A LESS DETERMINISTIC METHOD OF RANDOM NUMBER GENERATION

By

MARK STEPHEN MONROE

Bachelor of Arts

St. Olaf College

Northfield, Minnesota

1981

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December, 1983

A LESS DETERMINISTIC METHOD OF RANDOM NUMBER GENERATION

Thesis Approved:

Joe H. Mize
Thesis Adviser

[illegible]

J P Chandler

Philip M. Wolfe

Norman N. Durham
Dean of the Graduate College

PREFACE

As the number of uses for random numbers increases, the need for better methods of producing them increases. However, the most widely used method today was introduced over thirty years ago. These linear congruential generators are fast and compact, but the sequences they produce have been under fire for years.

Attempts have been made to improve congruential sequences by shuffling them, but while this solves some of the problems, it does not solve all of the problems inherent in such a highly deterministic method. The need was thus seen for a less deterministic approach. The following study introduces such an approach, and compares it to accepted generators.

I wish to express my gratitude to Dr. Joseph H. Mize for his ideas, encouragement, and guidance, as my major adviser. I also extend thanks to Dr. J. P. Chandler, Dr. C. Patrick Koelling and Dr. Philip M. Wolfe for their guidance as members of my committee. Thanks also to Claus Christiansen for his help.

Special thanks to my parents, without whom I would not be here. And to my wife for her support and assistance, and especially for putting up with my audible thought processes.

TABLE OF CONTENTS

LIST OF TABLES

CHAPTER I

INTRODUCTION

A Brief History of Random Number Generation

Man has used methods of generating random numbers for centuries. This has been mainly for recreational purposes, such as playing cards and rolling dice. Most people don't think of rolling dice for craps as generating random numbers, but it is actually a very good method, assuming that the dice are fair. We also make decisions using a simple 0-1 generator known as a coin.

Many more serious uses have arisen for random numbers. These include computer simulation techniques, random sampling for statistical analysis and a number of problems that are difficult or impossible to solve by other means. One might expect those coins and dice to be in constant motion, except that most of these applications require far too many numbers to be able to use such slow methods of generation. It is out of this need that faster methods of generation have been developed.

The first success came with machines that could generate "random noise" which was then interpreted as digits. Kendall and Babington-Smith produced a table of 100,000 random digits in 1939 by such a method (Knuth, 1969). The Rand Corporation published a table of one million random digits in 1955 using a similar method.

While these methods may produce good random sequences, it is difficult to attach such a machine to a computer. Also, the sequences produced are not repeatable, which is often desirable. A solution to these problems is to store a large list of numbers in a computer or on tape. However one runs into problems of space required and the possibility of running out of numbers if the sequence is not long enough.

Because of these shortfalls, a method was needed that could produce unlimited numbers and could repeat the same sequence, while not requiring an excessive amount of space on computers. Von Neumann (1951) developed the first such generator in 1946 using a computer algorithm known as the mid-square method. Since then many other algorithms have been developed. These algorithms have become the main source of random numbers, when large sequences are needed. It is on these types of generators that the remainder of this paper shall focus. Before discussing the various algorithms available, it is necessary to discuss what is meant by such terms as "random numbers" and "random sequences".

What is Randomness?

One can place pieces of paper with the digits 0 through 9 on them in a hat and then draw one out. Is the number chosen a random number? It would be better expressed as a number chosen at random. Thus, numbers are not random, but can be obtained in a random manner.

What one normally speaks of as a random number generator might be better called a random sequence generator. That is, a method of choosing numbers in a random manner so as to create a sequence with random characteristics.

What then are random characteristics of a sequence? Although random sequences can fit various distributions, if none is mentioned it is assumed that the sequence fits a uniform distribution. That is, the probability of finding any specific digit, 0-9, in a given position of the sequence should be about 1/10. Likewise, the probability of finding any pair of digits in a given pair of locations should be $(1/10)^2$, and so on.

Most pseudo-random number generators produce sequences of, hopefully, uniformly distributed real numbers on the interval (0, 1). Various functions can then be used to transform this standard uniform output to fit a desired distribution.

Why is it called a pseudo-random number generator? The reason is that the methods available for generating random sequences are deterministic. That is, each number is a function of some previous value and is thus dependent, and cannot truly be called random. The methods available simply produce sequences of numbers which can pass a certain number of statistical tests which check for specific characteristics of random sequences. If enough tests are passed then the method is assumed to be acceptable.

CHAPTER II

PSEUDO-RANDOM SEQUENCE ALGORITHMS

As mentioned earlier, Von Neumann first proposed generating random-sequences by computer algorithm in 1946. He proposed the "middle-square", or midsquare, method. Using this method, each number is produced by squaring the current number and using the middle digits of the solution for the next value. For example, 7316 squared equals 53523856; the next number is 5238. Unfortunately, this method was plagued with problems. It has a very short time before it starts repeating itself (its period). It may even turn into a very short loop, such as 5600, 3600, 9600, 1600, 5600. Or it may simply degenerate, for example 3741, 9950, 0025, 0006, 0000, ... Finally, it does not stand up well to empirical testing (Forsyth, 1951; Knuth 1969; Shannon, 1975). Because of these problems, the midsquare method is no longer used.

The linear congruential method of generating random numbers was developed in 1949 by Lehmer (1951; Coveyou, 1960; Hull and Dobell, 1962, 1964; Knuth, 1969; Smith, 1971). This method, and its numerous variations, are still the most widely studied and used methods. The congruential generators calculate each number from the preceding one using the formula $x_n = (a * x_{n-1} + c) \bmod m$. m is generally a large power of two, such as 2^{35}. (See Hull and Dobell (1962), or Knuth (1969), for a discussion on choosing values for a, c, and m.) When $c = 0$ the method is called multiplicative congruential; when $c \neq 0$ it it called mixed congruential.

There are many arguments throughout the literature on how to maximize the period of a congruential generator, minimize serial correlation, and minimize many other problems (Hull and Dobell, 1962; Coveyou, 1960; Greenberger, 1961). As with the midsquare method, each number generated with a congruential generator is completely dependent on the previous value. The result is that there will always be serial correlation, and other characteristics, that should not exist in a truly random sequence.

This problem becomes very obvious when mapping values from a sequence onto a line, square, cube, or higher dimensional space. Individual values mapped onto a line cover it uniformly, as do pairs of values mapped onto a square. However, when mapping triples onto a cube, the space is not covered uniformly. The points fall into planes cutting through the cube (Marsaglia, 1968).

Attempts have been made to improve linear congruential sequences through shuffling. MacLaren and Marsaglia introduced this concept in 1965. One congruential generator is used to fill a table with pseudorandom numbers. Another generator is used to "randomly" select values from the table. The size of the table is not critical. Its purpose is to store values so they may be withdrawn in a different order than they were produced. Once a value has been withdrawn the first generator replaces it with another (MacLaren and Marsaglia, 1965).

This method, while taking more time, does reduce many of the problems that plague congruential generators. Shuffling breaks the dependency between consecutive values generated. But this does not solve all of the problems. One example is that, in congruential sequences, each number will occur only once per period. A truly random sequence does not have this restriction, and shuffling the sequence does not solve the problem.

Even though shuffling of congruential sequences does not solve all of the problems, it does reduce them, and is worth the extra time required (Nance and Overstreet, 1978). Variations on this method have been developed by Westlake (1967), and Bays and Durham (1976).

Another type of generator was introduced by Tausworthe in 1965. The Tausworthe, or shift register, generator is related to the Lehmer congruential method. While linear congruentials are based on the residue of an integer product, modulo m; the Tausworthe methods are based on the residue of polynomials, computed modulo a primitive polynomial over the Galois field of base two (Tausworthe, 1965; Lewis, 1972, 1975; Tootill, 1973; Canovos, 1967; Albert, 1937). As might be expected, this is a slower method, however it is theoretically sound, and has been shown to be more consistent in tests on n-tuples, where congruential methods fail.

