NATIONAL LIBRARY
OTTAWA

CANADA

BIBLIOTHÈQUE NATIONALE
OTTAWA

NAME OF AUTHOR........ROBERT JAKES..............

TITLE OF THESIS. A NEW CONCEPT OF DIGITIZED....
UNIFORM PSEUDORANDOM NUMBER...
GENERATION..............................

UNIVERSITY...UNIVERSITY OF ALBERTA..............

DEGREE FOR WHICH THESIS WAS PRESENTED.....M.Sc............

YEAR THIS DEGREE GRANTED...SPRING..., 1974.............

Permission is hereby granted to THE NATIONAL LIBRARY OF CANADA to microfilm this thesis and to lend or sell copies of the film.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

(Signed).....R.W. Jakes......

PERMANENT ADDRESS:

#408....135 TYNDALL AVE
TORONTO 156, ONTARIO
M6K 2G4........

DATED...................19

THE UNIVERSITY OF ALBERTA

A NEW CONCEPT OF DIGITIZED UNIFORM PSEUDORANDOM NUMBER GENERATION

by

© ROBERT JAKES

A THESIS
SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
AND RESEARCH IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE
DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTING SCIENCE

EDMONTON, ALBERTA
SPRING, 1974

THE UNIVERSITY OF ALBERTA
FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled A NEW CONCEPT OF DIGITIZED UNIFORM PSEUDORANDOM NUMBER GENERATION submitted by Robert Jakes in partial fulfillment of the requirements for the degree of Master of Science.

....
Supervisor

Artur Wouk
....

....

Date November 16, 1973

ABSTRACT

A history of uniform pseudorandom number generation is presented from the experiments of W.S. Gossett up to the present day. A new concept of digitized uniform pseudorandom number generator (due to K.V. Leung) is described. Various models developed from the original concept have been implemented and tested for comparison with existing systems. These tests produce results showing that the new generators appear to have some statistical merit. One practical application of uniform pseudorandom number generation is given.

ACKNOWLEDGEMENTS

To Dr. K.V. Leung, I express my appreciation and thanks for his patience, advice and guidance throughout the preparation of this thesis. Also, I wish to thank my wife for being very patient and understanding during the course of my research.

TABLE OF CONTENTS

LIST OF TABLES

CHAPTER 1

HISTORY OF RANDOM NUMBER GENERATION

1.1 Pre-Computer

The generation of sequences of random numbers from the uniform distribution is a problem with a great historical background. Gossett (1908) was probably the first person to actually use random numbers — his general method consisted of taking a correlation table of heights and left middle finger measurements of 3000 criminals (see MacDonell (1901)). The table entries were written on 3000 pieces of cardboard, shuffled, and drawn at random (with replacement) four at a time. The resulting sequence of random numbers was used in formulating what is now known as Student's t distribution. However, his "random number generator" was extremely slow, and it was hard to say when the 3000 pieces of cardboard had been well shuffled. Tippett (1927) replaced this system by a table of 10000 four-digit numbers, which had been formed by taking 40000 digits "at random" from census tables, and grouping these in fours to obtain the 10000 numbers. Fisher and Yates (1938) improved on this by producing a table of 15000 random sampling numbers.

Kendall and Babington-Smith (1938) were the first to successfully generate random digits by mechanical means. They used a disc which was rotated by an electric motor at a rapid rate in a darkened room. This disc was divided into ten equal sections on which appeared the digits 0 - 9 inclusive. An electric spark illuminated the disc instantaneously, so as to make the disc appear stationary for a moment. When the spark occurred, a number was selected from the disc by means of a fixed pointer. The intervals between sparks were varied by means of a neon lamp in parallel with a condensor in an independent electrical circuit. A key tapped by an observer broke the circuit intermittantly at irregular intervals, to further add to the randomness. A table of random numbers generated by this mechanism was published in 1939 — this satisfied the criteria for random numbers until the late 1940's. At this time, computers were introduced — this made the use of tables of random numbers undesirable mainly due to the large amount of space required for storing tables. In order to be able to use random digits, it was necessary to derive some method of generating the numbers within the computer itself. Two basic types of generators, physical and arithmetic, have been developed for internal computer use. Physical generators, which basically consist of some external device which delivers a series of random pulses to the computer to be converted into random numbers, are discussed in some detail by Scott (1967), and will not be elaborated on here. Arithmetic generators, which form the basis for the majority of computerized random number generators, will be discussed in the following section of this chapter.

1.2 Arithmetic Generators

Arithmetic generators all have the same basic property — a sequence of digits is manipulated using a series of arithmetic operations to form a second sequence of digits. Part of this new sequence is retained as a new random number. The major advantage with this type of generator is that any sequence of random numbers can be reproduced exactly, if necessary, for checking purposes. It should be noted that all the numbers in a sequence of random numbers obtained from an arithmetic generator can be "predicted" — therefore, each number in such a sequence is actually a pseudo-random number.

The first arithmetic generator was developed in 1946 by von Neumann and Metropolis. Their method, called "mid-square", was as follows:

a) Define an n-digit number x_i.

b) Square this number to give a 2n digit product x_i^2.

c) Retain the middle n digits as x_{i+1}, and repeat the process.

For example, consider $n = 2$, and $x_0 = 63$. Then we obtain the following:

$$x_0^2 = 3969 \quad x_1 = 96 \quad x_1^2 = 9216 \quad x_2 = 21 \quad x_2^2 = 0441 \quad x_3 = 44 \ldots$$

This generator was proven to be unsatisfactory, as the length of a cycle (the length of a non-repeating series) depends on the value of x_0, and some initial values of x_0 lead to extremely short cycles. For example, consider $n = 4$, and $x_0 = 3600$. The generated sequence becomes:

$$x_1 = 9600 \quad x_2 = 1600 \quad x_3 = 5600 \quad x_4 = 3600 \quad x_5 = 9600 \ldots$$

Another drawback is that certain starting values x_0 lead to a non-repeating series of length 1 (for example, $n = 4$, and $x_0 = 1000$). Apart from these problems, the mid-square generator has been shown to not give a true uniform distribution of digits regardless of the value of x_0.

Forsythe (1951) described an improved version of the "mid-square" generator. It is known as the "mid-product" generator, and is given as follows:

a) Define two n-digit numbers x_{i-1}, x_i.

b) Multiply them together to obtain a $2n$ digit product.

c) Retain the middle n digits of this product as

x_{i+1}, and repeat the process.

For example, consider $n = 4$, $x_0 = 1234$, $x_1 = 5678$. Then we obtain the following:

$$x_2 = 0066 \quad x_3 = 3747 \quad x_4 = 2473 \quad x_5 = 2663 \ldots$$

This method may produce a much longer cycle than the mid-square method — note that both x_0 and x_1 must occur again consecutively before the cycle can repeat. However, even this method has starting values x_0 and x_1 that lead to a non-repeating series of length 1. For example, $n = 4$, $x_0 = 0100$, and $x_1 = 0500$ leads to the following series

$$x_2 = 0500 \quad x_3 = 2500 \quad x_4 = 2500 \quad x_5 = 2500 \ldots$$

Lehmer (1949) proposed a generator that not only eliminated degenerating sequences, but also guaranteed a non-repeating cycle of maximum length. His method, called "multiplicative congruential", consists of forming a sequence of pseudo-random numbers according to the formula:

$$x_{i+1} = ax_i \text{ (Mod M)}$$

for given a and x_0. The maximal length for a cycle is ensured by the following restrictions:

a) x_0 must be relatively prime to M .

b) a is a primitive root of p^α , if p^α is a factor of M , with p odd and α as large as possible.

c) a belongs to $2^{\alpha-2}$ if 2^α is factor of M with $\alpha \geq 2$. Moreover, for any value of M , there exist values of a satisfying these conditions, and finally, the maximal period is the lowest common multiple of the periods $(p-1)p^{\alpha-1}$ or $2^{\alpha-2}$ with respect to the prime power factors.

Proof of the above statements is given in Hull and Dobell (1962). This type of generator has been widely used, and has passed a number of tests for randomness. Marsaglia (1968) found, however, that "all the pseudo-random points generated in the unit n-cube by a multiplicative congruential generator will be found to lie in a relatively small number of parallel hyperplanes." To illustrate, suppose that the points $x_i, x_{i+1}, \ldots$ are normalized to be $U(0,1)$ numbers $x_i^{(N)}, x_{i+1}^{(N)}, \ldots,$ and that $\pi_i = (x_i^{(N)}, x_{i+1}^{(N)}, \ldots, x_{i+5}^{(N)})$ $i = 1,7,13,\ldots,$ be independent points on the unit 6-cube. Marsaglia showed that if all of the points were generated by a multiplicative congruential generator on a 32-bit word size computer such as the IBM 360/67, then all the points π_i would fall on less than 120 parallel hyperplanes (Marsaglia's theorem states that the number of hyperplanes is less than or equal to $(n!M)^{\frac{1}{n}}$]. In other words, multiplicative congruential generators suffer from lack of multidimensional uniformity.

