A FUNDAMENTAL CHANGE IN RANDOM NUMBER GENERATION

By

DAVID CHARLES MUSIL

Bachelor of Science

Marquette University

Milwaukee, Wisconsin

1988

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 1990

A FUNDAMENTAL CHANGE

IN RANDOM NUMBER

GENERATION

Thesis Approved:

Joe H. Mize
Thesis Adviser

Allen C. Ackermann

Manjunath Kamath

Norman N. Durham
Dean of the Graduate College

PREFACE

The current state of the art in computer random number generation uses a method developed over thirty-five years ago. Although much has been done to improve the sequences generated by these methods, they still have serious problems. As the results become more "random", the methodology becomes more complex. Perhaps this explains the number of poor generators in use today. The following study is an attempt to develop a fundamental change in the methodology of random number generation in an effort to both simplify and improve current methods.

I wish to thank Dr. Joseph H. Mize for his guidance as my advisor, and Dr. Allen Schuermann and Dr. Manjunath Kamath as members of my committee. I would also like to thank Dr. Schuermann for his valuable input and assistance.

I wish to extend special thanks to my wife, Maureen, for her support and encouragement, and to my parents for providing an environment making success possible.

TABLE OF CONTENTS

LIST OF TABLES

CHAPTER I

RANDOM NUMBER GENERATION: AN INTRODUCTION

Random numbers are used in many areas of the scientific community. They are used to provide probabilistic data for experiments, computer simulation, games, and modeling. They are also used in Monte Carlo methods, and for integrating higher dimensional functions. However, the numbers used for these purposes are not really random. They are actually numbers in a sequence that is generated by some type of algorithm that have the appearance of being random. For this reason, they are called pseudo-random numbers. Although they are not truly random, in many cases, they are sufficiently adequate for the purposes they serve. Since the beginning of random number generation on a computer, numerous empirical and theoretical tests have been developed by researchers to determine whether or not a particular method of generation is suitable. Most of these tests evaluate either the independence or the uniformity of the numbers in the sequence.

Ideally, a good random number generator should have the following properties:

1) Independent values
2) Uniformly distributed values

3) Fast (short generation time)

4) Long period

5) Use little computer memory.

Several methods of generation have been developed that will satisfy the first two requirements, relative to the testing procedures mentioned earlier. The fact that computers are continually getting faster eases constraints on speed.

Some of the recent research has been in an effort to improve the length of the period of the sequence. The period is defined as the number of values produced by the method before the numbers repeat in a cyclic manner. With the most common methods this length is primarily dependent on the word size of the computer, and with the existence of the 32 bit computer, the length can be as high as $2^{31}-1$ (just over 2 billion). Unfortunately, simplicity is not generally considered as one of the requirements of a good random number generator, and as the other properties improve, methods tend to become more complex. This is especially evident when trying to implement a given technique across a broad range of computers and computer languages.

CHAPTER II

WHAT IS "RANDOM"?

A common question to consider when developing these generators is what exactly is "random"? It is easy to say 'pick a number at random between 1 and 10', and then have someone do that, but how does a person determine if the number was chosen at random? The person selecting the number may have chosen his/her favorite number, the number of coins in his/her pocket, or in any other biased fashion. There is no quantifiable way to determine if the person chose the number at random or not. 'At random' and 'random behavior' are easy to illustrate, in terms of picking numbered balls out of a well mixed bowl or picking a card from a well shuffled deck. However, it is very difficult to quantify the meaning of randomness. The introduction of the concepts "well mixed", and "well shuffled" can cloud the picture even more. How does one define a well mixed deck of cards? Perhaps if one could quantitatively define the source of the random items, this definition could be then applied to the items themselves.

Applying these ideas to a sequence of numbers makes it a little easier to conceptualize. In a sequence of uniform random numbers, one would expect to find all numbers in the

sequence represented an equal number of times. The chance of any one sub-sequence occurring should be equally likely as any other. Lehmer (1951) proposed a definition of 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).