In brief, the Tausworthe theorem is as follows. Let $a=(a_k)$ be the sequence of 0's and 1's generated by $a_k = \sum_{i=1}^{n} c_i * a_{k-1}$ (mod 2) for any $c_i \in (0,1)$ and $c_n=1$, where $f(x) = \sum_{i=1}^{n} c_i * x^i$ is primitive over the Galois field of base two. Then $y = (y_k)$ will be uniformly distributed where $y_k = \sum_{t=1}^{L} 2^{-t} a_{q \cdot j+r-t}$ for $0 < = r < = 2^n - 1$, $L < = n$, $(q*2^n - 1) = 1$ and $q > L$ (Lewis, 1975). This type of generator is very sensitive to the choice of the primitive trinomial and other parameters, which makes it more difficult to use.

As with the Lehmer generator, there are variations of the Tausworthe generator. Lewis and Payne developed the Generalized Feedback Shift Regester (GFSR) method (Lewis, 1973, 1975). Bright and Enison produced the TLP, or Tausworthe, Lewis, Payne generator (Bright, 1979).

Although complicating the calculation of each number greatly, these methods have not escaped the major problem inherent in the previous methods. They are deterministic. Rather than each value being dependent on the previous one, it is now dependent on the last "L" values. These methods have not been tested as extensively as the others, which may be due in part to their slowness and difficulty of use.

CHAPTER III

RESEARCH OBJECTIVE

Based on the need for less deterministic methods of pseudo-random number generation, the purpose of this study is to develop a less deterministic method of generation. The objective is then to show that the proposed method's performance is superior to that of the accepted methods on empirical tests.

One cannot expect any method to perform better on all tests, as the methods compared should pass most of the tests used. However, a special battery of tests will be performed to highlight dependencies between consecutive numbers in the sequences. It is on this set especially that the proposed method should show promise.

In addition to empirical tests and comparisons, the theoretical properties of sequences produced by the proposed method will be evaluated based on a definition of a random sequence. Probabilists and Computer Scientists differ on such definitions. Therefore, Chapter V will discuss some of the definitions in depth, culminating in a fairly strong definition. In Chapter VI, the properties of the proposed method will be presented based on this definition.

CHAPTER IV

PROPOSED METHOD

To avoid the nonrandom characteristics inherent in highly deterministic methods of generating numbers the Proposed Method uses a uniformly distributed table for its source of random numbers. The table contains all of the numbers .000, .001, . . . , .999. The numbers are jumbled so that the probability of finding any specific number in a given location of the table is approximately 1/1000.

The table need be created only once and then stored in the computer. The random number generator is then initialized by reading the table from its file into an array. The user then supplies a seed value, or starting location, in the table. A fast congruential generator is then used to supply numbers "uniformly distributed" between 0 and 999. These values are then added to the current table location mod 1000 to arrive at the next location. The number stored there is the next random number.

Thus, once the generator is initialized each successive number is generated by the following steps:

1) Generate $c_i \in [0,999]$ by congruential method (Where c_i represents the ith element of the congruential sequence.)
2) Add c_i to current table location mod 1000:

$$t_j + c_i \bmod 1000 \Rightarrow t_{j+1}$$

(Where t_j is the current table location and t_{j+1} is the newly derived table location.)

3) Value $r \in [.000, .999]$ located in t_{j+1} is next number (Where r is the obtained pseudo-random number.)

4) Go to 1) and repeat for next number.

For the remainder of this paper, the generator will be referred to as the Pseudo-Random Access, Uniform Table, or PRAUT, method.

Because the PRAUT method uses exactly 1000 values, it does not have the resolution that other methods have. This is not a major problem. For most purposes the least significant digits will be irrelevant. However, there will be times when further resolution would be useful. Under these circumstances two values can be generated and combined as follows: $r = r_j + .001*r_{j+1}$. This produces six digit numbers while requiring more time to generate. However, if the last three digits are important it would not be wise to depend on the randomness of these digits in a congruential sequence (see Chapter VI).

Both the randomness and the relatively long period of this generator can be improved by an additional step. The only way the generator can cycle is if the congruential generator providing addresses begins its cycle when the current table location is the same as it was at the start of an earlier cycle of the congruential generator. But if we were to change the table as we generate numbers, the chances of cycling within the life time of this universe are extremely remote.

A method of doing this is as follows. After using the value located in a position, one switches the value with another. Given that the current table location is t_{j+1}, found by $t_j + c_i$ mod 1000; switch its contents with that in location $t_{j+1} + c_i$ mod 1000. Thus, the next time that the location t_{j+1} is arrived at, the probability that the number that brought it there was the same as c_j is 1/1000. So the numbers switched have a very low probability of getting switched back.

The generator will be discussed with and without this step as the author feels that its performance should be satisfactory without it. It involves a trade off between period length and possible randomness and time taken to generate.

The remaining unexplained part of the PRAUT generator is how the table is created. Because the table is created only once, its creation can be as thorough as one desires. However, the following method should be sufficient.

The proposed operation has two steps. First fill the table in a somewhat random manner and then shuffle it. The filling operation is similar to the method of locating numbers when generating. Starting at some position in the table, generate a number [0, 999] using a congruential generator and add it to the current location mod 1000. If that position is already taken then simply repeat until an empty one is found. When the numbers .000 through .999 are in the table the second step begins.

The second step is identical to the optional step in generating numbers. That is switching the contents of each location $t_j + c_i$ mod 1000 with $t_j + 2c_i$ mod 1000. Once this has been done a few thousand times there should be no resemblance to the table after step one.

Note that the randomness of the table is not critical as the method of obtaining each number from the table uses pseudo-random addresses. However, if the table was not jumbled, but merely the sequence .000, .001, . . . , .999, and if the addresses generated by the congruential method were used in an absolute fashion, rather than relative to the last location, then the generator would perform only as well as the congruential generator used. It is the relative addressing, the randomness of the table, and the optional jumbling of entries during execution that separates the sequence produced by PRAUT from the nonrandom characteristics of the congruential method. This will be discussed in more detail under theoretical properties of the PRAUT method (Chapter VI).

CHAPTER V

SOME DEFINITIONS OF RANDOMNESS

Before discussing the theoretical behavior of the PRAUT generator, we should look more closely at what a random sequence is. There is no single recognized definition of a random sequence, although many have been proposed.

Lehmer (1951) defined a random sequence as

> a vague notion embodying the idea of a sequence in which each term is unpredictable to the uninitiated and whose digits pass a certain number of tests, traditional with statisticians and depending somewhat on the uses to which the sequence is to be put (p. 141).

There are several problems with this definition which represent common mistakes or misconceptions of randomness. The most glaring is the phrase "unpredictable to the uninitiated". This would imply that those of us who have studied randomness could somehow observe a sequence and predict values that are to come. Nothing could be further from the truth. A factor that should be included in any definition of a random sequence is that each value is independent of all other values. Therefore, any method of predicting values based on previous values, or any other method, will succeed in the long run one out of n times, where n is the number of possible values.

The other main problem with the Lehmer definition is the phrase "whose digits pass a certain number of tests, . . .". A random sequence should pass statistical tests, but passing statistical tests in no way assures randomness. This is because, for example, each element of a sequence must be independent of all others, and must be uniformly distributed on [0, 1). But there is no way to empirically test for independence or for uniform distribution because one cannot generate an infinite sequence, which would be required for such tests. One can show that, in the long run, a sequence covers the interval [0, 1) fairly uniformly, but one cannot show that any element U_i has an equal chance of having any of the possible values.

> Given any finite set of tests, there will always be a sequence of numbers that will pass all of the tests but is totally unacceptable for some particular application. It is always possible that it will have patterns that remain undetected despite intensive testing (Shannon, 1975, p. 356).

From this we can see that a definition cannot be based on vague intuitive notions, or empirical tests. It must be based on theoretical characteristics. Or, as Knuth (1969, p. 128) says, "what we really want is a relatively short list of mathematical properties, ..."

A number of authors have delved into more theoretical definitions of random sequences. Knuth's (1969) discussion entitled "What is a Random Sequence?" covers some of the commonly mentioned definitions. These however are still not entirely correct. They will be discussed and compared to a more robust definition.

