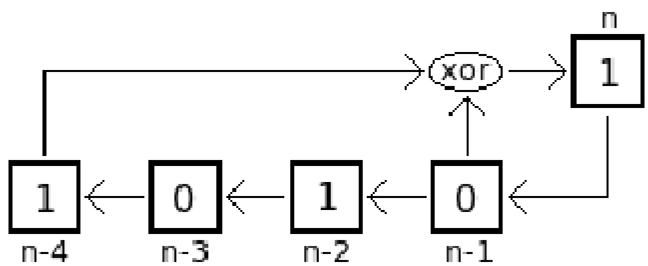

Figure 4.1: Feedback Shift Register Generators

This diagram illustrates the generator $x_n = x_{n-1} + x_{n-4}$ mod 2 with seed values (1,0,1,0), which produces the sequence listed below. This generator achieves its maximal period, $2^4 - 1 = 15$, and starts repeating with $x_4 = x_{19}$, $x_5 = x_{20}$, etc.

N	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
x_n	1	0	1	0	1	1	0	0	1	0	0	0	1	1	1	1	0	1	0

Table 4.1: Generate the binary numbers using FSRG

The conditions for coefficients that produce a maximal period involve finding primitive polynomials in $\mathbb{Z}_2$; see Lidl and Niederreiter [17] and Golomb [12] for an extensive discussion of FSRGs and their properties. For our purposes, it suffices to say that the conditions are restrictive, but not difficult to meet. Where other random number generators generate sequences of integers in $\mathbb{Z}_m$ for some large m, the feedback shift register generator has m = 2, and produces a sequence of bits. In the following sections, we discuss several methods of using this bit sequence to generate larger integers like the other generators we have discussed.

Implementation:

A first approach for computing ax mod m is approximate factoring [34][43]. It uses integer arithmetic and a clever decomposition of m. It works if a^2 < m or if a = m/i where a^2 < m, and if all integers between−m and m are well represented on the computer.

A second approach computes the product and the division by m (for the mod operation) directly in floating-point arithmetic. On computers that obey the IEEE 64-bit floating point standard (most computers do), all integers up to 2^{53} are represented exactly in floating point, and the floating-point implementation works if am < 2^{53}. See L'Ecuyer (1999a) for details and examples.

A third approach, introduced by Wu (1997) and generalized by L'Ecuyer and Simard (1999), which we call the powers-of-2 decomposition, assumes that a is a sum or a difference of a small number of powers of 2, e.g. a =± 2^q ± 2^r. The product of x by each power of 2 can be implemented by a left shift of the binary representation of x, and the product ax is computed by adding and/or subtracting. The details are given in Section 3. This approach turns out to be more efficient than the other two, according to our experiments. Note that replacing any a_i by a_i ± m changes nothing to the recurrence (9). If for some $a_i \in \mathbb{Z}_m$, $|a_i - m|$ satisfies one of the above conditions whereas a_i does not satisfy that condition, then one can replace a_i by $\tilde{a}_i = a_i - m$ when implementing (9). This is equivalent to allowing negatives values for thea_i's, which we shall do in the remainder of this paper.

4.2 Combined MRGs

A direct efficient implementation of the recurrence (9) can generally be obtained only when the number of non-zero coefficients a_i is small, and when special conditions are imposed on these coefficients, as explained in the previous subsection. However, imposing these constraints usually implies that the resulting MRG has a poor lattice structure [5]. In particular, good behavior is possible only if the sum of squares of the a_i is large [5]. This has motivated the introduction of combined MRGs, which are constructed so that the components are easy to implement efficiently while the structure of the resulting combined generator has good quality. L'Ecuyer has proposed and analyzed the following class of combined MRGs, based on J linear recurrences, with the same order k and distinct prime moduli m j, running in parallel. For 1≤ j ≤ J , let

$$x_{j,n} = (a_{j,1}\ x_{j,n-1} + \cdots + a_{j,k}\ x_{j,n-k}) \bmod m_j \ \ldots\ldots\ldots\ldots\ldots (11)$$

and suppose that the recurrence (1.3) has period $\rho_j = m_j^k - 1$. Suppose also that the least common multiple of ρ_1 ,..., ρ_J is ρ = $\rho_1 \cdots \rho_J / 2^{J-1}$. (This is the best that one can do, because each ρ j is necessarily even.)