The idea of 'passing a certain number of tests' epitomizes the difficulty in quantifying randomness. How many tests should it pass? Noting that this definition was developed in 1951, almost 40 years later there is still no one well recognized quantifiable definition for a random sequence.

Perhaps there is some significance in the use of the term "unpredictable" in Lehmer's definition. If the behavior of some object was truly random, its next move would be completely unpredictable. Also, if an object's behavior was somewhat random, its next move could be predicted given enough information about its prior behavior. Given these two ideas, one may be able to determine relationships of 'more random' or 'less random' based on how predictable the behavior of different objects are. Then predictability could be measured on a scale of how much

information is needed to reasonably predict ensuing behavior.

CHAPTER III

HISTORICAL METHODS

Rolling a fair die results in a random number between one and six (inclusive). Tossing a fair coin results in a yes or a no, a + or a -, or a one or a two. These methods could be used to generate pseudo-random numbers and are generally considered as completely random, although there is some belief that they may not be (see Kolata (1986), for a discussion of randomness). These methods could be used to generate the random numbers needed in the scientific community; however, the large amount of numbers needed, and the speed at which these methods can produce values make it unrealistic. A computer is a natural choice for the generation of pseudo-random numbers because of its speed and ability to work with large amounts of data.

The first algorithm for generating pseudo-random numbers on a computer was developed by Von Neumann in 1946. This method was called the middle square or mid-square method because it involved squaring a number and then using the middle digits as the next seed (the seed is a number used to generate the next number, and generally the next number then becomes the seed for the next consecutive number). For example, using the mid-square method, the

number 4615 would be squared, resulting in 21298225. The middle digits, 2982, would be the next number in the sequence and would also be used as the seed for the next number. The numbers would then be divided by the largest number in the sequence (10000) to produce pseudo-random numbers between 0 and 1. This method had problems in that often the period was very short, and the sequence had a tendency to degenerate to 0. The sequences produced by the mid-square method also do not fair well in empirical testing used to evaluate a generator.

Another common type of generator, and perhaps the most widely used today, is the multiplicative linear congruential generator (MLCG) developed by Lehmer in 1949 (Knuth,1981). This method is based on the algorithm of multiplying a seed by a large constant, and then taking the remainder from a division operation as the next number and next seed of the sequence (see Appendix A for a sample computer program). For example, with 4 as the seed, 6 as the constant multiplier, and 13 as the divisor the function would be: $seed_{i+1} = seed_i * 6 \bmod 13$. The first five numbers of this sequence would be 4, 11, 1, 6, 10. A variation of the MLCG is to add a constant c to the product of the seed and multiplier. This is called a mixed linear congruential generator. This enables the generator to produce a value of 0, which the multiplicative method itself can not produce.

A third type of generator, although not in as wide of

use today, is the Tausworthe, or shift register generator which used strings of binary digits or bits, and an algorithm similar to the linear congruential method (see Knuth, 1981). This method complicates the calculation of numbers greatly and is much slower than other methods.

CHAPTER IV

EMPIRICAL TESTING

The majority of tests performed on pseudo-random number generators are empirical tests performed on the sequences produced. These tests are primarily based on the chi-square test of an expected distribution. Given n independent observations and k categories such that each observation can fall into one category, the chi-square statistic is calculated as follows:

$$X^2 = \Sigma \, [(\text{actual} - \text{expected})^2 / \text{expected}].$$

The actual value is the observed total count for a particular category. The expected values are determined by multiplying the probability that one observation will fall into a given category by the total number of observations. This value is then compared with an acceptable value from a chi-square table based on the α level and the number of degrees of freedom, one less than the number of categories, k (see Knuth, 1981 for a complete discussion).

Several generally accepted empirical tests have been developed based on this criterion. Some of the most common include the frequency test, the serial test on pairs, 3triples, etc., the runs test, and the gap test. A brief description of each of these tests follows.