Knuth (1969) starts with the definition of equidistribution.

Definition 1: "The sequence of U_0, U_1, . . . is equidistributed if and only if Pr $(u <= U_n < v) = v-u$ for all u, v with $0 <= u < v <= 1$" (p. 128). (Note: The author finds the notation $(u <= U_n < v)$ to be inconsistent with previous use, however the following discussion should clarify the meaning.) It follows that any independent, uniformly distributed sequence is equidistributed, by the following theorem.

Theorem 1:

> Let us carry out a sequence of identical independent experiments, in each of which the event A has probability $p=P(A)$ $(0<p<1)$. Let v denote the frequency of the occurence of the event A in the course of the first n experiments. Then one has $v_n/n \to p$ (Renyi, 1970, p. 195).

In other words, if A is the event that $u <= U_n < v$, then the relative frequency of the occurrence of A tends in probability toward (v-u) as n increases.

Equidistribution is a useful definition, as it is empirically testable, to some extent. That is, one can determine, given a sequence of numbers, the number of values that fall in any interval in [0, 1). Unfortunately, this is not a strong enough condition to represent randomness. Many nonrandom sequences are equidistributed, for example (1/2, 1/4, 3/4, 1/8, 3/8, 5/8, 7/8, 1/16, . . .). Note that the components of this sequence are neither independent nor uniformly distributed.

Knuth (1969) also shows that equidistribution is inadequate and proceeds into more robust definitions.

Definition 2: "The sequence U_0, U_1, . . . is said to be k-distributed if $\Pr(u_1 <= U_n < v_1, \ldots, u_k <= U_{n+k-1} < v_k) = (v_1 - u_1) \ldots (v_k - u_k)$ for all choices of real numbers u, v with $0 <= u_j < v_j <= 1$ for $1 <= j <= k$" (p. 129).

Definition 3: "A sequence is said to be ∞-distributed if it is k-distributed for all positive integers k" (p. 129). After many more definitions Knuth (1969) falls back to his definition R1 which states that "a [0, 1) sequence is defined to be 'random' if it is an ∞-distributed sequence" (p. 138).

The concept of an ∞-distributed sequence was introduced by Franklin in 1963 under the name "completely equidistributed". It is a well studied concept, but as Knuth (1969) states, "an infinite sequence which is ∞-distributed satisfies a great many useful properties which are expected of random sequences, . . . " (p. 150). That is, ∞-distributed does not mean random, but rather it means that a sequence has a lot of random qualities. There are problems with it. For example, one can find non-random sequences which are ∞-distributed. There are also problems with definitions that require infinite sequences, when only finite sequences can be generated and used.

The first problem is probably the greater of the two. The fact that sequences exist which are ∞-distributed but definitely not random can be seen through a simple extension of Knuth's proof that equidistribution is not random. Knuth shows that any two equidistributed sequences U_0, U_1, . . . and V_0, V_1, . . . can be used to form $W = (W_0, W_1, . . .)$ where $W = 1/2U_0$, $1/2+1/2V_0$, $1/2U_1$, $1/2+1/2V_1$, ... While this is obviously not an acceptable random sequence, it is equidistributed. This example does not affect ∞-distributed as the sequence is not even 2-distributed. However the following example produces a sequence which is ∞-distributed.

Let U_0, U_1, . . . and V_0, V_1, . . . be ∞-distributed sequences. The values of U_0, U_1, . . . will be transformed to become W_0, W_1, . . . , and the positions and values of V_0, V_1, . . . will direct the process. The sequence V has two functions. First, even positions, V_0, V_2, V_4, . . . represent the function $W = 1/2U_i$, while odd positions, V_1, V_3, V_5, . . . , represent the function $W = 1/2 + 1/2U_i$. Second, the value of each V_i determines how many times the function its position represents will be used. That is, if $V_i < = .5$ then the next two elements of W will be determined by the function V_i represents. If $V_i < .5$ then only the next one element will be determined by the function V_i represents.

Control of the sequence then moves to V_{i+1} to determine the fate of the next one or two elements of W.

To illustrate, let the values of W produced by $1/2U_i$, and therefore lying in [0, 1/2), be depicted as "-", and the values produced by 1/2 + $1/2U_i$, and thus lying in [1/2, 1), be represented as "+". The following sequence V will then produce the sequence W shown.

	V_0	V_1	V_2	V_3	V_4	V_5	V_6	V_7	V_8
V {	.371	.423	.744	.119	.625	.978	.763	.234	.469
W {	-	+	- -	+	- -	+ +	- -	+	-

This sequence will have an equal number of values in [0, 1/2) to those in [1/2, 1), and there is no way to know which values will be which, other than W_0. This sequence would even appear to be random at first glance. Its only problem is that there are never more than two consecutive values greater than .5, or less than .5. But this sequence is ∞-distributed. That is, there is no way to subdivide the sequence into k-tuples so that +'s and -'s do not fall uniformly. Thus, as with equidistribution, a stronger property is needed. That does not necessarily mean an extension of ∞-distributed, as Knuth has already pointed out problems with some of these. Perhaps one should shy away from definitions requiring infinite sequences, as they tend to fall prey to such manipulation. The second problem mentioned will further illustrate this problem.

The second problem is that an infinite sequence can, and will, have finite subsequences that do not look random. For example, one million zeros in a row. This is perfectly acceptable in an infinite random sequence, and will not affect the sequence's chances of being ∞-distributed. Unfortunately, this is not true of finite sequences. Although long, seemingly non-random subsequences can occur in finite random sequences, it is both unlikely and unacceptable. Therefore, since all random sequences used are finite, this is not a very practical definition.

One could modify the definition of ∞-distributed to m-distributed.

Definition 4: A sequence is said to be m-distributed if it is k-distributed for all positive integers $k <= m$. But what happens when $k = m$? We then have a one-to-one correspondence, which means that $\Pr (u_j <= U_i < v_j) = (v_j - u_j)$ for all $i <= m$. Suddenly the meaning is changed. No longer is U_n used, as it represents an infinite number of values of which a certain percent fall between u and v. Now U_i is used, representing one component of the sequence, which has a certain chance of having a value between u and v. This assumes independent and uniform distribution of individual components, aspects which Knuth avoids.

The concept of independence of components of a sequence has been brought up throughout this discussion. In order to discuss independence, a different view of a random sequence is needed. Rather than being any infinite sequence that meets certain criterion, think of a random sequence as a sequence of random variables, or experiments. Each experiment has a number of possible outcomes, or events, each as likely as all others. These events must be independent of one another, which is to say that, given any finite number, n, of events, A_i, $\Pr (A_i * A_{i+1} * \ldots * A_{i+n-1}) = P(A_i) * P(A_{i+1}) * \ldots * P(A_{i+n-1})$. The following theorem shows why this is important to random sequences.

Theorem 2: Given a sequence of events A_i, whose probabilities $P(A)$ are all positive, ". . . a necessary and sufficient condition for mutual independence of the events is the satisfaction of the equations $P_{A_{i_1} A_{i_2} \ldots A_{i_k}} (A_i) = P(A_i)$, for any pairwise different $i_1, i_2, \ldots, i_k, i$" (Kolmogorov, 1956, p. 12). That is, given that events A_{i_1} through A_{i_k} have occurred, the probability that A_i occurs is equal to the probability that A_i would have occurred anyway. One should be able to stop at any point in the generation of a truly random sequence and have no way of predicting the next value based on previous values. This property is unique to independent components and is the major downfall of the Knuth definitions, which define no relationship between individual components.

Independence is not the only property needed. The numbers generated are expected to form a uniform distribution. Thus, each component must be identically uniformly distributed. The result is the same as that desired by the Knuth definitions.

Combining the properties given above produces the following definition.