Thompson (1958), Coveyou (1960), and Rotenberg (1960) all proposed variations to Lehmer's method, which consisted of adding a constant c to the equation for the multiplicative congruential generator to obtain:

$$x_{i+1} = ax_i + c(\text{Mod } M) .$$

This variation is referred to as the mixed congruential method. The maximum length for a non-repeating cycle is ensured by the following restrictions:

a) c is relatively prime to M .

b) $a \equiv 1(\text{Mod } P)$ if P is a prime factor of M .

c) $a \equiv 1(\text{Mod } 4)$ if 4 is a factor of M .

Proof of the above statements can be found in Hull and Dobell (1962).

Peach (1961) shows that any congruential generator with $M = 2^k$ and with a non-repeating cycle length of 2^i $0 < i \leq k$, has subcycles of length 2^j $0 < j < k$ that are subject to definite patterns. He gives as an example, the generator

$$x_{i+1} = 9x_i + 13 \text{ Mod}(2^5) .$$

The series produced by this generator is:

0 13 2 31 4 17 6 3 8 21 10 7 12 25 14 11
16 29 18 15 20 1 22 19 24 5 26 23 28 9 30 27
0 13 ...

The first two lines are half-cycles of length 2^4 — it can be seen that corresponding numbers in the two half cycles differ by 2^4 or 16 . In fact, whatever the value of 2^i , the difference between corresponding numbers in two half-cycles is 2^{i-1} If quarter cycles are taken, the difference between corresponding numbers in four quarter cycles is a multiple of 2^{i-2}. In general, the difference between corresponding numbers in p "$\frac{1}{p}$" cycles is a multiple of $2^{i-\log_2 p}$ for $p = 2^1, 2^2, \ldots, 2^i$. Mixed congruential generators have the advantage over multiplicative congruential generators in that a non-repeating cycle of length M can be attained — however, this is done at a cost of poor statistical behaviour (see Hull and Dobell (1964)).

MacLaren and Marsaglia (1965) suggested combining two or more congruential generators to produce what is called a composite congruential generator. They give the following general procedure for two generators:

a) Consider the congruential generators

i) $x_{i+1} = ax_i \text{ Mod}(M)$

ii) $y_{i+1} = by_i + c \text{ Mod}(M)$.

b) Define initial values for x_0 , y_0 .

c) Fill up a table of 128 locations in core with the values of $x_1,\ldots,x_{128}$.

d) To generate z_k , the k^{th} random number to be generated, use the first seven bits of y_k as an index to get z_k from the table. Refill the location of z_k in the table, with the value of x_{128+k} .

MacLaren and Marsaglia successfully developed such a generator with $a = 2^{17} + 3$, $b = 2^7 + 1$, $c = 1$, and $M = 2^{35}$. They state that the time-taken to generate a random number is about twice that required by a single congruential generator, but that this is offset by a great improvement in statistical properties. Marsaglia and Bray (1968) give a simple algorithm for a composite generator using three multiplicative congruential generators, first defining integer values $N(1),\ldots,N(128)$, L , , and K . They then give the following FORTRAN subprogram for generating uniform random numbers U , on the interval $0 < U < 10^Q$ $(Q = 0,1,2,\ldots)$ using an IBM 360 computer:

```
L=L*ML
M=M*MM
J=1+IABS(L)/16777216
U=(.5*FLOAT(N(J)+L+M)*.23283064E-9)*10**Q
K=K*MK
N(J)=K
```

In this program, the integer J is used as an index for picking a value from N(1), to N(128); J comes from the value of L after division by 2^{24} (or 16777216). In forming U, the argument of the float instruction is the sum (Modulo 2^{32}) of the randomly chosen N(J), the random integer L used to find J, and a third gratuitous integer M. The random integers K, L, and M are the outputs of the three congruential generators, and the constants MK, ML, and MM are chosen to ensure long non-repeating cycles. Marsaglia and Bray claim to have obtained excellent results using ML=65539, MM=33554433, and MK=362436069.

Hutchinson (1966) reopened the case for using a multiplicative congruential pseudo-random number generator by suggesting one in which the value of M (the modulus) is the largest prime less than 2^k. He states that this type of generator a) passes the basic statistical tests for randomness, b) has a sufficiently long non-repeating cycle length, and c) does not have sub-cycles following certain patterns that other congruential generators have been shown to have. Scott (1967) gives the generator $x_{i+1} = 24x_i$ (Mod 31) as an example of Hutchinson's generator.

Tausworthe (1965) proposed a new uniform random number generator, based on irreducible primitive trinomials over the field of characteristic 2. Whittlesey (1968) describes the basic properties of a Tausworthe generator, and gives the following algorithm for implementation purposes.

1) Define N as one less than the number of bits per word, and M as less than N/2 (as chosen from the list of primitive trinomials).

2) Let register A initially contain the previous random number (say, Y) in bit positions 1 to N, with zero in the sign bit (position 0).

3) Copy register A into register B, and then right-shift register B by M places.

4) Exclusive-OR register A into register B, and also store the result back into register A (Registers A and B now have bits for the *new* random number in bit positions M+1 to N, but still contain bits from the old N-bit random number in bit positions 1 through M).

5) Left-shift register B by N-M positions. (This places M bits for the new random number in bit

positions 1 through M of register B , and zero bits in positio M+1 through N).

6) Exclusive-OR register B into register A , and zero out register A's sign bit (Register A now contains all n bits of the new random number Y').

This algorithm will generate all possible 2^N-1 non-zero N-bit numbers before it repeats any of these numbers for proper choices of M used with a particular N . For example, on the IBM360, where the single-precision word length is 32 bits (N=31) , M must be set equal to 3, 6, 7, or 13 to obtain the maximum length of the non-repeating cycle. Such a generator has been tested extensively in recent years. Whittlesey (1968) states that a Tausworthe generator has no serial correlation, and speed. The same author (1969) states that this generator does have multidimensional uniformity. Tootill, Robinson, and Adams (1971) have shown that as long as M is not too small, not too close to N/2 , or not greater than N/2 , this generator also has a good runs up and down performance. However, the Tausworthe generator suffers from the fact that for many computer word lengths N , the values of M producing non-repeating cycles of length $2^N - 1$ are not known (only Zierler and Brillhart (1969) have made any attempt to tabulate values of M for certain given values of N).

CHAPTER 2

DESCRIPTION OF NEW SYSTEM OF GENERATORS

2.1 Introduction

In the previous chapter, the development of uniform pseudorandom number generators was outlined, with special emphasis being placed on the so-called congruential system of generators, and the primitive polynomial based generator of Tausworthe. It is apparent, however, that the optimal use of any congruential or Tausworthe generator is dependent on the word-size in bits allowed on various computers. For example, the multiplicative congruential generator defined by:

$$x_{n+1} = 65539x_n \text{ (Modulo } 2^{31}\text{)}$$

can be used on an IBM360 model computer with 32-bit words, but not on a PDP series computer with 16-bit words. This problem can be solved by a generator that selects sufficiently random digits one at a time in order to form a number of a fixed digit length, without sacrificing a great deal of computer time in the process. It is the purpose of this chapter to describe a proposal for such a system, which will be shown in a later chapter to have good statistical behaviour, while using a minimal amount of computer storage.

2.2 Description of General System

The basic elements for the new system are as follows:

$$E = \{0,1,2,\ldots,9\}$$

$\vec{p}$ = pointer index vector,

z = column pointer,

$$A = \begin{bmatrix} a_{0,1} & a_{0,2} & \cdots & a_{0,n} \\ a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ \vdots & & a_{i,j} & \\ a_{9,1} & a_{9,2} & \cdots & a_{9,n} \end{bmatrix}_{10\times n}$$

where $a_{i,j}$ is a digit value located at the position defined by (i,j) such that $a_{i,j} \in E$, and, $a_{i,j} \neq a_{k,j}$ if $i \neq k$ for all $j = 1,\ldots,n$.

There are three elementary operations involved in the generating of each digit of a random number. They are as follows:

1. Select one digit from a column of A. The digit value is located at a position being referred to directly or indirectly by a pointer.

2. Define new value(s) for any pointers used.

3. Reorder the column of A from which the digit was selected.

There are many general procedures that could be derived for manipulating A in order to produce a sequence of random digits. Two such procedures, namely Sequential Single Arrays and Sequential Pair Arrays, have been analyzed in terms of the components given; both of these will be discussed in some detail in the succeeding sections.

2.3 Sequential Single Arrays

For this type of generator, the number of columns of A can be any integer greater than one. The pointer vector $\vec{p}$ has only one element p which is contained in E . The value of the column pointer z , if used, is in the range $1 \leq z \leq n$. To generate a random number of L digits, a digit Q is to be selected from column j of A . This can be done by one of the following three methods.

a. This method uses two columns, j^{th} and $[(j \text{ Mod } n) + 1]^{th}$, to select Q . If $J = a_{p,j}$ and $K = a_{J,(j \text{ Mod } n)+1}$, the value of Q will be

$$Q = a_{K,j} ,$$

while p is the value of the previously generated digit. The next digit will be selected from the $[(j \text{ Mod } n) + 1]^{th}$ column.