Define the two combinations

$$z_n = (\textstyle\sum_{j=1}^{J} \delta_j x_{j,n}) \bmod m_1 \ ; \quad u_n = z_n / m_1 \ \ldots\ldots\ldots\ldots\ldots (11)$$

and

$$w_n = \left(\sum_{j=1}^{J} \frac{\delta_j x_{j,n}}{m_j}\right) \bmod 1 \ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (11)$$

where the δ_j 's are integers such that δ_j is relatively prime with x_j for each j . L'Ecuyer (1996) has shown that the sequence $\{w_n$,n ≥ 0} defined by (4.2) is the same as the sequence $\{u_n$,n≥ 0} produced by the MRG (1–2), with m = $m_1 \cdots m_J$, and explains how to compute the corresponding a_i's. Moreover, the numbers u_n and w_n produced by (4.2) and (4.2), respectively, differ only by a very small quantity when the m_j are close to each other. Explicit bounds on the difference are given in L'Ecuyer (1996). The generators considered in this paper are combined MRGs of this form, constructed so that the structure of the combined MRG has excellent quality while (4.2) can be implemented efficiently for each j. This was already achieved by L'Ecuyer [1] for implementations based on approximate factoring and on floating point arithmetic, respectively. The aim of this paper is to propose combined MRGs that are faster for an equivalent statistical quality, by using the powers-of-2 decomposition method.

4.3 Overview of the Remainder

The remainder of the paper is organized as follows. In Section 4.4, we recall the quality criteria for selecting MRGs based on an analysis of their lattice structure by the spectral test. In Section 4.4.1, we describe the implementation method for coefficients that are a sum or a difference of a small number of powers of 2. In Section 4.5, we explain how we searched for good single and combined MRGs with coefficients of this form, and why we prefer the combined MRGs over the non-combined ones. In Section 4.5.1, we give a specific implementation of a combined MRG of this form and we compare its speed with other combined MRGs proposed in L'Ecuyer, and implemented via approximate factoring and floating point arithmetic.

4.4 Lattice Structure and Quality Criteria

Let

$$\Psi_t = \{(u_0, ..., u_{t-1}) : (x_0, ..., x_{k-1}) \in \mathbb{Z}_m^k\}, \ldots\ldots\ldots\ldots\ldots\ldots (11)$$

the set of all vectors of t successive output values of the MRG, from all possible initial states. If the initial seed of the MRG is chosen at random, this Ψ_t is viewed in a sense as the sample space from which points are chosen at random to approximate the uniform distribution over the t-dimensional unit hypercube $[0,1)^t$. This means that the generator should be constructed so that Ψ_t covers $[0,1)^t$ very evenly, for t up to some arbitrary number. It is well known that the set ̋ t for an MRG is equal to the intersection of a lattice L_t with the unit hypercube $[0,1)^t$ [5]. This implies that Ψ_t lies on a limited number of equidistant parallel hyperplanes, at a distance d_t apart, where $1/d_t$ turns out to be equal to the Euclidean length of the shortest nonzero vector in the dual lattice of L_t, defined as the set of vectors in $\mathbb{R}^t$ whose scalar product by any vector of L_t is an integer. Computing d_t is called the spectral test [14]. For Ψ_t to be evenly distributed over $[0,1)^t$, we want d_t to be small. Here, we use the same figure of merit as in L'Ecuyer (1999a), namely

$$M_{t_1} = \min_{t \le t_1} d_t^*(m^k)/ d_t,$$

where $t_1 > k$ is a selected constant (the maximal dimension that is considered) and $d_t^*(m^k) = 1/(\rho_t m^{k/t})$ is an absolute lower bound on d t, for given k and t . For $t \le 8$, we take ρ_t as the value of γ_t defined in Knuth, page 109, whereas for $t > 8$, we take $\rho_t = \exp[R(t)/t]$ where R(t) is the bound of Rogers on the density of sphere packing [35][21]. This M_{t_1} is always between 0 and 1 and we want it to be as large as possible. We recall that a general upper bound on $1/d_t^2$ is given by

$$1/d_t^2 \le 1 + \sum_{i=1}^{k} d_t^2 \,,$$

which means that a necessary condition for good quality is that the sum of squares of the coefficients must be large.