Definition 5: For the purposes of uniform random sequence generation on [0, 1), a sequence $U_0, U_1, \ldots$ is said to be random if and only if each element, U_i, consists of an identical set of possible outcomes, $A_1, A_2, \ldots, A_n$, which are uniformly distributed on [0, 1), are all equally likely, and for any k elements $P(A_{j_1} * A_{j_2} * \ldots * A_{j_k}) = P(A_{j_1}) * P(A_{j_2}) * \ldots * P(A_{j_k})$. That is to say, for the purpose of uniform random sequence generation, a sequence is said to be random if and only if each component of the sequence is uniformly distributed on [0, 1), and is independent of all other components, or combinations thereof.

This definition is assumed throughout the remainder of this paper, and is especially important to the discussion of the theoretical properties of the PRAUT method.

CHAPTER VI

SOME THEORETICAL PROPERTIES OF THE PRAUT METHOD

There are a number of characteristics to consider when comparing random number generators. These include the speed of the generator, the amount of space it requires in the computer, and the actual numbers it produces. The first two characteristics, speed and space, will be discussed in Chapter VIII, Comparison of Methods, where several generators are compared to the PRAUT method.

In this chapter several characteristics of congruential generators and the PRAUT generator will be compared. The relative length of the periods of the two methods will be discussed first. The second topic discussed will be how the non-random characteristics of the congruential addressing generator affect the PRAUT sequences. Finally, the expected characteristics of the PRAUT method will be discussed in relation to the definition of randomness.

The Period

All of the previously discussed generators cycle, or repeat themselves. Since the numbers they produce are deterministic, derived from a previous number, whenever a number produced is exactly the same as at some time earlier the sequence will begin to cycle. The number of values produced per cycle is called the period.

If the parmaeters for a linear congruential generator are well chosen the period will be very long, such as 2^{35}. This should easily be long enough that, for practical purposes, it will never occur.

The PRAUT generator cycles only if the addressing generator cycles, and then the table location must be exactly the same as it was when the addressing generator cycled some time previously. Suppose the congruential addressing generator cycles after 2^{35} elements, or 34,359,738,368. Then 2^{35} mod 1000 is the difference between the table location at the start and end of the cycle. Since in this case that equals 368, each number generated during this cycle of the addressing generator will be 368 table positions (mod 1000) from the number generated 2^{35} values earlier. The PRAUT generator has not cycled. It will not cycle until the addressing generator cycles 125 times, as (125*368) mod 1000 = 0. At this point almost 4.2 trillion numbers will have been generated.

The period of most congruential generators is long enough to assure that the period of the PRAUT generator is more than long enough for practical purposes. If by some chance there would be an astronomical problem to solve requiring constant, noncycling random numbers, the period of the PRAUT generator with shuffling is longer than 10^{2500}.

Severe Congruential Problems Produce Only Small PRAUT Problems

Random numbers, when plotted in any dimension, should cover it uniformly. Naturally, a random number generator does not have the resolution to achieve all possible values in [0, 1). Therefore, each dimension will be considered in .001 increments. This means that one space has 1000 possible values. Two space has 1000*1000, or one million pairs. Three space has one billion possible triples, and four space has one trillion possible quadruples. Since a good congruential generator can cover one and two space, it should be able to achieve all one thousand and one million values and pairs. It will not be able to achieve all one billion triples, because the triples will fall into planes in three space. Assume that the congruential generator can achieve only 100 million of the one billion possible triples, and only 3 billion of the one trillion quadruples. What effect will this have on the PRAUT sequence?

The PRAUT generator will certainly be able to produce the 1000 values in one space. Since the addressing generator can produce all 1000 address increments, and they can occur at any table location, the PRAUT generator will be able to produce all one million pairs. In the same manner, since the addressing generator can produce any pair, the PRAUT generator, starting at any location, can produce all one billion triples. However, since the addressing generator can produce only 100 million triples, the PRAUT generator can produce only 100 billion quadruples. The PRAUT method is thus flawed by its addressing generator. Table I shows, however, that it remains better than the congruential.

TABLE I

POINTS IN N-SPACE ACHIEVABLE BY CONGRUENTIAL OR PRAUT GENERATORS

	Possible	Congruential	PRAUT
1 space	1000	1000	1000
2 space	1 million	1 million	1 million
3 space	1 billion	100 million	1 billion
4 space	1 trilion	3 billion	100 billion

The flaw is even smaller than it first appears. Although the 100 million triples produced by the congruential generator fall into planes, the quadruples produces with the PRAUT generator do not. For example, suppose that 100, 100, 100 is a possible triple for the addressing generator. This sequence may start at any location in the PRAUT table. Therefore locations 1, 101, 201, 301 are possible in sequence, as are 2, 102, 202, 302, and so forth. These location sequences obviously form patterns; however, the values stored at each location are unrelated to the location. Therefore the location sequence 1, 101, 201, 301 may produce .933, .271, .358, .647, or any other set of values. It can be seen then, that although the congruential triples fit into planes, the quadruples that they produce are scattered at random throughout four space.

The difference in magnitude of these two problems can be seen through empirical testing. Graphing only a few hundred triples from a congruential generator shows the development of distinct planes. In contrast, even 100 billion quadruples from the PRAUT generator, if somehow graphed, would be scattered across four space in a random manner. The only way to tell that not all one trillion quadruples were possible would be to graph several hundred billion of them and realize that although the pattern still looks random, it is not covering any new space. All possible quadruples, quintuples, and so on would be obtainable if the congruential sequence were shuffled, or if the PRAUT table, or sequence, were shuffled. However, this would be unnecessary unless more than a trillion number sequence was needed.

Theoretical Distribution of Sequences Generated

According to the definition of randomness, each number should be independent of all others and have an equal chance of having any of the possible values. This means that if values for . . . U_{i-2}, U_{i-1} have occurred, $Pr(u <= U_i < v) = v - u$. When using a congruential generator, if U_{i-2} was in [.50, .51), and U_{i-1} was in [.65, 66), it may not be possible for U_i to occur in [.31, .32). This is demonstrated through its failure to cover three space. This problem is smoothed out when using the PRAUT method.

In the PRAUT method, if U_{i-2} is in [.50, .51), this means that the table location could have been any of ten locations spread randomly throughout the table. This is also true for u_{i-1} occurring in [.65, 66). In order for this sequence to occur the addressing generator has to produce one of the 100 increments that define the relationships between the locations of these values. Not all of these possible increments will have the same probability of occurring, but the average of the probabilities will be very close to 100/1000, or .1, the expected overall probability.

Given that U_{i-2} and U_{i-1} have occurred as stated, there are 100 possible address increments to make U_i occur in [.31, .32). Again, not all of these increments are as probable as they should be, others will be more probable; so the average will be near .1. Thus the probability of U_{i-1}, U_i having these values, given U_{i-2}, is roughly .1 * .1 or .01, using the PRAUT method, which is the expected outcome. The probability of the above sequence occurring using a linear congruential generator was 0.

CHAPTER VII

EMPIRICAL TESTS

The tests used for this research are all standard accepted tests, discussed by such authors as Knuth (1969) and Lewis (1975). The tests used are the frequency test, serial test on pairs and on triples, minimum of five values, maximum of five values, sum of five values, and the gap test. Each will be explained here. Each uses the chi-square test to compare the sequence generated to theoretical distributions (see Appendix A for computer code of the tests).

Frequency Test

Given some number, n, of random numbers, and some values, u and v, such that $0 < = u < v < = 1$; approximately $n^*(v - u)$ values should fall in $[u, v)$. By dividing the interval $[0, 1)$ into k subintervals, and generating n pseudo-random numbers, the number that fall within each subinterval can be tallied. These totals are then compared to the expected totals for goodness of fit.

Serial Test on Pairs and Triples

These tests are very similar to the frequency test. Just as $P(u < = U < v) = (v - u)$, $P(u_1 < = U_k < v_i, u_2 < = U_{i+1} < v_2) = (v_1 - v_2)(v_2 - u_2)$, and so on. Sequences are generated, the number of pairs that fall within any set of subsequences are tallied and the totals compared to the expected totals. The process is the same when testing triples.