The reordering of the j^{th} column consists of interchanging the two elements $a_{p,j}$ and $a_{K,j}$ and then, using modulo 10 arithmetic, adding $Q + \delta_{Q,0}$ to each element in the j^{th} column, where

$$\delta_{Q,0} = \begin{cases} 0 & \text{if } Q \neq 0 \ , \\ 1 & \text{if } Q = 0 \ . \end{cases}$$

Q will become the new value of p ; to be used in generating the next digit.

The complete algorithm for generating a L digit random number using this method is given as follows:

<u>Algorithm I</u>

i) Set the digit index N to 1 . Q j will be selected from the j^{th} column of A .

ii) Set $J = a_{p,j}$.

iii) Set $K = a_{J,(j \text{ Mod } n)+1}$, and $Q = a_{K,j}$.

iv) Interchange $a_{p,j}$ with $a_{K,j}$.

v) Set the N^{th} digit of the number equal to Q .

vi) Replace $a_{m,j}$, $m = 0,1,\ldots,9$, by $(a_{m,j} + Q + \delta_{Q,0})$ Mod 10 .

vii) If N is equal to L , the generation of the number is completed. Otherwise,

viii) Set $N = N+1$, $j = (j \text{ Mod } n)+1$, and go to step ii).

b. This method uses only one column of A to select each digit Q . Its value will be

$$Q = a_{p,j} ,$$

where p is the value of the previously generated digit.

The reordering of the j^{th} column is done by interchanging the two elements $a_{p,j}$ with $a_{9-p,j}$ and then using modulo 10 arithmetic to add $Q + \delta_{Q,0}$ to each element in the j^{th} column. Q will also be the new value of p . The complete algorithm for this method can be derived from Algorithm I by deleting steps ii) and v) and replacing steps iii) and iv) by:

iii) Set $Q = a_{p,j}$ and set the N^{th} digit of the number equal to Q .

iv) Interchange $a_{p,j}$ with $a_{9-p,j}$.

The expression $\delta_{Q,0}$ is defined as before.

c. This method uses the j^{th} column, and another column referred to by a column pointer z , to select the digit Q . Its value is

$$Q = a_{a_{J,z},j}$$

where J is the same as described in method a., and z has also been defined while generating the previous digit. The new value of z is

$$\left[z + Q\left(1 - \delta_{Q\mathrm{Mod}\ n,n-1}\right)\right]\mathrm{Mod}\ n + 1 \ .$$

The reordering of the j^{th} column is done by interchanging the two elements

$$a_{p,j}$$

and

$$a_{a_{J,z},j}$$

and using modulo 10 arithmetic to add $Q + \delta_{Q,0}$ to each element of the j^{th} column. Q will be the new value of p.

The expression $\delta_{QMod\ n,n-1}$ is defined as:

$$\delta_{QMod\ n,n-1} = \begin{cases} 0 & QMod\ n \neq n-1 \\ 1 & QMod\ n = n-1 . \end{cases}$$

The expression $\delta_{Q,0}$ is similarly defined.

The complete algorithm can be derived from Algorithm I by replacing step iii) by

iii) Set $K = a_{J,z}$, $Q = a_{K,j}$. Replace z by $[z + Q(1 - \delta_{QMod\ n,n-1})]\text{Mod } n + 1$.

If the generator is being used for the first time, the matrix A, and the pointers p and z must be assigned initial values by the user, and the starting value of j is 1. Otherwise they have been defined from a previous run.

2.4 Sequential Pair Arrays

For this type of generator, the number of columns in A is $n = 2m$ $(m = 1,2,\ldots)$. The pointer vector $\vec{p}$ has m components, each of which has a value ϵE. The column pointer z has a value in the range $1 \leq z \leq m$. In some cases, a second column pointer $\bar{z}$ may be required, which has the same range as z . All of these pointers have been assigned values during generation of the previous digit.

In order to generate a random digit, each of the m adjacent column pairs, starting with the first two columns, are manipulated before actually choosing a digit. This suggests modifying the third elementary operation to read:

> 3. Reorder each of the non-overlapping column pairs of A before selecting a digit.

The value of the digit selected then depends on the value of the column pointer z , and an element of the vector $\vec{p}$.

As with Sequential Single Arrays, three methods have been developed for generating a random number of length L digits. They are described as follows:

a. Suppose that:

$$[a_{i,j}] \quad i = 0,\ldots,9 \quad j = 2\ell - 1, 2\ell \quad \ell \text{ odd} \quad 1 \le \ell \le m$$

represents an adjacent column pair to be manipulated. Define $J = a_{p_\ell, 2\ell-1}$, and $K = a_{J,2\ell}$. The reordering of columns $2\ell-1$ and 2ℓ is done by:

a) interchanging $a_{p_\ell, 2\ell-1}$ with $a_{K,2\ell-1}$,

b) using modulo 10 arithmetic to add

$$\left(J - K - a_{K,2\ell-1}\right)^{1-\delta_{j+K+a_{K,2\ell-1},0}}$$

to each element in column $2\ell-1$,

c) interchanging columns $2\ell-1$ and 2ℓ .

The new value of the reference pointer for that column pair becomes $P_\ell = a_{K,2\ell}$. After all of the m adjacent column pairs have been manipulated in this way, the value of the selected digit will be

$$Q = a_{p_z,2z-1} .$$

The new value of the special pointer will be

$$z = (Q + z) \text{ Mod } m + 1 .$$

An algorithm to generate an L-digit number using the preceding principles is given on the following page.

Algorithm II

a) Let the digit index (say, N) be set equal to 1 .

b) Let an index for the adjacent column pairs (say, I) be set equal to 1 .

c) Set $J = a_{p_I,2I-1}$, $K = a_{J,2I}$, and $K1 = a_{K,2I-1}$.

d) Interchange $a_{K,2I-1}$ with $a_{p_I,2I-1}$.

e) Replace $a_{q,2I-1}$ $q = 0,\ldots,9$ by $a_{q,2I-1} + (J-K-K1)^{1-\delta_{J-K-K1,0}}$.

f) Interchange $[a_{t,2\ell-1}]$ with $[a_{t,2\ell}]$ $t = 0,1,\ldots,9$.

g) Replace P_I by K1 .

h) If $I \geq m$, then go to step j . Else

i) Increment I by 1 , and go to step c).

j) Set $Q = a_{p_z,2z-1}$, and thus, the N^{th} digit of the number being generated = Q .

k) If $N \geq L$, then stop. Else

l) Increase N by 1 , set $z = (z+Q)\text{Modulo}(m) + 1$, and go to step b).

b. Suppose that

$$[a_{i,j}] \quad i = 0,\ldots,9 \quad j = 2\ell-1, 2\ell \quad \ell \text{ odd} \quad 1 \leq \ell \leq m$$

represents an adjacent column pair to be manipulated. Define J to be the same as in method a, and K to be $a_{J,2\bar{z}}$. The reordering of columns $2\ell-1$ and 2ℓ is done by

a) interchanging $a_{p_\ell,2\ell-1}$ with $a_{K,2\ell-1}$,

b) using modulo 10 arithmetic to add

$$\left(J + K + a_{\text{[illegible]},2\ell-1}\right)^{1-\delta_{j+K+a_{K,2\ell-1},0}}$$

to each element in column $2\ell-1$.

c) interchanging column $2\ell-1$ with column 2ℓ.

The new value of the reference pointer for that column pair becomes:

$$p_\ell = a_{K,2\ell-1}.$$

The new value of $\bar{z}$ becomes:

$$\bar{z} = (\bar{z} + K)\text{Modulo}(m) + 1.$$

After all of the m adjacent column pairs of A

have been manipulated, the value of the selected digit will be:

$$Q = a_{p_z,2z-1}$$

The new value of z will be defined in the same manner as in the previous method. The algorithm to generate an L-digit number using the preceding principles is the same as that for the previous method, but with the following two changes:

i) Step c) is replaced by:

"c) Set $J = a_{p_I,2I-1}$, $K = a_{J,2\bar{z}}$, and $K1 = a_{K,2I-1}$."

ii) Step g) is replaced by

"g) Replace p_I by K1 , and $\bar{z}$ by $(\bar{z} + K)$ Modulo (m) + 1 .

c. Suppose that

$$[a_{i,j}] \quad i = 0,\ldots,9 \quad j = 2\ell-1,2\ell \quad \ell \text{ odd} \quad 1 \le \ell \le m \quad (m=10)$$

represents an adjacent column pair to be manipulated. Define J as in method a., and K to be $a_{p_{J+1},J+1}$.

The reordering of columns $2\ell-1$ and 2ℓ is now done using the same procedure as in the preceding two methods, and the value of the selected digit d (as well as the new values for any pointers used) are defined as before. An algorithm to generate an L-digit number using the preceding principles is the same as for method a., but with the following change:

step c) is replaced by:

"c) Set $J = a_{p_I, 2I-1}$, $K = a_{p_{J+1}, J+1}$, and $K1 = a_{K, 2I-1}$."