4.4.1 Implementation by the powers of 2- decomposition method

We want to compute

$$y = 2^q \; x \bmod m \; \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots.. (11)$$

where 0 < x < m. We decompose m and x as $m = 2^e - h$ and $x = x_0 + 2^{e-q} x_1$, where h> 0, x_0 = x mod 2^{e-q}, and x_1 == x/ 2^{e-q}. We then have

$$y = 2^q \; (x_0 + 2^{e-q} \; x_1) \bmod (2^e - h)$$

$$= (2^q x_0 + h x_1) \bmod (2^e - h)$$

We assume that the following inequalities hold:

$$h < 2^q \text{ and } h(2^q - (h + 1)\, 2^{-e+q}) < m.$$

Under these conditions, each of the two terms $2^q x_0$ and hx_1 in (11) is less that m, and y can be computed as follows [10]: Shift the binary representation of x_0 by q positions to the left to obtain $2^q \; x_0$, add h times x_0, and subtract m if the result exceeds m− 1. This can be implemented using unsigned integers and the intermediate results will never exceed 2m− 1. The procedure requires a single mu ltiplication, between h and x_1. To multiply x by a =$\pm\, 2^q \pm 2^r$ modulo m, repeat the procedure with r instead of q, and add (or subtract) the results modulo m. and it is implemented in C. Note that if q = 0 in (11), nothing needs to be done (y = x), whereas if q = 1, one can simply add x + x, and subtract m if the result exceeds m– 1. We will exploit these special cases when we will select the parameters of our combined MRGs.

Wu [9] introduced this method for the special case where h = 1. In this case, one obtains $y = 2^q \; x_0 + x_1$, which means that the binary representation of y is obtained simply by exchanging the blocks of bits x_0 and x_1 in the binary representation of x , i.e., rotating the bits by q positions. This simple rotation does not change the bits of x very much. The Hamming weights of x and of ax mod m (i.e., the number of 1’s in their respective binary representations) tend to be strongly dependent when m and a have the form $m = 2^e - 1$ and a =$\pm\, 2^q \pm 2^r$. For k = 1 (i.e., LCGs), they showed that this dependence also appears between the number of 1’s in the binary representations of two successive output values u_{n-1} and of u_n, and they proposed a simple statistical test to detect it. The specific LCGs proposed by Wu fail this independence test decisively, whereas LCGs whose multipliers have a more complicated binary representation typically pass this test. LCGs with multipliers of the form a =± 2^q ±2^r have in fact been proposed and used a long time ago: The infamous RANDU generator [36] has indeed a = 2^{16} + 2+ 1 and m = 2^{32}. These parameters were selected for the ease of implementation, but led to important deficiencies in the generator's structure.

4.5 Search for good parameters

We performed computer searches to find good single and combined MRGs that can be implemented via the powers- of-2 decomposition method. The first search was for MRGs of order k = 6, with modulus m = 2^{31}− 1. With this m, we have h = 1 and we thus avoid the multiplication by h: We are back to the special case considered by Wu. We imposed the condition that each coefficient a_i had to be of the form

a =± 2^q± 1, or a =± 2^q, or a = 0. ……….. (11)

Even with these conditions, an exhaustive search would be too long, so we made a random search. Almost all of the generators that we examined and that satisfied these conditions had a very bad lattice structure in dimension 7, 8 or 9. These generators typically had a good behavior in higher dimensions. The best generator that we found, based on the criterion M_{16}, has M_{16} = 0.25012. This is not very good. Its coefficients are a_1 = 2^{15}, a_2 = 0, a_3 =− 2^9 + 1, a_4 = 2^{20} − 1, a_5 =− 2^6 − 1, and a_6 = 2^{26} − 1. We call it MRG31k6s. It has 5 non-trivial powers of two in its coefficients, so it can be compared to MRG31k3p, to be presented in the next section, in terms of the number of multiplications by powers of two. We cannot recommend it, however, because its lattice structure is relatively poor and, perhaps more importantly, because the small number of powers of 2 in the coefficients means (intuitively) that there is not much mixing of the bits, similar to, e.g., the generator of Wu.