Frequency Test

Given a sequence of n pseudo-random numbers, and two constants u and v such that $0 <= u < v <= 1$; approximately $n*(v-u)$ values should fall in the interval [u,v). A chi-square test can be performed on the sequence by dividing the interval [0,1) into k sub intervals or categories and counting the number of values that fall into each category. The expected number for each category is n/k. It is generally accepted practice that n should be great enough such that $n/k > 5$. The frequency test is a check of uniformity.

Serial Test

This test is a generalization of the frequency test to higher dimensions. In the case of the serial test on pairs, the n values in the sequence are used as n/2 pairs of points that are 'plotted' on a two-dimensional grid. The grid measures k x k categories and the number of degrees of freedom is $k^2 - 1$. This test can be performed for higher dimensions although the number of values required becomes quite large beyond three dimensions. The serial test is also a check of uniformity.

Runs Test

This test is a test of the independence assumption of the sequence of values. Runs of length i, i = 1,2,3,4,5,

and i >=6 are calculated and compared against the expected number of runs for each length. A run i is a sub-sequence of i values where the next consecutive number is greater than the preceding value. This is generally called a runs up test. A runs down test is performed in a similar manner.

Gap Test

Given a sequence of n values, the probability that consecutive numbers will be within the same sub-interval [u,v) is v - u. The probability of these numbers in the same sub-interval occurring with a gap of one value between them is

$$[\ 1 - (v - u)\ ] * (v - u).$$

A chi-square test can be performed on the actual totals of each gap length for n lengths and compared to the expected distribution. Special cases of the gap test with (u,v) = (0,½) or (½,1) are called runs below the mean and runs above the mean respectively.

These tests are primarily the criteria that individuals such as Lehmer are referring to when they mention 'passing a certain number of tests'. Ideally, a good pseudo-random number generator will pass these tests and a bad one will fail them; however, some sequences of numbers can be assembled that are not random at all but will pass many of these tests.

CHAPTER V

CURRENT RESEARCH

The most common method of pseudo-random number generation today is the linear congruential generator (LCG). Several attempts have been made to improve the numbers produced by this method by using shuffling techniques or by using two or more of these generators simultaneously. Combined generators have shown that they do increase the period beyond that of the component generators, however, this is the only mathematically demonstrated improvement (L'Ecuyer, 1988).

Much has been written on the length of the period achievable by the linear congruential algorithm. Essentially, if a proper multiplier is chosen, the result is what is called a full period generator. This generator has a period length equal to the modulo m (or m - 1 when c = 0). The algorithm used is:

$$x_i = (a * x_{i-1} + c) \bmod m.$$

Any of these generators will eventually repeat, and the longest period that can be achieved will be m, given that the following three conditions hold (Law and Kelton, 1982):

(1) The only positive integer that exactly divides both m and c is 1 (also stated, c is relatively

prime to m).

(2) If q is a prime number that divides m, then q divides (a - 1).

(3) If 4 divides m, then 4 divides (a - 1).

Often the number chosen for m is equivalent to some multiple of 2, which would replace a division by simply shifting binary digits, a technique which will improve the speed of a generator. Unfortunately, many of these generators have serious flaws in terms of uniformity or independence.

Even though the linear congruential generator is the most widely used method of producing pseudo-random numbers today, there are infinitely many different generators conceivable considering the possible values for a, c, and m. The current state of random number generation was summed up in the late 1960s by Knuth who stated "...look at the subroutine library of each computer installation in your organization, and replace the random number generators by good ones. Try to avoid being too shocked at what you find" (Knuth, 1981). Considering how much science has advanced since that time, it is a bit surprising to see similar references in the recent literature (see Park and Miller, 1988). From much of the recent research, an accepted 'minimal standard' generator has evolved (Park and Miller, 1988, Carta, 1990). The minimal standard uses a multiplicative linear congruential method (c = 0) with a

multiplier of 7^5 (16807) and a modulus of $2^{31} - 1$. This is a full period generator and has become the standard in that it is the generator employed in commercial software packages such as subroutines RNUN in the IMSL library and DRAND in the simulation language SLAM II (Park and Miller, 1988).