For all three Sequential Pair Array methods, the expression $\delta_{J+K+a_{K,2\ell-1},0}$ is defined in a manner similar to that as defined in Sequential Single Arrays Method 3.

2.5 Modifications

The methods described in Sections 2.3 and 2.4 can be modified in many ways, in order to attempt to improve the randomness of a generated sequence of random numbers. One possible modification is to add to the list of basic elements, the following selector matrix:

$$B = \begin{bmatrix} b_{11} & b_{12} \\ \vdots & \vdots \\ b_{L1} & b_{L2} \end{bmatrix}_{L\times 2}$$

where $b_{ij} \neq b_{kj}$ for $i \neq k$, and $1 \leq b_{ij} \leq L$.

Each time a sequence of L digits is generated by any of the previously described methods, the first column of B can be used as an index vector to re-order the L digits. The matrix B can then be modified as follows:

a) Interchange two random digits in the first column.

b) Replace b_{i1} by $\left[(b_{i1} - \text{a random value constant}) \text{ Modulo } L\right] + 1$ for $i = 1,2,\ldots,L$.

c) Interchange the two columns.

A further modification is to redefine the function $\delta_{Q,0}$ as:

$$\delta_{Q,0} = \begin{cases} 0 & Q \neq 0 \\ KZ & Q = 0 \end{cases}$$

where $KZ = K$ for Sequential Single Arrays Method 1

$a_{9-p,j}$ for Sequential Single Arrays, Method 2.

J for Sequential Single Arrays Method 3.

For definitions of K, $a_{9-p,j}$, and J , see the descriptions of the various methods involved. Similar modifications to the δ function can be made for the Sequential Pair Arrays methods.

CHAPTER 3

TESTING THE NEW CLASS OF PSEUDORANDOM NUMBER GENERATORS

3.1 Introduction - Description of Tested Generators

A series of tests (see Appendix A) were applied to each of the developed methods using different values of the matrix A . This was done in order to enable us to compare these methods with other already existing methods of uniform pseudorandom number generation. Some preliminary testing was done on the following generators:

a) Sequential single arrays, all three methods, with $n = 3$; $L = 2$, and 3. The generating process begins with:

$$p = 2 , \quad z = 1 , \quad \text{and}$$

$$A = \begin{bmatrix} 1 & 2 & 0 \\ 3 & 6 & 3 \\ 5 & 4 & 6 \\ 7 & 3 & 9 \\ 9 & 1 & 1 \\ 8 & 9 & 4 \\ 6 & 7 & 7 \\ 4 & 8 & 2 \\ 2 & 0 & 5 \\ 0 & 5 & 8 \end{bmatrix} \quad \text{or} \quad \begin{bmatrix} 1 & 2 & 0 & 7 \\ 3 & 6 & 3 & 2 \\ 5 & 4 & 6 & 6 \\ 7 & 3 & 9 & 8 \\ 9 & 1 & 1 & 4 \\ 8 & 9 & 4 & 1 \\ 6 & 7 & 7 & 5 \\ 4 & 8 & 2 & 9 \\ 2 & 0 & 5 & 0 \\ 0 & 5 & 8 & 3 \end{bmatrix}$$

b) Sequential pair arrays, all three methods, with $n = 20$ $(m = 10)$; $L = 2$ and 3.

The generating process begins with:

$$\vec{p} = \{2, 4, 6, 5, 7, 8, 4, 1, 8, 0\},$$

$z = 4$, $\bar{z} = 3$, and

$$A = \begin{pmatrix} 1&5&8&8&9&3&3&7&4&6&6&4&7&2&0&9&2&1&5&0 \\ 3&2&4&7&5&0&0&5&9&8&2&1&1&3&6&6&7&9&8&4 \\ 7&0&2&6&1&8&5&1&6&4&4&9&9&5&8&2&0&3&3&7 \\ 4&7&0&1&6&9&9&3&2&5&8&6&3&4&1&8&5&0&7&2 \\ 9&6&1&2&7&4&2&8&0&3&3&7&5&0&4&1&8&5&6&9 \\ 0&4&5&9&2&6&8&2&3&0&7&3&6&1&9&7&1&8&4&5 \\ 6&3&9&0&8&5&7&9&1&2&0&8&4&7&5&4&3&6&2&1 \\ 5&9&3&5&0&1&4&6&7&7&1&2&8&8&2&0&6&4&9&3 \\ 2&8&7&4&4&2&6&0&8&1&5&5&0&9&3&3&9&7&1&6 \\ 8&1&6&3&3&7&1&4&5&9&9&0&2&6&7&5&4&2&0&8 \end{pmatrix}$$

c) IBM Multiplicative Congruential generator (see May (1967)) based on:

$$x_{n+1} = 65539x_n \text{ (Modulo } 2^{31}\text{)}$$

$$x_0 = 123321 .$$

d) Composite Congruential Generator (see Marsaglia & Bray (1968)) based on:

$$x_{n+1} = 65539x_n \text{ (Modulo } 2^{31}\text{)}$$

$$y_{n+1} = 33554433y_n \text{ (Modulo } 2^{31}\text{)}$$

$$s_{n+1} = 362436069 s_n \left(\text{Modulo } 2^{31}\right)$$

$$x_0 = 13$$

$$y_0 = 15$$

$$s_0 = 17 .$$

e) Tausworthe Generator (see Whittlesey (1968)) based on:

$$n = 31$$

$$m = 13 .$$

f) Sequential single arrays, method 1 (modified with selector matrix and revised δ function), with $n = 3$ $L = 2 \left(B = \begin{matrix} 2 & 1 \\ 1 & 2 \end{matrix}\right)$ and $3 \left(B = \begin{matrix} 2 & 3 \\ 1 & 1 \\ 3 & 2 \end{matrix}\right)$. The generating process begins with other parameters defined as in a).

For each of c), d), and e), the L leftmost digits of each normalized number generated were taken as being the number desired for testing purposes.

3.2 Results

Tables 3.1 - 3.4 give a summary of the results of the tests described in Appendix A as applied to sequences of L-digit numbers generated by the random number generators of section 3.1. For Tables 3.1 and 3.2, a total of 100 sequences of L-digit numbers (10^L numbers per sequence) were generated, and the average of the various statistics calculated for each sequence were entered in the tables. For Table 3.3, a total of 100 non-overlapping sequences of 3-digit numbers (1000 numbers per sequence) were generated by the various generators. The sample covariances of lags {1, 2, 3, 4, 5, 7, 10, 15, 20} were calculated for each sequence. Table 3.3 shows the largest (in magnitude) positive and negative sample covariances for the various lags, for the 100 sequences of numbers tested from each generator. For the purpose of determining the serial correlation coefficients of Table 3.4, a total of 100 series of length x digits each were generated by each of the random number generators (using the leftmost digit of each number generated (normalized) in forming such a sequence from already existing generators). For each digit series length x given in Table 3.4, the correlation coefficient given for any particular generator is the average correlation coefficient from the 100 non-overlapping series (each of length x).

The following abbreviations are used in the tables for representing the various random number generators tested:

MC - IBM Multiplicative Congruential.

CC - Marsaglia & Bray Composite Congruential.

T - Tausworthe.

$M_3$1, $M_3$2, $M_3$3, $M_3$4 - Sequential Single Arrays, methods 1 to 3, and 1 (modified), with 3 column A matrix.

$M_4$1, $M_4$2, $M_4$3, $M_4$4 - As above, with 4 column A matrix.

P1, P2, P3 - Sequential Pair Arrays, methods 1 to 3.