While the frequency test alone reveals nothing about the distribution of individual values, the addition of the serial tests greatly strengthens the evidence for, or against, uniformity.

Minimum of Five Values

Given any five random numbers, the probability that all of them are greater than some value, x, is $(1-x)^5$. Based on this expected distribution, a sequence can be generated and the minimum value of each five elements tallied. The resulting distribution is compared to the expected distribution.

Maximum of Five Values

This test is identical to the minimum test, except x^5 is used for calculation, instead of $(1-x)^5$.

Sum of Five Values

If each number generated is converted to an integer, ie. .000-.009 became 0, then the sum of each five numbers will range from 0 to 45. Each of these outcomes has a positive probability. For example, of the 100,000 possible outcomes, only (0, 0, 0, 0, 0) produces a sum of 0. It therefore has a probability of .00001. There are five ways to obtain a sum of 1, fifteen ways of obtaining a sum of 2, and so on. As with the other tests, a sequence is generated, the sums of each set of values are calculated, tallied, and the outcome compared to the theoretical distribution.

Gap Test

The probability that consecutive elements of a sequence will have values from an interval such as [.700, .800) is (.800 - .700), or 0.1. The probability of values within the interval occurring separated by one other value is 0.09. The probability of values within the interval occurring separated by two other values is 0.081, and so on. The gap test measures the intervals, or gaps, between elements with values from a selected interval, and how often each size gap occurs. The results are tallied and compared to theoretical results.

Special Tests

This battery of tests should discredit most methods of generation, however the better methods should pass. It is therefore necessary to perform more stringent testing in order to compare these methods. One may recall that a major problem with pseudo-random sequences is that the numbers are not independent of one another. In most cases each number is dependent on the previous value. If the dependency is too great, the tests already described will uncover it. The following method was developed to catch some of the methods whose dependency is a little better disguised.

Usually empirical tests are applied to all values generated by a method, however if one were to test only the values occurring after elements with values in a chosen interval, such as [.700, .799), the subsequence obtained should be perfectly random, if the entire sequence is. But if the values are dependent on the previous value, then the subsequence will not be random at all, and should fail the empirical tests too frequently. Therefore, each pseudo-random number generator tested will not only be submitted to the seven accepted tests described, but this subsequence will be determined and submitted to the same battery of tests.

CHAPTER VIII

COMPARISON OF METHODS

One may discuss the theoretical characteristics of a generator all one likes, but the only way to determine if it is as good as other methods is to empirically test them side by side and compare results. The PRAUT method will be compared to a linear congruential method (Schrange, 1979), the GFSR method (Lewis, 1973), and the Hewlett Packard (HP3000) RNG. A shuffling method will also be compared using various methods for values (Bays, 1976). The testing procedures will be discussed first, followed by the results and comparisons.

The testing procedures can be broken into three sections: 1) Standard tests performed on each generator, 2) special indepedence tests performed on each generator, and 3) special comparison of PRAUT and shuffling methods.

For the standard tests a sequence of 100,000 numbers from a generator are submitted to the seven tests discussed earlier. The result is seven chi-square values. If a chi-square value fails at the 5% or 95% level then the sequence is considered to have failed that test. Ten sequences from each generator are tested so that the expectation is one failure out of the ten sequences on each test.

Recall that the special independence test developed only tests about one out of every ten values. About 500,000 values were generated each time in order to be able to test sequences of 50,000. Again, ten sequences were tested from each generator so that one should expect about seven failures out of 70 tests.

The PRAUT method and shuffling methods may look similar because of the use of a table of numbers and random accessing, however this last set of tests shows that the similarity ends there. In the previous two sections of tests the HP generator was used with the two methods. In this section a very poor generator is used for addressing in the PRAUT method, and for the shuffling method. The intent is to show that shuffling the sequence improves it very little, while using it for addresses in the PRAUT method still produces a very good sequence.

The same tests as those in sections one and two are used on sequences of 100,000 and 50,000 numbers. Only five sequences are tested for each generator as the results are clear by that point. Failure rate is expected to be 0 or 1 out of 5.

Results and Comparisons

Testing began with the hope that all five generators would pass the first set of tests, to illustrate that they are all relatively good generators. It was then hoped that the special independence test would show some superiority of the PRAUT method. Finally, the special comparison of the PRAUT and shuffling methods was intended to show that the PRAUT is not simply a shuffling method in disguise.

The results of the standard tests are shown in Table II. The PRAUT, HP3000, shuffling, and linear congruential methods all did very well, all passing around 90% of the tests with no generator falling below 80% on any given test. Unfortunately, the GFSR generator did not fare as well. It appears that if the delay parameter was too small then the sequence failed the gap test. If the delay was large enough to pass the gap test then the Sum of 5 test had a high failure rate. As the purpose of this study is not to uncover such problems they will not be discussed in depth. Let if suffice to say that the small word size of the HP3000 and the extreme sensitivity of the input parameters for the GFSR method led to the problem. Because the rest of the results were good and GFSR results were seemingly unavoidable, analysis proceeded to the special tests.

TABLE II

RESULTS OF STANDARD STATISTICAL TESTS*

Tests	PRAUT	HP3000	Shuffle	LinCong	GFSR
Frequency	10	9	10	10	9
Pairs	9	10	9	9	8
Triples	9	10	8	9	8
Max of 5	8	10	8	10	8
Min of 5	9	9	9	10	9
Sum of 5	10	8	10	8	5
Gap	9	9	8	9	4
Average	9.14	9.28	8.86	9.28	7.28

*Numbers represent the number of sequences that passed the test out of the ten tries.

The results of the special independence tests can be found in Table III. It can easily be seen that while the PRAUT method maintained its average, the other methods did not. All four of the methods compared had tests which were passed only 70% of the time or less. Note that the GFSR method actually performed better than in the standard tests. Since its failure of the gap test was due to problems with consecutive strings of numbers, this disappeared when working with only selected values. The problem with the Sum of 5 test remained.

TABLE III

RESULTS OF SPECIALIZED STATISTICAL TESTS*

Test	PRAUT	HP3000	Shuffle	LinCong	GFSR
Frequency	9	9	9	7	10
Pairs	10	9	9	8	9
Triples	10	9	10	7	8
Max of 5	9	9	6	9	9
Min of 5	10	7	9	9	9
Sum of 5	9	9	7	8	5
Gap	8	7	9	9	9
Average	9.28	8.43	8.43	8.14	8.43

*Numbers represent the number of sequences that passed the test out of ten tries.

The results of these two sets of tests illustrate that the normally accepted empirical tests do not uncover dependencies which are known to exist. It shows that these dependencies significantly affect the behavior of the sequences generated by linear congruential and feedback shift register methods, and that shuffling the sequences does not solve the problem. Finally, these tests strengthen the claim that consecutive values generated by the PRAUT method are independent. They do not, however, prove that claim, nor do they indicate anything about the independence of nonconsecutive values.

Although the PRAUT and shuffling methods were compared in the previous tests and the PRAUT method preformed better, their differences are better illustrated through a comparison using a poor seed generator. See Table IV for results.

Looking first at the standard tests, one can see that the poor generator only passed around 50% of the time. Shuffling the sequence did not help at all as it only passed 43% of the time. On the other hand, the PRAUT sequence passed over 88% of the time and would thus be considered acceptable.

Moving on to the special tests, the poor generator failed completely, showing that each value is very dependent on the last. Shuffling the values helped, but still passed less than 40% of the time. The PRAUT method, however, showed no signs of deterioration, even at this stage. This would indicate that one does not need a very good addressing generator at all in order to obtain acceptable sequences.

Other Attributes of the Generators

While the quality of the sequences produced is the main concern in generating pseudo-random numbers, the size and speed of the algorithms are also important. The algorithm should not use excessive amounts of core storage, and because the algorithm is generally called many times during a program, it should be as fast as possible. Because the quality of the sequence is most important, it is generally assumed that if the size and speed are acceptable to the user, then the sequence is the sole determining factor in choosing a method.