The choice of $2^{31} - 1$ for the modulus of the minimal standard generator is by no means arbitrary. It happens to be the largest integer possible in a 32 bit word size computer. By multiplying a seed (the previous number in the sequence) by the constant 16807, a number much greater than that allowable in a 32 bit word size can be achieved. When this occurs, the overflow is simply disregarded and what remains is the next number in the sequence. This may seem simple enough, but the difficulty arises when trying to transport this algorithm to many different computers or to different languages. Most contemporary systems will produce an integer overflow error which will, in most cases, stop execution. This is where the LCG begins to get very complicated. If the maximum integer in a system is $2^{46} - 1$ or greater, a simple implementation of the minimal standard is adequate. If the maximum integer is not that large, other programming variations must be used to provide proper implementation of the minimal standard (see Carta, 1990 for some examples). This shouldn't be much worse if one can already understand the reasoning behind the derivation of the full period generator.

CHAPTER VI

RESEARCH OBJECTIVES

Given the current state of pseudo-random number generation and the increasing complexity with which it is developing, the following three objectives were developed to define the purpose of this study:

(1) Provide an intuitive definition for the concept of random behavior as applied to number generation.

(2) Develop a method for generating random numbers (sequence generation) that is less complex primarily in terms of implementation and understanding. This method should provide repeatable sequences when initiated with the same seed.

(3) Develop a modification to the method in (2) that will provide truly random number generation at the expense of repeatable sequences with the same seed.

It is equally important that the methods developed prove comparable to, if not better than, the current methods available. The empirical tests discussed earlier, and a modified approach to the given methods will be used to evaluate the new method against the linear congruential generators, or the minimal standard.

CHAPTER VII

A FUNDAMENTAL CHANGE

The need for a simpler approach to pseudo-random number generation arises mainly from two points of view: 1) in implementing the method across all languages and computers, and 2) in providing methodology understandable to the non-expert encouraging the replacement of a poor generator with an improved one. The basis for the first point is that different computer languages handle the overflow caused by the congruential methods in different ways. Implementing this method would require knowledge of how a particular language deals with overflow, and then development of an algorithm that would provide the desired result. This reasoning is also considered in the second point, along with the knowledge necessary to create a full period generator with the use of the proper multiplier and modulus.

The proposed method was developed with a table look-up approach, but not requiring a large amount of memory. The table does not contain pseudo-random numbers between 0 and 1 (or any other range), but rather the digits 0 through 9. A pointer is used to 'move around' the table producing a stream of integers to be assembled to provide a number. The table is simply a one dimensional table, or string,

consisting of a random permutation of the digits 0 through 9. The movement of the pointer is determined by the previous numbers generated. The resolution, or number of decimal places of the numbers generated is completely determined by the algorithm. This methodology can be used to produce numbers with any number of decimal places, within the limits of the computer.

In the following discussion, the term 'number' will refer only to a pseudo-random value between 0 and 1 produced by the generator, and the term 'digit' will refer to the values in the string or table, or the positions or places to the right of the decimal within the number.

An initial attempt was made using the previous two numbers to determine the movement of the pointer. The digits of the most recent number determine the distance moved by the pointer, and the digits of the second most recent number determine the direction moved by the pointer. The two seeds are denoted by:

$$.d_1d_2d_3d_4$$

$$.a_1a_2a_3a_4$$

The pointer is positioned at the digit represented by a_4. The direction is determined by whether d_i is odd or even. An odd digit implies movement to the left in the table, and an even digit implies movement to the right (a 0 is considered even). The component digits, n_i, are determined by their corresponding d_i, and a_i.

For example, consider the string '4705631829'. Initialization of the algorithm requires two seeds. The numbers .2701 and .4398 were used for this example. This generator will produce numbers with four decimal places. The sequence generated is as follows:

.2701,
.4398,
.7289,
.1613,
.6472, etc.