Table 3.1 Statistical Test Results, L = 2

		Moments			χ^2_{100}	Runs Up and Down	Runs Above and Below Mean
		1st	2nd	3rd			
Expected Value		.500	.333	.250	100	66.33	51
GENERATOR	MC	.4952	.3276	.2439	100.6	66.6	51.07
	CC	.4933	.3259	.2425	99.56	66.3	50.94
	T	.4956	.3288	.2453	100.22	66.68	50.94
	$M_3$1	.4970	.3311	.2479	99.94	66.0	49.63
	$M_4$1	.4964	.3308	.2481	96.44	65.74	50.57
	$M_3$2	.4947	.3266	.2426	97.8	66.59	51.48
	$M_4$2	.4977	.3319	.2488	98.38	65.76	50.24
	$M_3$3	.4887	.3325	.2399	101.36	65.71	50.79
	$M_4$3	.4960	.3289	.2451	97.38	66.32	50.56
	$M_3$4	.4987	.3317	.2479	97.76	65.95	50.87
	$M_4$4	.4975	.3311	.2474	99.6	66.25	50.34
	P1	.487	.318	.233	94.4	64.3	52.2
	P2	.500	.336	.252	99.8	65.9	51.6
	P3	.490	.325	.244	95.8	67.5	52.7

Table 3.2 Statistical Test Results, L = 3

		Moments			χ^2_{1000}	Runs Up and Down	Runs Above and Below Mean
		1st	2nd	3rd			
Expected Value		.500	.333	.250	1000	666.33	501
GENERATOR	MC	.49864	.33163	.24823	1003.88	668.67	500.41
	CC	.49915	.33219	.24884	995.06	664.87	497.83
	T	.49976	.33300	.24962	998.06	668.47	501.10
	$M_3 1$	.49991	.33338	.25011	998.18	666.68	501.54
	$M_4 1$	.49897	.33199	.24853	999.48	665.87	501.7
	$M_3 2$	.50004	.33353	.25023	1004.62	664.97	501.39
	$M_4 2$	.49797	.33131	.24810	1000.34	664.19	498.7
	$M_3 3$	.50011	.33406	.25106	1024.24	668.97	502.03
	$M_4 3$	.49967	.33357	.25051	998.46	664.91	501.64
	$M_3 4$	.49902	.33221	.24890	1000.92	664.57	499.06
	$M_4 4$	.49877	.33222	.24905	1006.6	666.46	500.61
	P1	.4990	.33354	.24905	1009.81	666.85	505.7
	P2	.49126	.32211	.23871	1013.27	665.1	505.69
	P3	.49367	.32841	.24467	1015.63	662.41	486.32

Table 3.3 Sample Covariances

		Lag 1		Lag 2		Lag 3	
		maximum +	maximum −	maximum +	maximum −	maximum +	maximum −
	MC	.01901835	.02316471	.02081275	.02195925	.01632577	.01943672
G	CC	.01712263	.02396256	.02335012	.02164644	.01635188	.02371913
E	T	.01902288	.02292240	.02577430	.01882613	.01652348	.02515548
N	$M_3 1$	.02243072	.02377337	.02207267	.01726127	.01965195	.01952928
E	$M_4 1$	.02359688	.01376563	.01703537	.02621007	.01682222	.02395767
R	$M_3 2$	.01790768	.02136540	.02589786	.02305722	.01693791	.01848930
A	$M_4 2$	.02099341	.01959455	.01703322	.01776278	.02677888	.02191240
T	$M_3 3$	.02036148	.01666689	.02222055	.01843625	.02320236	.02408600
O	$M_4 3$	.01963347	.01908273	.02522922	.02036721	.01571190	.02069843
R	$M_3 4$	.01969737	.02117985	.02497673	.01835477	.02186227	.01938635
	$M_4 4$	.02192307	.02113056	.01601058	.02638024	.02189398	.01846695

		Lag 4		Lag 5		Lag 7	
		maximum +	maximum −	maximum +	maximum −	maximum +	maximum −
	MC	.02504158	.01871359	.01574999	.02426082	.02239877	.02208483
G	CC	.02848285	.02116770	.01964670	.02558970	.03082705	.02292681
E	T	.01531214	.02045643	.02378374	.02043569	.01809084	.02443165
N	$M_3 1$	.01764828	.02208018	.01547039	.02889735	.02064329	.02169812
E	$M_4 1$	.01752037	.02194905	.01802725	.01947057	.01784194	.03094923
R	$M_3 2$	.01794446	.01719046	.01737756	.02390039	.01473558	.03169173
A	$M_4 2$	.02000433	.02003938	.02368927	.02001894	.02196926	.02971965
T	$M_3 3$	.02610642	.02011329	.01686752	.02000713	.01655871	.02346992
O	$M_4 3$	.02239919	.01912284	.02615678	.01912671	.01425946	.02252066
R	$M_3 4$	.01700616	.02308607	.01962841	.02080506	.02080274	.02049083
	$M_4 4$	.01460218	.02495801	.03170270	.01951635	.01642436	.03121573

Table 3.3 Sample Covariances (continued)

		Lag 10		Lag 15		Lag 20	
		maximum +	maximum −	maximum +	maximum −	maximum +	maximum −
	MC	.01565522	.02593273	.01971841	.02038187	.01889032	.02547860
G	CC	.01545042	.02506441	.01741731	.02238601	.01984209	.02015865
E	T	.02286863	.02494532	.02054280	.02286071	.01867527	.02540678
N	$M_3 1$	.01963478	.03013986	.02538252	.01547837	.03304946	.02728808
E	$M_4 1$	.02797441	.02673942	.01659322	.02411312	.02920538	.02523404
R	$M_3 2$	.01835030	.02160770	.02035809	.02362400	.02064977	.02103829
A	$M_4 2$	.01909184	.02123976	.02388716	.02609020	.02170926	.02485698
T	$M_3 3$	.02375668	.02322906	.02153707	.02242440	.02369136	.02883750
O	$M_4 3$	.01713234	.02114284	.02059901	.02668661	.02349776	.03040141
R	$M_3 4$	.02602476	.02056301	.01924860	.02001607	.02292496	.02368999
	$M_4 4$	.02172583	.02028894	.02111971	.02435941	.02345341	.02558398

Table 3.4 Serial Correlation Coefficients

		Digit Series Length						
		200	600	1000	2000	5000	15000	30000
GENERATOR	MC	.0466417	.0151888	.0091332	.0046106	.0018439	.0006246	.0002878 (35)
	CC	.0447958	.0151032	.0091729	.0044792	.0018094	.0006039	.0003017 (34)
	T	.0454965	.0148421	.0087381	.0043835	.0018124	.0005823 (37)	
	$M_3 1$	.0452844	.0146433	.0087501	.0043828	.0017494	.0005872	.0002919 (9)
	$M_4 1$	.0433684	.0144784	.0087953	.0043342	.0017707	.0006022	
	$M_3 2$	.0454211	.0149993	.0090097	.0045992	.0017998	.0005913	.0003307 (9)
	$M_4 2$	.0451244	.0151776	.0089936	.0045773	.0017851	.0005946	
	$M_3 3$	.0467582	.0161885	.0103518	.0058430	.0031852	.0019437	
	$M_4 3$	.0445786	.0158209	.0096346	.0052842	.0025701	.0013488	
	$M_3 4$	.0448398	.0149859	.0090415	.0046055	.0017698	.0005849	.0002867 (6)
	$M_4 4$	.0450476	.0149749	.0088008	.0044189	.0017943	.0006021	

The numbers in brackets represent the number of trials used to obtain the serial correlation when machine limitations limited the number of trials to less than the 100 used for filling in the rest of the table.

Due to machine limitations, the results obtained for a digit series length of 30000 cannot be considered as conclusive, as in most cases, the number of trials involved was very small. In fact, whenever the number of trials was less than 5, the average serial correlation coefficient obtained was not used for the purpose of setting up Table 3.4.

3.3 Conclusions

The results of the tests as summarized in Tables 3.1 and 3.2 show that both Sequential Single Array and Sequential Pair Array generators appear to have good statistical behaviour. It should be noted that a Sequential Pair Array generator using m array pairs takes more computer time to generate a random number than a Sequential Single Array generator using 2m arrays, without improving the statistical behaviour noticeably. Therefore, the data in Tables 3.3 to 3.5 compares already existing generators with Sequential Single Array generators only.

For all series lengths of generated digits, the Sequential Single Array systems appear to compare very favorably with already existing systems with respect to the serial correlation between consecutive pairs of overlapping digits:

improvements in the Sequential Single Array methods (or in the starting value of the matrix A used) could cause a further improvement in this respect.

From observing Table 3.3, there does not appear to be a noticeable difference between Sequential Single Array systems, and already existing systems with respect to the observed sample covariances of generated sequences of numbers. The difference between the maximum positive and negative values for all tested lags is smaller in many cases for the Sequential Single Array generated sequences than for sequences generated by already existing methods (note that zero lies well within the observed range of values in all cases). Again, improvements in the methods (or in the starting value of A) could cause the differences to become more noticeable.

The same algorithms for the Sequential Single Array generators were run on an IBM 360/67 (32-bit word length) and a DEC-PDP9 (16-bit word length). As these two computers represent an extreme in allowable bit-word length, it would appear that Sequential Single Array generators should be able to be implemented without any changes in the algorithm involved, or parameter values used, on any computer that allows array manipulation. The same is not necessarily true for a congruential or Tausworthe generator.

The following table shows a comparison between various generators in terms of time and core used, using actual data taken from testing with L = 2, and all testing subroutines eliminated.

Table 3.5 Time and Core Comparison

Generator	Time (sec)	Core Used (bytes)
MC	0.63	900
CC	1.04	1812
T	56.36	2556
$M_3 1$	9.85	1748
$M_4 1$	9.85	1788
$M_3 2$	9.27	1620
$M_4 2$	9.27	1660
$M_3 3$	10.36	1880
$M_4 3$	10.36	1920
$M_3 4$	12.61	2444
$M_4 4$	12.61	2484

Time refers to the time in seconds taken to generate 100 numbers of 2 digits each. Core Used refers to the number of bytes used to store the FORTRAN object code plus all variables needed for generating a random number of L digits.