In our second search, with the same k and m, we allowed coefficients with more non-trivial powers of 2. The condition on the coefficients was that they had to be of the form a =± 2^q ± 2^r. MRGs with good lattice structures were much easier to find under these relaxed conditions. The best generator that we found, based on M_{16}, has M_{16} = M_{48} = 0. 59149. We call it MRG31k6l. Its coefficients are a_1 = 2^{23} + 2^{16}, a_2 = 2^{19} − 2^{12},

$a_3 = 2^{27} + 2^{15}$, $a_4 =- 2^{10} - 2^7$, $a_5 =- 2^4 - 1$, and $a_6 = 2^{27} + 2^{16}$. Of course, this generator will be slower than MRG31k3s, because there is more multiplications by powers of 2 to perform. We will compare their speeds at the end of the next section. Our third search was for combined MRGs with 2 components of order k = 3, with moduli $m_1 = 2^{31} - 1$ and $m_2 = 2^{31}- 21069$, and with some coefficients equal to zero and the others of the form $\pm 2^q$ or $\pm 2^q \pm 1$. We performed a random search. For each coefficient of each component, we specified the desired form: either 0, or $\pm 2^q$,or $\pm 2^q \pm 1$. Each coefficient received randomly between 1 to 10 possible values, depending on the specified form. We retained only the coefficients, for which the inequalities (11) were satisfied, and only the recurrences (or characteristic polynomials) that satisfied the maximal period conditions. We then examined all the possible combinations of one MRG component of each type, among those retained, to find out which combined generator performed best on spectral test. These combined MRG have approximatively the same period length as the MRGs of order 6 in our first two searches. The best combined MRG that we found, MRG31k3p, is described in the next section.

4.5.1 A specific generator and some timing

In our third search, we found the following combined MRG with J = 2 components of order k = 3, whose lattice structure is good at least up to 48 dimensions, with M_{48} = 0.60159. The two components are defined by the parameters

$$m_1 = 2^{31} - 1 = 2147483647$$
$$a_{11} = 0$$
$$a_{12} = 2^{22}$$
$$a_{13} = 2^7 + 1$$
$$m_2 = 2^{31} - 21069 = 2147462579$$
$$a_{21} = 2^{15}$$
$$a_{22} = 0$$
$$a_{23} = 2^{15} + 1.$$

Thus, each component has only 2 nonzero coefficients, one of them of the form $a_{ij} = 2^q$ and the other one of the form $a_{ij} = 2^q + 1$. This will simplify the implementation. The combination (4.2) is exactly equivalent to an MRG of order 3 with parameters

$m = 4611640770946945613$

$a_1 = 4341088847531259234$

$a_2 = 2349160800583431525$

$a_3 = 3927818590467337243$.

```
y1 = (((s11 & mask12) << 22) + (s11 >> 9)) + (((s12 & mask13) << 7) + (s12 >> 24));

if (y1 > m1) then y1 -= m1;

y1+= s12;

if (y1 > m1) then y1 -= m1;

s12 = s11; s11 = s10; s10 = y1; /* Second component */

y1 = ((s20 & mask21) << 15) + 21069 * (s20 >> 16);

if (y1 > m2) then y1 -= m2;

y2 = ((s22 & mask21) << 15) + 21069 * (s22 >> 16);

if (y2 > m2) then  y2 -= m2;

y2 += s22;

if (y2 > m2) then y2 -= m2;

y2 += y1;

if (y2 > m2) then y2 -= m2;

s22 = s21; s21 = s20; s20 = y2; /* Combination */

if (s10 <= s20) then  return ((s10 - s20 + m1) * norm);
```

Figure 4.2: Implementation of CMRG the Powers-of-2 Decomposition method

This generators has 2 distinct cycles of length ρ = $m_1 m_2$ / 2 ≈ 2^{185}. Figure 4.2 gives an implementation of this generator in the C language. In this code, the bit masks mask* are used to separate the bits of the $s_{j,i}$'s in 2 blocks as explained in Section 4.4.1. For example, mask12 contains 23 zeros followed by 9 ones. So, in the first line of the code for the first component, the statement

((s11 & mask12) << 22) + (s11 >> 9)

extracts the 9 least significant bits of x11, shifts them to the left by 22 positions, and adds this to s11 shifted to the right by 9 positions. The result is the product of s11 by a_{12} = 2^{22}, modulo m_1 = $2^{31} - 1$. The other products are implemented in a similar way. The several instructions of the form

if (y2 > m2) y2 -= m2;

are necessary to avoid overflow. Indeed, since the terms added are between 0 and $2^{31} - 1$ and since unsigned integers cannot exceed $2^{32} - 1$, we cannot safely add more than 2 terms at a time without reducing the sum modulo a_j.