With the two seeds given, the pointer is positioned at the 8 in the table or string. The pointer movement for the next digit, n_1, is made from this position. For determining the next number, d_1 is 2 (from .2701), and a_1 is 4. The pointer is moved to the right because d_1 is even and the distance moved is 4 places, so the first digit of the next number is 7. If the pointer movement goes beyond the end of the table in either direction, the movement 'wraps' around to the other end. The second digit of the next number is determined by moving a_2 places in the d_2 direction. The third and fourth digits are found similarly. When all four digits have been determined, the second seed becomes the first seed, and the number just generated becomes the second seed and the algorithm repeats.

The sequences produced by this algorithm had very short periodic cycle lengths (< 100). These sequences were not considered long enough to perform empirical testing. It was noted that whenever a 0 occurred as a pointer movement

distance a digit would repeat. This appeared to have an adverse effect on the sequence produced.

In an effort to reduce the problem caused by a zero in the seed, a third seed value was used. It was hoped that this would increase the cycle length and make the digits more uniform. The algorithm used is essentially the same, except that now the digits of the second seed and the third seed are added together to determine the distance moved by the pointer.

The new algorithm is denoted as follows:

$.d_1d_2d_3d_4$

$.a_1a_2a_3a_4$

$.b_1b_2b_3b_4$ <- Pointer

$.n_1n_2n_3n_4$ $\quad n_i = \text{Table(Pointer)} \pm (a_i + b_i) \bmod 10$

The pointer references the position in the table or string.

This addition to the previous algorithm made a tremendous improvement in the results. The cycle length produced by this 3 seed 4 decimal place method was over 41 million numbers.

Empirical testing of this generator was successful on all tests performed, which was very encouraging. This method also showed some very distinct advantages over the linear congruential generators. Any given number could be repeated in the sequence without the generator cycling, a feat not possible with the LCGs. Also, the new method can

yield a value of zero without degenerating, not possible with the multiplicative generator.

A cycle length of 41 million showed tremendous promise, but the maximum cycle length for the minimal standard is over 2 billion numbers. Given the nature of the new method for generating pseudo-random numbers, the cycle length of the algorithm must be determined empirically, i.e. by running the generator in a computer program until a cycle is detected. One fact that makes this possible is that any three numbers (seeds) in a row can uniquely define the sequence generated, and a second occurrence of these three numbers in a row indicate a periodic cycle. Even considering the speed of computers today, running these programs to determine cycle length takes a significant amount of time. Over twelve hours of computing time was required on a 12 Mhz 80286 personal computer to determine the 4 decimal place cycle length of 41 million numbers.

It seemed apparent that with this new methodology, generators with more decimal places might have longer cycle lengths. The potential existence of a mathematical relationship between the number of decimal places and the cycle length could make empirical determination of cycle lengths unnecessary. The results of empirical testing to determine maximum cycle length for each number of decimal places are given in Table I.

TABLE I

MAXIMUM CYCLE LENGTHS

Decimal Places	1	2	3	4
Max Cycle Length	6	2339	277,118	41,098,617

From this information, the thought that the cycle length, for a six or seven decimal place generator with the proposed method, would be greater than 2 billion is legitimate.

CHAPTER VIII

TESTING AND COMPARISON (AGAINST THE MINIMAL STANDARD)

When proposing any new methodology it is important to provide evidence that the new alternative is comparable, if not better, than the current state of the art. As mentioned earlier, the current minimal standard, as presented by Park and Miller (1988), is the multiplicative linear congruential generator with multiplier 16807 and modulus $2^{31} - 1$. A brief description of the set up and testing procedures precedes the results of the comparison.

Set up - All of the testing was performed on an Epson Equity II+ personal computer (80286 processor running at 12 Mhz - no math chip) using Microsoft's QuickBasic compiler, version 4.5. Appendix B contains a complete listing of the Basic code for the programs used in testing the pseudo-random number generators.