It should be noted that the extremely high table values for the Tausworthe generator are caused by the fact that the FORTRAN implementation of such a generator involves simulation of the Exclusive-OR operation. The use of a language such as ASSEMBLER (for which Exclusive-OR is part of the basic instruction set) would drastically reduce both time and core used. As for the other generators, both the Multiplicative Congruential and Composite Congruential generators are much faster than the new generators. However, in terms of core used, the new systems as they now stand (particularly Methods 1 and 2) compare favorably with the Composite Congruential generator. It may be that more efficient program coding will reduce the core used for the new generators to such an extent that they will all save core storage when compared to the Composite Congruential generator, and may even approach the 900 bytes used for a Multiplicative Congruential generator.

When looking at CORE USED for all of the generators, two factors should be considered:

a) For M1, M2, and M3, as L (the number of digits used to form each random number) increases by 1, CORE USED will increase by 4 bytes (these 4 bytes being used to store the extra digit generated). For M4, as L increases by 1, CORE USED will increase by $(4+4L^2)$ bytes (the same 4 bytes as before, plus the extra core used to store the extra elements of the selector matrix B).

b) For all the new generators, as M increases by 1, CORE USED will be increased by the 40 bytes used to store the new column of the A matrix.

The following is an example of how CORE USED was calculated for method 3 with L=2, M=3.

CORE USED

BYTES	EXPLANATION
1728	FORTRAN object code
120	A-matrix of (10x3) numbers
4	value of L
4	value of M
4	column pointer
4	special pointer
4	column index K
4	the final L-digit number
8	the two generated digits, stored separately

TOTAL 1880 bytes

With respect to the already reasonable behaviour of the new systems, further research should be done in order to determine 1) if the starting parameter values for a Sequential Single Array system can be optimized for a particular number of columns in matrix A, and 2) if a further increase in the number of columns in A improves the statistical behaviour of the generator enough to offset any increase in computer storage required. Any work done along these lines should take the following fact into account: an increase in the number of columns of A causes an increase in both the amount of computer storage required, and the expected length of a unique series of generated digits. The latter part of this statement can be seen from the fact that for a starting value for matrix A with n columns, the probability that A will again take on its starting value is:

$$\frac{n}{10 \times (10!)^n} .$$

This suggests that the expected value of the unique digit series length is

$$\frac{10 \times (10!)^n}{n} .$$

If one also takes into account the probability that the various pointers will also return to their starting values at the same time as Matrix A , this expected unique digit length will be increased further.

There is another distinct advantage to the new system of generators. Unlike any Tausworthe or congruential generator, the new generators can be easily adapted to the generation of random digits using any base arithmetic up to base 10. The general programs for generating random digits using Sequential Single Array methods for any such base arithmetic are given in Appendix B. Only base 10 generators were tested here in order to provide direct comparisons with the already existing generators.

It might be wise in the future, when analyzing the new generators, to incorporate tests of significance in analyzing the results of specific tests for uniformity. In this way, one can truly determine if a generator passes or fails certain criteria.

To summarize, in spite of an increase in computer time required for implementing a Sequential Single Array digitized generator, these new generators as tested appear to compare favorably with existing generators with regard to statistical behaviour, and, in most cases, storage. The new generators are also machine-independent: this is not necessarily true for any congruential or Tausworthe generator. It is concluded, therefore, that Sequential Single Array generators do have some merit, and that any further work to be done should concentrate on optimizing these generators.

BIBLIOGRAPHY

Coveyou, R.R., 1960. "Serial Correlation in the Generation of Pseudo-Random Numbers", J.ACM., 7: 72-74.

Downham, D.Y., 1969. "The Runs Up and Down Test", Computer Journal, 12: 373-76.

Fisher, R.A., and Yates, F., 1968. "Statistical Tables for Biological, Agricultural and Medical Research", Oliver and Boyd, London.

Gorenstein, S., 1967. "Testing a Random Number Generator", C.ACM., 10: 111-18.

Hull, T.E., and Dobell, A.R., 1962. "Random Number Generators", SIAM Review, 4: 230-54.

Hutchinson, D.W., 1966. "A New Pseudorandom Number Generator", C.ACM., 9: 432-33.

Jansson, B., 1966. "Random Number Generators", Victor Petterson's, Stockholm.

Keeping, E.S., 1962. "Introduction to Statistical Inference", D. Van Nostrand Company, Inc., Princeton, N.J.

MacLaren, M.D., and Marsaglia, G., 1965. "Uniform Random Number Generators", J.ACM., 12: 83-89.

Marsaglia, G., 1968. "Random Numbers Fall Mainly in the Planes", Proc. Nat. Acad. Sci., 61: 25-28.

________ and Bray, T.A., 1968. "One-Line Random Number Generators and their Use in Combinations", C.ACM., 11: 757-59.

May, K.F., 1967. "Pseudo Random Number Generator", Computing Services, University of Alberta, Edmonton.

Peach, P., 1961. "Bias in Pseudo-Random Numbers", J.A.S.A., 56: 610-18.

Rotenberg, A., 1960. "A new Pseudo-Random Number Generator", J.ACM., 7: 75-77.

Scott, D.A., 1967. "The Generation of Pseudo-random Numbers", M.Sc. Thesis, University of Alberta, Edmonton.

Student, 1908. "The Probable Error of a Mean", Biometrika, 6: 1-25.

Tausworthe, R.C., 1965. "Random Numbers Generated by Linear Recurrence Modulo Two", Math. Comp., 19: 201-09.

Thompson, W.E., 1958, "A Modified Congruence Method of Generating Pseudo-Random Numbers", Comp. J., 1: 83-86.

Tootill, J.P.R., Robinson, W.D., and Adams, A.G., 1971. "The Runs Up and Down Performance of Tausworthe Pseudo-random Number Generators", J.ACM., 18: 381-89.

Whittlesey, J.R.B., 1968. "A Comparison of the Correlational Behaviour of Random Number Generators for the IBM 360", C.ACM., 11: 641-44.

________, 1969. "On the Multidimensional Uniformity of Pseudo-random Number Generators", C.ACM., 12: 247.

Zierler, N., and Brillhart, J., 1968. "On Primitive Trinomials (Mod 2)", Information and Control, Vol. 13, No. 6: 541-54.

Appendix A

Description of Tests for Uniform Pseudorandom Number Generation

A series of tests were used in order to determine the randomness of a generated sequence of uniform pseudorandom numbers. Suppose, then, that we have a sequence of I numbers of length L digits each $(x_1, x_2, \ldots, x_I)$, generated by some uniform random number generator. The following should hold true if the generated numbers are from the true Uniform distribution:

a) Moments (see Gorenstein (1967)).

i) The first moment, $\left(\sum_{i=1}^{I} x_i\right) \Big/ (I * 10^{L})$, has expected value $\frac{1}{2}$.

ii) The second moment, $\left(\sum_{i=1}^{I} x_i^2\right) \Big/ (I * 10^{2L})$, has expected value $\frac{1}{3}$.

iii) The third moment, $\left(\sum_{i=1}^{I} x_i^3\right) \Big/ (I * 10^{3L})$, has expected value $\frac{1}{4}$.

iv) The sample covariance,

$$\left[\left(\sum_{i=1}^{I-\gamma} x_i x_{i+\gamma}\right) \Big/ \left((I-\gamma) * 10^{2L}\right)\right] - \bar{x}_i \bar{x}_{i+\gamma} .$$

should be equal to zero for any value of $\gamma \geq 1$

$$\left[\text{where } \bar{x}_i = \left(\sum_{i=1}^{I-\gamma} x_i\right) \Big/ \left((I-\gamma) * 10^L\right) ,\right.$$

$$\left. \bar{x}_{i+\gamma} = \left(\sum_{i=1}^{I-\gamma} x_{i+\gamma}\right) \Big/ \left((I-\gamma) * 10^L\right)\right] .$$

b) <u>Runs Up and Down</u> (see Downham (1969)).

Define r as the number of runs in the sequence of I numbers, and $r^{(m)}$ as the number of runs above and below the mean in the sequence of I numbers. Then:

i) $E(r) = \frac{1}{3}(2I - 1)$.

ii) $E\left(r^{(m)}\right) = \frac{I}{2} + 1$.

c) <u>Frequency Tests</u> (see Jansson (1966)).

i) <u>One-dimensional</u>. One divides the interval $0 - 10^L$ into 10^L equal non-overlapping subintervals, and calculates the frequency distribution of the $I = 10^L$ numbers with respect to these intervals. As the expected value of the number of L-digit numbers falling into each interval is 1 , a χ^2 statistic can then be calculated for testing purposes by using

$$\chi^2 = \sum_{i=1}^{10^L} \frac{(O_i - E_i)^2}{E_i}$$

where O_i is the observed number of L-digit numbers falling in the i^{th} interval, while $E_i = 1$ for all i . For the purpose of testing the generators described here, 100 values of χ^2 were calculated for 100 sequences of $I = 10^L$ numbers, and the average of these values taken as a representative χ^2 statistic with 10^L degrees of freedom.

ii) Two-dimensional. One divides the existence square $(0 - 10^L, 0 - 10^L)$ into equal non-overlapping squares, and calculates the frequency distribution of successive pairs of the I numbers in these squares. These pairs may be overlapping $[(x_1,x_2), (x_2,x_3), \ldots, (x_{I-1},x_I)]$, or non-overlapping $[(x_1,x_2), (x_3,x_4), \ldots, (x_{I-1},x_I)]$. A graph of these points should show a uniform distribution over the existence square.

d) Serial Association (see Keeping (1962)).