None of the methods compared are excessively long. The PRAUT method is probably the longest because of the table of 1000 values

TABLE IV

RESULTS OF COMPARISON OF PRAUT AND SHUFFLING METHODS
USING A POOR SEED GENERATOR*

	Standard Tests			Special Tests		
Tests	PRAUT	Shuffle	Poor	PRAUT	Shuffle	Poor
Frequency	5	0	2	5	1	0
Pairs	5	2	4	5	4	0
Triples	4	3	3	5	3	0
Max of 5	3	0	0	4	0	0
Min of 5	5	0	1	3	1	0
Sum of 5	4	5	4	5	0	0
Gap	5	5	4	4	4	0
Average	4.43	2.14	2.57	4.43	1.86	0.0

*Numbers represent the number of sequences that passed the test out of five runs.

used. The GFSR method uses a table which may be that large, but does not have to be. It also requires much more code than the other methods. These differences, however, are unimportant, as none of the methods use an excessive amount of space.

The speeds of the generators are more easily compared, as actual times per number can be calculated. The following times pertain to an HP3000 series 30. Although speeds will differ between machines, the relative times will remain about the same.

Three of the generators require initialization. The shuffling method requires 296 CPU milliseconds to initialize a table of size ten. The PRAUT method requires 21,048 milliseconds to read the table from a file, or up to 5300 milliseconds to generate a table. The GFSR generator requires 177,500 milliseconds to initialize. This may be considered excessive as it translates to nearly three minutes.

The actual generation time per number varies a great deal as well. The fastest method was the HP3000 generator at 0.26 ms per number. Because this generator was used for addressing in the PRAUT method and for the shuffling method, these must naturally be slower. The PRAUT required 0.44 ms and shuffling required 0.48 ms. The slowest methods were the portable linear congruential at 0.72 ms, and the GFSR at 0.77. The PRAUT generator was thus the second fastest.

The poor generator was also timed, as it is merely a deformation of a fairly good generator designed by the author. (The code for this method can be found in Appendis B.) This generator requires an initialization and takes 0.27 ms per number. It therefore is nearly as fast as the HP3000 generator and could thus be used for the PRAUT method. This is useful, as it is portable, as is the PRAUT method.

CHAPTER IX

CONCLUSIONS

The main objective of this research was to develop a less deterministic method of random number generation than the accepted methods, and to show that it out performs them on empirical tests. The PRAUT generator was shown to be less deterministic than the extensively used linear congruential methods through a theoretical discussion of the properties of the sequences. The probability of an element of a sequence having a value within a specific interval is much less influenced by previous values in the PRAUT method. The PRAUT generator can produce more of the possible combinations of values than congruential methods.

The important result of a less deterministic method is that the sequences produced more closely resemble truly random sequences. Through empirical testing of the PRAUT and accepted methods, the PRAUT generator was shown to perform better. Specifically, on empirical tests of subsequences of the sequences generated, the PRAUT method passed over 90% of the time, while the other methods tested only passed between 80% and 85% of the time. Also, when a poor address generator was used for the PRAUT and shuffling methods, the PRAUT method was hardly affected, passing 88% of the time. The shuffled sequence did not even pass 50% of the time.

Based on these results, the PRAUT generator is recommended as a method of producing better pseudo-random sequences. There may be cases where only very specific attributes of a sequence are needed, which a congruential generator may provide. In these cases a congruential generator may be acceptable.

The PRAUT method fairs well on the other important factors in choosing a generator as well. While not the fastest generator, it does produce numbers faster than the shuffling, GFSR, and portable congruential generators tested, making it the fastest of the portable generators tested. The PRAUT method also requires an acceptable amount of space. The table of 1000 values used makes its the most space consuming generator tested, however, there is very little need for being as small as the congruential methods.

The PRAUT method is thus a fast generator which produces better pseudo-random sequences, through a less deterministic approach. Its only trade off is a small, but acceptable, amount of space. It is therefore recommended as a better method of generating pseudo-random sequences.

A SELECTED BIBLIOGRAPHY

Albert, A. Adrian. Modern Higher Algebra. Chicago: The University of Chicago Press, 1937.

Allard, J. L., A. R. Dobell, and T. E. Hull. "Mixed Congruential Random Number Generators for Decimal Machines." Journal of the Association for Computing Machinery, Vol. 10 (1963), pp. 131-141.

Bays, Carter and S. D. Durham. "Improving a Poor Random Number Generator." Association for Computing Machinery Transactions on Mathematical Software, Vol. 2, No. 1 (March 1976), pp. 59-64.

Bright, Herbert S. and Richard L. Enison. "Quasi-Random Number Sequences from a Long-Period TLP Generator with Remarks on Applications to Cryptography." Computing Surveys, Vol. 11 (December 1979), pp. 357-379.

Canavos, George C. "A Comparative Analysis of Two Concepts in the Generation of Uniform Pseudo-Random Numbers." Proceedings of the Association for Computing Machinery National Meeting (1967), pp. 485-491.

Coveyou, R. R. "Serial Correlation in the Generation of Psuedo-Random Numbers." Journal of the Association for Computing Machinery, Vol. 7, No. 1 (January 1960), pp. 72-74.

Coveyou, R. R. and R. D. MacPherson. "Fourier Analysis of Uniform Random Number Generators." Journal of the Association for Computing Machinery, Vol. 14, No. 1 (January 1967), pp. 100-119.

Fishman, George S. Principles of Discrete Event Simulation. New York: John Wiley and Sons, Inc. 1978.

Forsythe, George E. "Generation and Testing of Random Digits at the National Bureau of Standards, Los Angeles." Monte Carlo Methods, National Bureau of Standards, Applied Mathematics Series, Vol. 12 (1951), pp. 34-35.

Franklin, J. N. "On the Equidistribution of Psuedo-Random Numbers." Quarterly of Applied Mathematics, Vol. 16 (1958), pp. 183-188.

Greenberger, M. "An A Priori Determination of Serial Correlation in Computer Generated Random Numbers." Mathematics of Computation, Vol. 15 (1961), pp. 383-389.

Hull, T. E. and A. R. Dobell. "Random Number Generators." Journal of the Association for Computing Machinery, Vol. 4, No. 3 (July 1962), pp. 230-254.

Hull, T. E. and A. R. Dobell. "Mixed Congruential Random Number Generators for Binary Machines." Journal of the Association for Computing Machinery, Vol. 11, No. 1 (January 1964), pp. 31-40.

Knuth, Donald E. The Art of Computer Programming. Massachusetts: Addison-Wesley Pub. Co., 1969.

Kolmogorov, A. N. Foundations of the Theory of Probability. New York: Chelsea Pub. Co., 1956.

Lehmer, D. "Mathematical Methods in Large-Scale Computing Units." A-nals of the Computer Laboratory, Harvard University, Vol. 26 (1951), pp. 141-146.

Lewis, T. G. Distribution Sampling for Computer Simulation. Massachusetts: Lexington Books, 1975.

Lewis, T. G. and W. H. Payne. "Generalized Feedback Shift Register Pseudorandom Number Algorithm." Journal of the Association for Computing Machinery, Vol. 20, No. 3 (1973), pp. 456-468.

Loeve, Michel. Probability Theory. New Jersey: D. Van Nostrand Co., Inc., 1960.

MacLaren, M. Donald and George Marsaglia. "Uniform Random Number Generators." Journal of the Association for Computing Machinery, Vol. 12, No. 1 (January 1965), pp. 83-89.

Maisel, Herbert and Givliano Guagnoli. Simulation of Discrete Stochastic Systems. Chicago: Science Research Associates, Inc., 1972.

Marsaglia, G. "Random Numbers Fall Mainly on the Planes." Proceedings of the National Academy of Science, Vol. 61, No. 1 (September 1968), pp. 25-28.

Nance, Richard E. and Claude Overstreet, Jr. "A Bibliography on Random Number Generation." Computing Reviews (October 1972), pp. 495-508.

Nance, Richard E. and Claude Overstreet, Jr. "Some Experimental Observations on the Behavior of Composite Random Number Generators." Operations Research, Vol. 26, No. 5 (September-October 1978), pp. 915-935.