To have an idea of the speed improvement of this new generator over the previous combined MRG implementations, for each generator we generated 10 million (10^7) random numbers and added them up, looked at how much CPU time it took (user time + system time), and printed the sum (this may be convenient for checking correctness of an implementation). In all cases, each integer in the seed was 12345.

Chapter 5

Results

These figures have been shown the timing for corresponding any arbitrary numbers. The timings for the C++ compilers are practically identical to those with the corresponding C compiler.

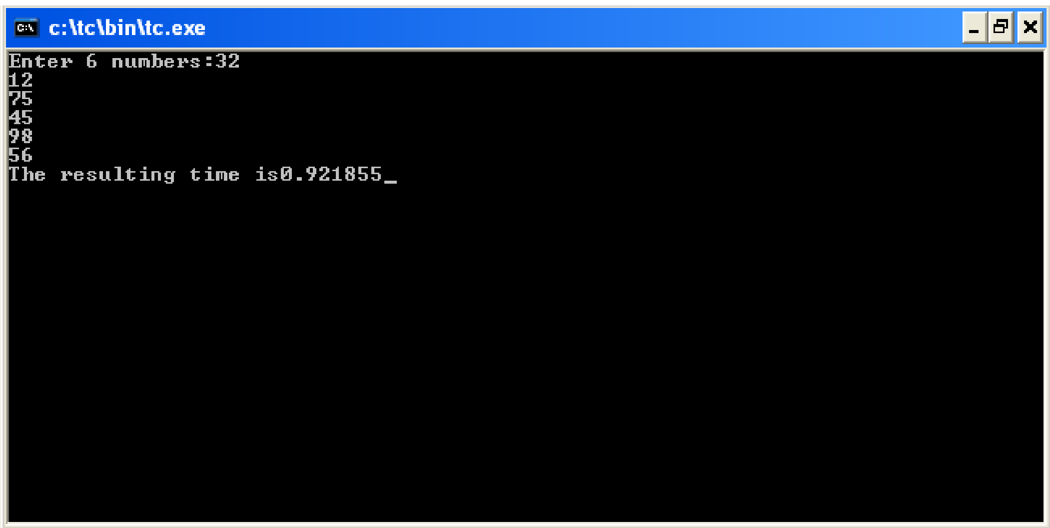

Figure 5.1(a): Resulting time for integers

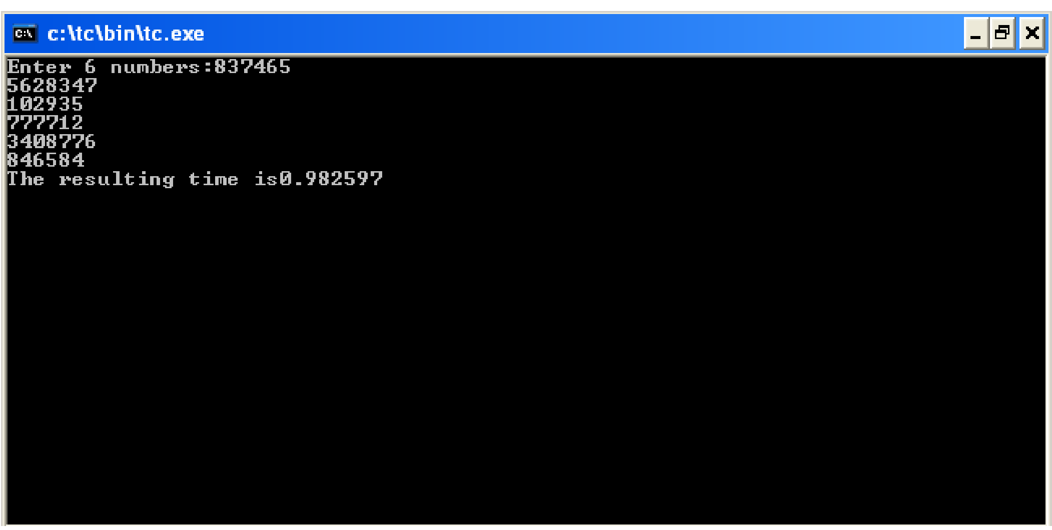

Figure 5.1(b): Time resulting for long numbers

The period length is indicated, the type of implementation (AF for approximate factoring, FP for floating-point, and P2D for powers-of-2 decomposition), and the sum of the 10^7 numbers generated. The generator MRG31k3p is that of Figure 4.2, MRG31k6s and MRG31k6l were introduced in the previous section, MRG32k3a is the combined MRG proposed and combMRG96b is a variant of combMRG96a with the moduli and multipliers defined as constants in the code instead of variables as in combMRG96a. Aside from MRG31k3s, which we discard because of its poor performance in the spectral test. This illustrates the fact that speed comparisons depend heavily on compilers and machine architecture. The generator of figure 4.2 gives only 31 bits of precision even though it returns 53-bit floating-point numbers. If more precision is desired, one can combine two successive numbers produced by the generator to construct each output value. For example, if MRG31k3p outputs the sequence u_1 , u_2 ,..., one can use the sequence v_1 , v_2 ,.. of pseudo random numbers defined by v_i = (νu_{2i} + u_{2i-1}) mod 1 for some constant ν between 2^{-21} and 2^{-32}.

Chapter 6
Conclusion & Future Scope

6.1 Conclusion:

Combined MRGs with multipliers of the form $\pm\ 2^q\ \pm\ 2^r$ are the fastest good-quality MRGs available to date, when comparing generators having approximatively the same period length. These combined MRG possess good theatrical properties in terms of their period length and the quality of their lattice structure, and behave well in empirical statistical tests.