The following is an algorithm for calculating a serial association coefficient for a sequence of I numbers of length L digits each:

i) Treat the series of I numbers as a series of (L * I) digits.

ii) Calculate the frequency distribution table of the number of times digit value i is followed by digit value j in the series.

iii) Divide all elements in this table by the total number of overlapping pairs of consecutive digits to obtain a probability table. Therefore, define Π_{ij} as the probability that digit value i occurs, and is followed by digit value j, and Π_i as the probability that digit value i occurs, regardless of digit value j.

iv) Calculate the quantity

$$\phi^2 = \left[\left(\sum_{i=0}^{9} \sum_{j=0}^{9} \frac{\Pi_{ij}^2}{\Pi_i \Pi_j}\right) - 1\right] \Big/ 9.$$

This quantity is a measure of the association of consecutive digits in the series, and should be close to zero for the series of digits to be independent.

Appendix B

Programs

The purpose of this appendix is to present listings of program segments written in FORTRAN IV to generate an L-digit random number using the various models of Sequential Single Array generators described in Chapter 2, as well as listings of programs for testing sequences of numbers generated by such generators. In all cases, the following variables have been defined as the result of generating some previous number (or initializing the system involved):

P	-	the arithmetic base value being used.
L	-	the digit length of the number being generated.
IA	-	the matrix A .
M	-	the number of columns in A .
IP	-	reference pointer to an element in a column of A .
K	-	reference column index pointer for A .
IP2	-	special column index pointer for A .
IB	-	the 2×L selector matrix B .

In addition, the following two variables are assigned values in the course of generating a number:

NUM - a vector containing the L digits of a generated number.

KNUM - the final generated number, formed from the elements of NUM.

For testing programs, a sequence of LM consecutively generated values of KNUM is taken (call this sequence NUMB), and the following variables defined:

IFRQ - a 10 × 10 matrix, in which the i,j th element will contain the number of times digit value i is followed by digit value j in the sequence of LM L-digit numbers.

ILN - a vector of length L containing the L digits of a particular element of NUMB to be used in defining IFRQ.

AVG, AVG2, AVG3 - the three moments.

IRUNS - the number of runs up and down.

IRNS - the number of runs above and below the mean.

PHI - the serial correlation coefficient.

COV - the sample covariance.

VFREQ - a vector containing the frequency count used for calculating χ^2.

CHISQ - the χ^2 value.

The program segments are as follows:

a) Method 1.

```
            KNUM=0
            DO 2 N=1,L
            J=IA(K,IP)
            IL=IA(MOD(K,M)+1,J+1)
            IQ=IA(K,IL+1)
            IA(K,IP)=IQ
            IA(K,IL+1)=J
            NUM(N)=IQ
            IP=IQ+1
            IF(IQ.NE.O) GO TO 3
            IQ=1
  3         DO 5 KL=1,P
  5         IA(K,KL)=MOD(IA(K,KL)+IQ,P)
  2         K=MOD(K,M)+1
            DO 11 JJ=1,L
 11         KNUM=KNUM+NUM(JJ)*10**(L-JJ)
```

b) Method 2.

```
            KNUM=0
            DO 2 N=1,L
            J=IA(K,IP)
            NUM(N)=J
            IA(K,IP)=IA(K,P+1-IP)
            IA(K,P+1-IP)=J
            IP=J+1
            IF(J.NE.0) GO TO 3
            J=1
  3         DO 5 KL=1,P
  5         IA(K,KL)=MOD(IA(K,KL)+J,P)
  2         K=MOD(K,M)+1
            DO 11 JJ=1,L
 11         KNUM=KNUM+NUM(JJ)*10**(L-JJ)
```

c) Method 3.

```
            KNUM=0
            DO 2 N=1,L
            J=IA(K,IP)
            IL=IA(IP2,J+1)
            IQ=IA(K,IL+1)
            IF(MOD(IQ,M).NE.M-1) GO TO 3
            IP2=MOD(IP2,M)+1
            GO TO 5
   3        IP2=MOD(IP2+IQ,M)+1
   5        IA(K,IP)=IQ
            IA(K,IL+1)=J
            IP=IQ+1
            NUM(N)=IQ
            IF(IQ.NE.0)GO TO 983
            IQ=1
 983        DO 6 KL=1,P
   6        IA(K,KL)=MOD(IA(K,KL)+IQ,P)
   2        K=MOD(K,M)+1
            DO 11 JJ=1,L
  11        KNUM=KNUM+NUM(JJ)*10**(L-JJ)
```

d) Method 1, with selector matrix.

```
            KNUM=0
            DO 2 N=1,L
            J=IA(K,IP)
            IL=IA(MOD(K,M)+1,J+1)
            IQ=IA(K,IL+1)
            IA(K,IP)=IQ
            IA(K,IL+1)=J
            NUM(N)=IQ
            IP=IQ+1
            IF(IQ.NE.0) GO TO 3
            IQ=(MOD(J,P) +1)
   3        DO 5 KL=1,P
   5        IA(K,KL)=MOD(IA(K,KL)+IQ,P)
   2        CONTINUE
            DO 11 JJ=1,L
  11        KNUM=KNUM+NUM(IB(1,JJ))*10**(L-JJ)
            IZ=IB(2,MOD(IP,L)+1)
            IR=MOD(IP,L)+1
            IT=IB(1,IZ)
            IB(1,IZ)=IB(1,IR)
            IB(1,IR)=IT
            DO 96 I=1,L
            IB(1,I)=MOD(IB(1,I)+NUM(1),L)+1
            IT=IB(1,I)
            IB(1,I)=IB(2,I)
  96        IB(2,I)=IT
```

e) Tests giving data for Tables 3.1 - 3.2.

```
            IF(NUMB(2)-NUMB(1)) 15,15,16
   15       ISG=-1
            GO TO 17
   16       ISG=1
   17       IRUNS=1
            DO 14 I=3,LM
            IF((NUMB(I)-NUMB(I-1))*ISG) 18,14,14
   18       IRUNS=IRUNS+1
            ISG=-ISG
   14       CONTINUE
            IF(NUMB(1).LT.LM/2.0) GO TO 32
            ISG=-1
            GO TO 33
   32       ISG=+1
   33       IRNS=1
            DO 34 I=2,LM
            IF((NUMB(I)-LM/2.0-0.5)*ISG) 35, 34, 34
   35       IRNS=IRNS+1
            ISG=-ISG
   34       CONTINUE
            AVG=0.0
            AVG2=0.0
            AVG3=0.0
            DO 24 I=1,LM
            AVG=AVG+FLOAT(NUMB(I))/LM
            AVG2=AVG2+(FLOAT(NUMB(I))/LM)**2
   24       AVG3=AVG3+(FLOAT(NUMB(I))/LM)**3
            AVG=AVG/LM
            AVG2=AVG2/LM
            AVG3=AVG3/LM
            DO 1 I=1,LM
    1       VFREQ(I)=0.0
            DO 2 I=1,LM
            KT=NUMB(I)+1
    2       VFREQ(KT)=VFREQ(KT)+1
            CHISQ=0.0
            DO 3 I=1,LM
    3       CHISQ=CHISQ+((VFREQ(I)-1)**2)
```

f) Test for Serial Correlation Coefficient.

```
      DO 868 I=1,10
      DO 868 J=1,10
868   IFRQ(I,J)=0
      DO 50 I=1,LM
      DO 51 K=1,L
      ILN(K)=(NUMB(I)/(10**K))*10**K
51    ILN(K)=NUMB(I)-ILN(K)
      DO 52 K=2,L
52    ILN(L+2-K)=(ILN(L+2-K)-ILN(L+1-K))/10**(L+1-K)
      DO 53 K=2,L
53    IFRQ(ILN(K-1)+1,ILN(K)+1)=IFRQ(ILN(K-1)+1,ILN(K)+1)+1
      IF(I.EQ.LM) GO TO 50
      KR=NUMB(I+1)/10**(L-1)
      IFRQ(ILN(K-1)+1,KR+1)=IFRQ(ILN(K-1)+1,KR+1)+1
50    CONTINUE
      FSUM=0.0
      DO 212 I=1,10
      DO 212 J=1,10
212   FSUM=FSUM+IFRQ(I,J)
      DO 213 I=1,10
      DO 213 J=1,10
213   FMAT(I,J)=IFRQ(I,J)/FSUM
      DO 4 I=1,10
      FPRB(I)=0.0
      DO 4 J=1,10
4     FPRB(I)=FPRB(I)+FMAT(I,J)
      DO 5 J=1,10
      GPRB(J)=0.0
      DO 5 I=1,10
5     GPRB(J)=GPRB(J)+FMAT(I,J)
      PHI=0.0
      DO 6 I=1,10
      DO 6 J=1,10
6     PHI = PHI+((FMAT(I,J)**2.0)/(GPRB(J)*FPRB(I)))
      PHI=(PHI-1.0)/9.0
```

g) Test for Sample Covariance.

```
      DO 550 IG=1,20
      IF((IG.GT.10.AND.IG.LT.15).OR.(IG.GT.15.
     1OR.IG.LT.20) GO TO 550
      LP=LM-IG
      SUM1=0.0
      SUM2=0.0
      SUM3=0.0
      DO 551 KG=1,LP
      SUM1=SUM1+NUMB(KG)/FLOAT(LM)
      SUM2=SUM2+NUMB(KG+IG)/FLOAT(LM)
  551 SUM3=SUM3+NUMB(KG)*NUMB(KG+IG)/FLOAT(LM**2)
      COV=(SUM3/LP)-((SUM1*SUM2)/(LP**2))
  550 CONTINUE
```