Payne, W. H. "Fortran Tausworthe Pseudorandom Number Generator." Communications of the Association for Computing Machinery, Vol. 13, No. 1 (January 1970), p. 57.

Renyi, Alfred. Foundations of Probability. San Francisco: Holden-Day, Inc., 1970.

Schrange, Linus. "A More Portable Fortran Random Number Generator." Association for Computing Machinery Transactions on Mathematical Software, Vol. 5, No. 2 (June 1979), pp. 132-139.

Shannon, Robert E. Systems Simulation: The Art and Science. New Jersey: Prentice-Hall, Inc., 1975.

Smith, C. S. "Multiplicative Pseudo-Random Number Generators With Prime Modulus." Journal of the Association for Computing Machinery, Vol. 18, No. 4 (October 1971), pp. 586-593.

Tausworthe, Robert C. "Random Numbers Generated by Linear Recurrence Modulo Two." Mathematics of Computation, Vol. 19 (1965), pp. 201-209.

Tootill, J. P. R., W. D. Robinson and D. J. Eagle. "An Asymptotically Random Tausworthe Sequence." Journal of the Association for Computing Machinery, Vol. 20, No. 3 (1973), pp. 469-481.

Van Gelder, A. "Some New Results in Psuedo-Random Number Generation." Journal of the Association for Computing Machinery, Vol. 14, No. 4 (October 1967), pp. 785-792.

Von Neumann, J. "Various Techniques Used In Connection With Random Digits." Monte Carlo Methods, National Bureau of Standards, Applied Mathematics Series, Vol. 12 (1951), pp. 36-38.

Westlake, W. J. "A Uniform Random Number Generator Based on the Combination of Two Congruential Generators." Journal of the Association for Computing Machinery, Vol. 14, No. 2 (April 1967), pp. 337-340.

Whittlesey, J. RB. "On the Multidimensional Uniformity of Pseudo-Random Generators." Communications of the Association for Computing Machinery, Vol. 12, No. 5 (May 1969), p. 247.

APPENDICES

APPENDIX A

FORTRAN CODE FOR EMPIRICAL TESTS

```
C   **   RANDOM NUMBER GENERATOR TESTER    **
C
C   USES: FREQUENCY, SERIAL ON PAIRS & TRIPLETS, GAP,
C         MIN,MAX & SUM OF 5.
C
      REAL RN(1000)
      INTEGER*4 LAST,TOT,ISEED
      COMMON /TEST/ RN,LAST,TOT
      SYSTEM INTRINSIC RAND1,RAND
      DISPLAY" THIS RUN IS TESTING HP'S RAND  "
      DISPLAY"                     **   "
      ISEED=RAND1
      ISEED=RAND(ISEED)*1.E+07
      DISPLAY " "
      DISPLAY " ISEED = ",ISEED
      TOT= 100000
      LAST= 0
      DO 10 I=1,100
      IF(I.EQ. 100) LAST= 1
       DO 20 J=1,1000
   20  RN(J)= RAND(ISEED)
      CALL EQUID
      CALL MMS
      CALL GAPTST
   10 CONTINUE
      STOP
      END
```

```
C   ** RANDOM NUMBER GENERATOR TESTER   **
C      SPECIAL SUBSEQUENCE TEST
C
C   USES: FREQUENCY, SERIAL ON PAIRS & TRIPLETS, GAP,
C         MIN,MAX & SUM OF 5.
C
C   TESTS ONLY THE SUBSEQUENCE OF THOSE VALUES WHICH OCCUR
C   AFTER A VALUE IN [.700,.799].
C
    REAL RN(1000)
    INTEGER*4 LAST,TOT,ISEED
    COMMON /TEST/ RN,LAST,TOT
    SYSTEM INTRINSIC RAND1,RAND
    DISPLAY" "
    DISPLAY" "
    DISPLAY" TESTING HP'S RAND USING SPECIAL TESTT
    DISPLAY"                                     **   "
    ISEED=RAND1
    ISEED=RAND(ISEED)*1.E+07
    DISPLAY " "
    DISPLAY " ISEED = ",ISEED
    TOT= 50000
    LAST= 0
    DO 10 I=1,50
    IF(I.EQ. 50) LAST= 1
    K=0
    DO 20 J=1,15000
    XX= RAND(ISEED)
 15 II=XX*10
    IF(II.NE.7) GOTO 20
    XX=RAND(ISEED)
    K=K+1
    RN(K)=XX
    IF(K.GE.1000) GOTO 25
    GOTO 15
 20 CONTINUE
 25 CALL EQUID
    CALL MMS
    CALL GAPTST
 10 CONTINUE
  5 CONTINUE
    STOP
    END
```

```
C
C     SUBROUTINE FOR FREQ., SERIAL ON PAIRS AND TRIPLES
C
      SUBROUTINE EQUID
      REAL RN(1000)
      INTEGER ONE(10),PAIR(10,10),TRIP(10,10,10)
      INTEGER*4 LAST,TOT
      COMMON /TEST/ RN,LAST,TOT
      DATA ONE/10*0/,PAIR/100*0/,TRIP/1000*0/
      DO 10 I=1,1000
       J=RN(I)*10+1
   10  ONE(J)= ONE(J)+1
C
      DO 20 I=1,999,2
       J=RN(I)*10+1
       K=RN(I+1)*10+1
   20  PAIR(J,K)=PAIR(J,K)+1
C
      DO 30 I=1,997,3
       J=RN(I)*10+1
       K=RN(I+1)*10+1
       L=RN(I+2)*1.0+1
   30  TRIP(J,K,L)=TRIP(J,K,L)+1
C
      IF(LAST.EQ.0) RETURN
      EXP1=TOT/10.
      EXP2=TOT/(2.*100.)
      EXP3=(TOT*0.999)/(3.*1000.)
      CHI1=0
      CHI2=0
      CHI3=0
      DO 110 J=1,10
       CHI1=CHI1+ (ONE(J)-EXP1)**2
       DO 120 K=1,10
        CHI2=CHI2+ (PAIR(J,K)-EXP2)**2
        DO 130 L=1,10
         CHI3=CHI3+ (TRIP(J,K,L)-EXP3)**2
  130 CONTINUE
  120 CONTINUE
  110 CONTINUE
      CHI1=CHI1/EXP1
      CHI2=CHI2/EXP2
      CHI3=CHI3/EXP3
      DISPLAY" FREQUENCY TEST = ",CHI1,"  3.3 / 16.9"
      DISPLAY" SERIAL ON PAIRS= ",CHI2,"  76  / 123  "
      DISPLAY" SERIAL ON TRIPL= ",CHI3," 927 /1073 "
      RETURN
      END
```

```
C
C   **  MIN., MAX., AND SUM OF 5 TESTS   **
C
      SUBROUTINE MMS
      REAL RN(1000),MX,MN
      REAL EXP,DIST(46)
      INTEGER MAX(10),MIN(10),SUM(46),SM
      INTEGER*4 LAST,TOT
      COMMON /TEST/ RN,LAST,TOT
      DATA MAX,MIN/20*0/,SUM/46*0/
      DATA DIST/.00001,.00005,.00015,.00035,.0007,.00126,
     1     .0021,.0033,.00495,.00715,.00996,.0134,.01745,
     2     .02205,.0271,.03246,.03795,.04335,.0484,.0528,
     3     .0563,.05875,.06,.06,.05875,.0563,.0528,.0484,
     4     .04335,.03795,.03246,.0271,.02205,.01745,.0134,
     5     .00996,.00715,.00495,.0033,.0021,.00126,.0007,
     6     .00035,.00015,.00005,.00001/
C
      DO 10 I=1,1000,5
       SM=1
       MN=1
       MX=0
       CHISM=0.
       CHIMN=0.
       CHIMX=0.
       K=I+4
       DO 20 J=I,K
        IF(RN(J).LT. MN) MN=RN(J)
        IF(RN(J).GT. MX) MX=RN(J)
   20   SM=SM+RN(J)*10
      IMX=MX**5 * 10 + 1
      MAX(IMX)= MAX(IMX)+1
      IMN=(1.-MN)**5 * 10 +1
      MIN(IMN)= MIN(IMN)+1
   10 SUM(SM)= SUM(SM)+1
C
      IF(LAST.EQ.0) RETURN
      EXP=TOT/(5.*10.)
      DO 30 I=1,10
       CHIMX=CHIMX + (MAX(I)-EXP)**2
   30  CHIMN=CHIMN + (MIN(I)-EXP)**2
      CHIMX=CHIMX/EXP
      CHIMN=CHIMN/EXP
      SMTOT=TOT/5.
      DO 40 I=1,46
       EXP=SMTOT*DIST(I)
   40  CHISM=CHISM + ((SUM(I)-EXP)**2)/EXP
C
      DISPLAY" MAX OF 5   =  ", CHIMX,"   3.3 / 16.9"
      DISPLAY" MIN OF 5   =  ", CHIMN,"   3.3 / 16.9"
      DISPLAY" SUM OF 5   =  ", CHISM,"   30.3/ 61.4"
      RETURN
      END
```

```
C
C  **    GAP TEST    **
C
      SUBROUTINE GAPTST
      REAL RN(1000)
      INTEGER GAP(22),GP,TOT
      INTEGER*4 LAST,DUMMY
      COMMON /TEST/ RN,LAST,DUMMY
      DATA GAP/22*0/,GP,EXT,CHIGP/1,0.,0./
      DATA TOT/00/
C
      DO 10 I=1,1000
      N=RN(I)*10
      IF(N .EQ. 7) GOTO 1
      GP=GP+1
      GOTO 10
   1  IF(GP .GT. 22) GP=22
      GAP(GP)= GAP(GP)+1
      GP=1
      TOT=TOT+1
  10  CONTINUE
C
      IF(LAST .EQ. 0) RETURN
      DO 20 I=1,22
      IF(I.EQ.22) GOTO 2
      EX = .1 * .9**(I-1)
      EXT=EXT+EX
      GOTO 3
   2  EX=1.-EXT
   3  EXP=EX*TOT
      CHIGP=CHIGP + ((GAP(I)-EXP)**2)/EXP
  20  CONTINUE
      DISPLAY" GAP TEST = ",CHIGP,"  11.6 / 32.7"
      RETURN
      END
```