6.2 Future Scope:

In the future, to search for good generators of this form by applying the spectral test not only to the vectors of successive output values produced by the generator (as usual), but to certain vectors of non-successive output values as well, as suggested by L'Ecuyer and Lemieux in the context of selecting lattice rules for quasi-Monte Carlo integration. These combined MRGs could also be combined with small, efficient, nonlinear generators to destroy the (linear) lattice structure and we intend to analyze such combinations.

References

[1] L'Ecuyer P. 1996. "Combined multiple recursive random number generators, Operations Research", 44(5):816– 822.

[2] Deng, Lih-Yuan, and Dennis K.J. Lin. "Random Number Generation for the New Century". The American Statistician, May 2000, Vol. 54, No. 2. pp. 145-150.

[3] L'Ecuyer P. and C. Lemieux, "Variance reduction via lattice rules", Management Science 2000.

[4] Niederreiter H, "Random number generation and quasi-monte carlo methods", Volume 63 of SIAM CBMS- NSF Regional Conference Series in Applied Mathematics. Philadelphia: SIAM 1999.

[5] L'Ecuyer P. and R. Couture. "An implementation of the lattice and spectral tests for multiple recursive linear random number generators". INFORMS Journal on Computing1997, 9(2):206–217.

[6] Eichenauer-Herrmann, Jurgen, "Inversive Congruential Pseudorandom Numbers: a Tutorial", International Statistical Review, August 1992, Vol. 60, No. 2. pp. 167-176.

[7] L'Ecuyer P. and S. Cote, "Implementing a random number package with splitting facilities". ACM Transactions on Mathematical Software1991, 17(1):98–111.

[8] Fischer, Viktor, and Milos Drutarovsky "True Random Number Generator Embedded in Reconfigurable Hardware ". Proceedings of the Cryptographic Hardware and Embedded Systems Workshop, 2002, pp. 415-430.

[9] Wu, P.-C "Multiplicative, congruential random number generators with multiplier $\pm 2^{k_1} \pm 2^{k_2}$ and modulus 2^p-1", ACM Transactions on mathematical Software1997, 23(2):255-265.

[10] L’Ecuyer P. and R. Simard, “Beware of linear congruential generators with multipliers of the form a =± 2^q ± 2^r”, ACM Transactions on Mathematical Software, 1999, 25(3):367–374.

[11] Gentle, James Z, “Random number generation and Monte Carlo methods”, Springer, New York, 2003.