Appendix C

Periodicity

One aspect of the new random number generators that was not discussed previously in any detail was the actual periodicity of the sequences of digits generated. Due to a lack in availability of computer time and resources, it was not considered feasible to test the base 10 Sequential Single Array generators of Chapter 3 extensively for periodicity[1]. However, as the question of the possible effect of an increase in the number of columns of matrix A on the periodicity was raised, it was decided to perform some testing for periodicity on a base 4 version of one of the new Sequential Single Array generators. Method 2 was chosen, and the following values of A were used:

[1] A total of 60,000,000 digits were generated using the base 10 generator referred to in Chapter 3 by $M_4 1$, without any repetition in the set of starting parameter values, and without any internal degenerate sequences occuring. However, due to the high cost involved in generating that many digits, the process was not carried out any further.

$$A_1 = \begin{bmatrix} 2 & 0 & 3 & 1 \\ 1 & 2 & 0 & 3 \\ 3 & 3 & 2 & 2 \\ 0 & 1 & 1 & 0 \end{bmatrix} \qquad A_3 = \begin{bmatrix} 2 & 3 & 0 & 1 \\ 1 & 0 & 2 & 3 \\ 3 & 2 & 3 & 2 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

$$A_2 = \begin{bmatrix} 2 & 0 & 3 & 1 & 1 \\ 1 & 2 & 0 & 3 & 3 \\ 3 & 3 & 2 & 2 & 2 \\ 0 & 1 & 1 & 0 & 0 \end{bmatrix} \qquad A_4 = \begin{bmatrix} 2 & 3 & 0 & 1 & 1 \\ 1 & 0 & 2 & 3 & 3 \\ 3 & 2 & 3 & 2 & 2 \\ 0 & 1 & 1 & 0 & 0 \end{bmatrix}$$

For each of the starting values of A used, all possible combinations of starting values for the column pointer z and pointer index p were tested, in order to determine which combination of p and z for a particular A produced a sequence of digits of maximum periodicity. The results were as follows:

4-column matrices		5-column matrices	
Matrix	Periodicity Length	Matrix	Periodicity Length
A_1	24128	A_2	298590
A_3	27200	A_4	254730

It would appear as though the increase in the number of columns of A from 4 to 5 causes a great increase in the actual periodicity - it is expected that a further increase in the number of columns (and the optimization of A for any number of columns) will cause the increase in periodicity to be more pronounced.

Appendix D

Applications to Queueing Theory

C.1 Introduction

One major use of Uniform pseudorandom number generation is in the study and simulation of queueing systems, in which the generation of random arrivals and departures plays an important role. It is the purpose of this section to define one particular queueing model, and to describe how one can optimize this model by use of Uniform pseudorandom number generation.

C.2 Description of Single-Channel, Single-Server, Dynamic Queueing Model

A single-channel single-server dynamic queueing system (no feedback) can be denoted by $S(T,\vec{\alpha},\vec{\beta},N)$, in which:

1) T , the processing period (or simulation period) is the lapse of time between the start $(t = 0)$ and the end $(t = T)$ of each processing.

2) $\vec{\alpha} = \alpha(t)$ $t = 0,1,\ldots,T$ is a vector of time-dependent job arrival rates.

3) $\vec{\beta} = \beta(t)$ $t = 0,1,\ldots,T$ is a vector of time-dependent job departure rates, and

4) N is the maximum allowable queue length.

It should be noted that:

1) $\beta(t)$ is independent of $\alpha(t)$, but dependent on the efficiency of the server.

2) The maximum allowable queue length, N , is dependent entirely on the setup of the server. An incoming job will be rejected if there are already N jobs waiting in the queue. N is assumed here to be time independent.

When simulating such a system, the following quantities should be calculated:

1) The total number of customers arriving at the system ARR .

2) The total number of customers rejected by the system REJ .

3) The total number of customers entering the server ENT .

4) The total number of customers departing from the server DEP .

5) The number of time units "t" for which the server is idle IDL .

6) The aggregate total of the number of units of time all customers have waited in the queue QTM .

The following is an algorithm for simulating this system:

1. Initialize APR, REJ, ENT, DEP, IDL, and QTM to be zero.

2. Set the index I , and the queue-length to be zero.

3. Generate two normalized L-digit random numbers ARVL and DEPT using a sequential single array method.

4. If DEPT is less than $\beta(I)$, then go to statement 10 . Else,

5. If ARVL is less than $\alpha(I)$, then go to statement 18 . Else,

6. If the server is busy, then go to statement 9 . Else,

7. If the queue-length is not equal to zero, then go to statement 20 . Else,

8. Increase the value of IDL by one as the server is idle, and go to statement 25 .

9. Increase QTM by the value of the queue-length, and go to statement 25 .

10. If the server is not busy, then go to statement 16 . Else,

11. Increase the value of DEP by 1 , as a job is leaving the server.

12. If ARVL is less than $\alpha(I)$, then go to statement 15 . Else,

13. If the queue-length is equal to zero, then go to statement 8 . Else,

14. Go to statement 20 .

15. Increase the values of ENT and ARR by 1 (as a job is arriving at the system, and a job is entering the server) and go to statement 9 .

16. If ARVL is less than $\alpha(I)$, then go to statement 15 . Else,

17. Go to statement 8 .

18. If the queue-length is not equal to N , then go to statement 21 . Else,

19. Increase the value of REJ by 1 (as a job is rejected by the system) and go to statement 9 .

20. Decrease the queue-length by 1 , and increase the value of ENT by 1 (as a job is entering the

server from the queue) and go to statement 9 .

21. Increase the queue-length by 1 .

22. If the server is not busy, then go to statement 24 . Else,

23. Increase the value of ARR by 1 , and go to statement 9 .

24. Decrease the queue-length by 1 , increase the values of ENT and ARR by 1 , and go to statement 9 .

25. Increase the value of I by 1 .

26. If the value of I is less than T , then go to statement 3 . Else,

27. Stop, as the simulation is finished.

C.3 Optimization of the System for Given $\vec{\alpha}$ and N .

In order to determine the optimal behaviour of the system described in section C.2, it is necessary to find the answers to the following two questions in turn.

a) For any system where the area under the curve defined by $\alpha(t)$ vs t is equal to the area under the corresponding curve defined by $\beta(t)$ vs t, what pattern of $\beta(t)$ will optimize the system for a given $\alpha(t)$?

b) Starting with the optimal system of $\alpha(t)$ and $\beta(t)$ values determined from answering question a), how will overstaffing and understaffing* change the behaviour of the system?

Before attempting to answer either of these questions, one must first assign arbitrary values to:

i) LOSS - the loss in profit for each time unit a customer waits in the queue.

ii) GAIN - the gain in profit for each time unit a customer spends in the server.

* By overstaffing and understaffing, we refer to defining values of $\beta(t)$ such that the curve defined by $\beta(t)$ vs t is either greater than or less than (in area underneath) the curve defined by $\alpha(t)$ vs t .

iii) IDCOST - the upkeep cost for each time unit the server is idle.

iv) SVCOST - the labor cost for each time unit the server is busy.

Once these variables have been given values, any curve $\beta(t)$ vs t that satisfies the criteria of either questions a) or b) can be used to determine the behaviour of the complete system for given $\vec{\alpha}$, T, and N by the following:

i) Define the set of values $\beta(t)$ $t = 0,1,\ldots,T$.

ii) Perform a series of simulations using $S(T,\vec{\alpha},\vec{\beta},N)$, and calculate the average values of the parameters IDL , QTM .

iii) Calculate the system profit by evaluating:

PROFIT=((GAIN-SVCOST)*(T-IDL))-(IDCOST*IDL)-(LOSS*QTM).

By testing various $\beta(t)$ patterns in this way, one can eventually find a pattern that produces a maximum value of PROFIT for a given $\vec{\alpha}$, T , and N in order to answer the two questions posed above.