APPENDIX B

FORTRAN CODE FOR GENERATORS TESTED

```
C    ***  THIS IS THE PRAUT GENERATOR      **
C     NOTE!! YOU MUST TYPE :FILE FTN03=TABLE1,OLD
      SUBROUTINE INITTAB
      REAL T(1000)
      INTEGER*4 IX
      COMMON /MONROE/T,IX
      ACCEPT IX
      DISPLAY "ISEED FOR RAND = ",IX
      DO 10 I=1,1000
   10 READ(3,*) T(I)
      RETURN
      END
C
      FUNCTION TGEN(IPOS)
      REAL T(1000)
      INTEGER*4 IX
      COMMON /MONROE/T,IX
      INC=RAND(IX)*1000
      IPOS=MOD(IPOS+INC,1000)+1
      TGEN=T(IPOS)
      RETURN
      END
```

```
C     THIS IS THE SHUFFLING METHOD
C     BY BAYS AND DURHAM, 1976
C
       SUBROUTINE INITSHUF(ISEED)
       REAL T(10)
       INTEGER*4 ISEED
       COMMON /SHUFF/ T,IADDR
       DO 10 I=1,10
    10 T(I)=RAND(ISEED)
       IADDR=RAND(ISEED)*10+1
       RETURN
       END
C
       FUNCTION SHUFGEN(ISEED)
       REAL T(10)
       INTEGER*4 ISEED
       COMMON /SHUFF/ T,IADDR
       SHUFGEN=T(IADDR)
       T(IADDR)=RAND(ISEED)
       IADDR=SHUFGEN*10+1
       RETURN
       END
```

```
C     THIS IS "A PORTABLE RNG" BY SHRANGE,1979
C
      FUNCTION PORT(IX)
      INTEGER*4 A,P,IX,B15,B16,XHI,XALO,LEFTLO,FHI,K
      COMMON /INIT/ A,B15,B16,P
      XHI=IX/B16
      XALO=(IX-XHI*B16)*A
      LEFTLO=XALO/B16
      FHI=XHI*A+LEFTLO
      K=FHI/B15
      IX=(((XALO-LEFTLO*B16)-P)+(FHI-K*B15)*B16)+K
      IF(IX.LT.0)IX=IX+P
      PORT=IX*4.656612875E-10
      RETURN
      END
C
      SUBROUTINE INITPORT
      INTEGER*4 A,B15,B16,P
      COMMON /INIT/ A,B15,B16,P
      A=16807
      B15=32768
      B16=65536
      P=2147483647
      RETURN
      END
```

```
C    THIS IS THE GFSR GEN     LEWIS,1975
C
      FUNCTION SETR(M,P,DELAY,Q,WDSIZE)
      INTEGER DELAY,Q,ONE,WDSIZE,M(111)
      SETR=P+1
      ONE=2**(WDSIZE-1)
      DO 1 I=1,P
    1 M(I)=ONE
      DO 4 K=1,WDSIZE
       DO 2 J=1,DELAY
    2  X=RND(M,P,Q,WDSIZE)
      KOUNT=0
       DO 3 I=1,P
       ITEMP=ONE/2**(K-1)
       ITEMP=(M(I)-M(I)/ONE*ONE)/ITEMP
       IF(ITEMP .EQ. 1) KOUNT=KOUNT+1
       IF(K .EQ. WDSIZE) GOTO 3
       M(I)=M(I)/2 + ONE
    3  CONTINUE
      IF(KOUNT .EQ. P) SETR=K
    4 CONTINUE
      DO 5 I=1,5000
       DO 5 J=1,P
    5  X=RND(M,P,Q,WDSIZE)
      RETURN
      END
C
      FUNCTION RND(M,P,Q,WDSIZE)
      LOGICAL AA,BB,LCOMPJ,LCOMPK
      INTEGER A,B,Q,WDSIZE,M(111)
      EQUIVALENCE (AA,A),(BB,B),(MCOMPJ,LCOMPJ)
      EQUIVALENCE (MCOMPK,LCOMPK)
      DATA J/0/
      N=(2**(WDSIZE-1)-1)*2+1
      J=J+1
      IF(J .GT. P) J=1
      K=J+Q
      IF(K .GT. P) K=K-P
      MCOMPJ=N-M(J)
      MCOMPK=N-M(K)
      A=M(K)
      B=M(J)
      BB=LCOMPJ .AND. AA .OR. LCOMPK .AND. BB
      M(J)=B
      RND=FLOAT(M(J))/FLOAT(N)
      RETURN
      END
```

```
C     THIS IS A BAD GENERATOR
C
      FUNCTION BD(A)
      DATA B/.90/,P/3.25/
      A = A * B
      IF(A .GT. 10000) GOTO 5
      A = A * P
    5 BD=A - JINT(A)
      RETURN
      END
```

VITA /

Mark Stephen Monroe

Candidate for the Degree of

Master of Science

Thesis: A LESS DETERMINISTIC METHOD OF RANDOM NUMBER GENERATION

Major Field: Industrial Engineering and Management

Biographical:

Personal Data: Born in Cortez, Colorado, January 8, 1959, the son of William L. and Mary L. Monroe. Married to Judith J. Lackore on May 23, 1981

Education: Graduated from the American Community School of London, England, in May, 1977; received a Bachelor of Arts Degree in Mathematics from St. Olaf College, Northfield, Minnesota, in May, 1981; received a Master of Science Degree in Industrial Engineering and Management from Oklahoma State University in December, 1983.

Professional Experience: Teaching Assistant, Department of Mathematics, St. Olaf College, January, 1980 to May, 1981. Teaching Assistant, School of Industrial Engineering and Management, Oklahoma State University, August, 1981 to May, 1983.