[12] Golomb, Solomon W, “Shift Register Sequences” Aegean Park Press, 1982.

[13] L’Ecuyer P, F Blouin, and R Couture, “A search for good multiple recursive random number generators”, ACM Transactions on Modeling and Computer Simulation, 1993, 3(2):87–98.

[14] Knuth, Donald E, “The Art of Computer Programming, Volume 2: Seminumerical Algorithms (second edition)”, Addison Wesley, 1997.

[15] Lehmer D.H, “Mathematical methods in large-scale computing units. Proceedings of the Second Symposium on Large Scale Digital Computing Machinery”, pages 141-146, Harvard University Press, 1951.

[16] Lewis T.G., and Payne, W.H, “Generalize Feedback Shift Register Pseudorandom Number Algorithm”, Journal of the ACM, July 1973, Vol. 20, No. 3. pp. 456-468.

[17] Lidl, Rudolf and Harald Niederreiter, “Finite fields ”, Cambridge University Press, Cambridge, 1997.

[18] Random Number Generators web page
http://www.math.utah.edu/~pa/Random/Random.html

[19] Quasi-Random Number web page http://www.taygeta.com/rwalks/node3.html

[20] L’Ecuyer, P, ”Bad lattice structures for vectors of non- successive values produced by some linear recurrences”, INFORMS Journal on Computing, 1997, 9(1):57–60.

[21] L’Ecuyer, P, “Tables of linear congruential generators of different sizes and good lattice structure”, Mathematics of Computation, 1999b, 68(225):249–260.

[22] Metcalfe, Andrew V, "Statistics in management science", Arnold, London, 2000.

[23] Morgan, Byron J.T, "Elements of Simulation", Chapman and Hall, New York, 1984.

[24] NIST, "A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications", NIST Special Publication 800-22, National Institute for Standards and Technology, 2000.

[25] Schumer, Peter D, "Introduction to Number Theory", PWS Publishing Company, Boston, 1996.

[26] Talamba, Sorin, "A Theoretical and Empirical Study of Uniform Pseudo-Random Number Generators", Middlebury College Thesis, Department of Computer Science, 2001.

[27] Tausworthe, Robert C, "Random Numbers Generated by Linear Recurrence Modulo Two", Mathematics of Computation, April 1965, Vol. 19, No. 90. pp. 201-209.

[28] Watson, E.J, "Primitive Polynomials (mod 2)", Mathematics of Computation, July 1962, Vol. 16, No. 79. pp. 368-369.

[29] Wichman, B.A and I.D. Hill, "Algorithm AS 183: An Efficient and Portable Pseudo-Random Number Generator", Applied Statistics, 1982, Vol. 31, No. 2. pp. 188-190.

[30] Zeisl, H, "A Remark on Algorithm AS 183: An Efficient and Portable Pseudo-Random Number Generator", Applied Statistics, 1986, Vol. 35, No. 1. p. 89.

[31] R. Panneerselvam, "Design and Analysis of Algorithms", pp. 280-283.

[32] L'Ecuyer, P, "Good parameters and implementations for combined multiple recursive random number generators", Operations Research, 1999a, 47(1):159–164.

[33] L'Ecuyer, P and R. Couture, "An implementation of the lattice and spectral tests for multiple recursive linear random number generators", INFORMS Journal on Computing, 1997, 9(2):206–217.

[34] Bratley, P., B. L. Fox, and L. E. Schrage, ”A guide to simulation”, Second Ed, New York: Springer-Verlag, 1987.

[35] Conway, J. H. and N. J. A. Sloane, “Sphere packing, lattices and groups”, 3rd ed, Grundlehren der Mathematischen Wissenschaften 290, New York: Springer-Verlag, 1999.

[36] IBM, “System/360 scientific subroutine package”, Version III, Programmer’s Manual, White Plains, New York, 1968.

[37] Law, A. M and W. D. Kelton, “Simulation modeling and analysis”, Third ed, New York: McGraw-Hill, 2000.

[38] L’Ecuyer, P, “Uniform random number generation”, Annals of Operations Research, 1994, 53:77–120.

List of Publications

Communicated:

Prasada Rao Gurubill, Dr. Deepak Garg "Better technique of random number generation", National Conference on Emerging Trends in Engineering and Sciences (NCETES-2010).