An important aspect of any number generator is that it be fast. All speed tests were performed on the various generators tested using identical conditions. Using Basic's TIMER, the time, in seconds, to generate 1000 values was determined. This time included displaying a counter on the

screen. Tests were repeated several times for consistency. Table II displays the comparison of the speed of the minimal standard versus several lengths (decimal places) of the proposed method. It should be noted that a smaller time represents a faster generator.

TABLE II

TIME (SEC.) REQUIRED TO PRODUCE
1000 RANDOM NUMBERS

MLCG	4PLACE	5PLACE	6PLACE	7PLACE
1.85	0.89	1.19	1.26	1.35

A one way analysis of variance ($\alpha = .05$) found these differences to be significant, and a test of differences between means showed a statistically significant difference between the MLCG and 7PLACE generators ($\alpha = .05$). This not only shows comparable performance in speed, but rather superior speed provided by the proposed method.

The amount of computer memory required to use a generator may not be as important as other issues, however it should be given some consideration. The linear congruential generators are very short algorithms and hence require very little memory. By comparison of the QuickBasic program files used for the speed testing, the file for the proposed method was slightly more than twice the size of the

program file for the MLCG. A true "table" look up pseudo-random number generator would require much more memory, but because the proposed method really employs a string of ten digits, its memory usage is essentially not much more than any of the congruential generators.

The most emphasis of any comparison of pseudo-random number generators would most likely fall on the empirical testing. For each of the tests described in Chapter IV, a series of 20 tests were performed on each generator. The results from these tests are presented in Table III for comparison. The table lists the number of tests that passed out of a possible 20. The parameters used for the various tests were:

Frequency test	k=100, and n=10,000,
Serial test (pairs)	k=50, and n=25,000 pairs (d=2),
Serial test (triples)	k=14, and n=27,440 triples (d=3),
Serial test (quadruples)	k=10, and n=100,000 quad. (d=4),
Runs test	n=10,000,
Gap test	k=16 (runs down).

TABLE III

EMPIRICAL TEST RESULTS

	Number of Tests Passed Out of 20	
	Proposed Method	Minimal Standard
Frequency	19	19
Pairs	18	17
Triples	18	18
Quadruples	19	20
Runs	18	19
Gap	20	20

For all of the tests, an α level of .05 was used with the null hypothesis stating the test sequence is uniformly distributed, independent, or both. The chi-square test statistic was calculated and compared to the theoretical value with k-1 (or $k^d - 1$) degrees of freedom.

A second hypothesis test was set up based on the fact that the number of successes (tests passed) is a binomial variable with a probability, $p = .95$, of success. The expected value of the number of passes is 19, but the reject region for an α of .05 is $x < 18$. This implies that any number of successes less than 18 would be considered a rare occurrence and is suspect. Examining Table III shows that only one number of successes, x, is less than 18, which is the serial test on pairs for the minimal standard. With this information, statistically, one would reject the null

hypothesis that the sequence of values are independent and identically distributed uniform random variables over two space.

As was mentioned at the close of Chapter IV, some sequences of numbers that are not random at all can and do pass many, if not all of the empirical tests. Unfortunately, even if a seemingly good generator fails a particular test, running the same test again with a different value of k or perhaps a different seed may provide a successful result. This lessens enthusiasm about the results in Table III. If anything, the results in Table III show that the proposed method is just as good as the minimal standard, or any other generator, in terms of empirical testing.

CHAPTER IX

AN ALTERNATIVE EVALUATION

In Chapter II, an intuitive definition of randomness was approached based on the idea of unpredictability. It was noted that, if a sequence of numbers were uniform random values, each number would have an equally likely chance of occurring, and in the long run, each number would occur approximately the same number of times. The term "uniform" is important here, because random numbers could come from any distribution. All of the pseudo-random number generators considered here attempt to produce uniformly distributed values on the interval [0,1].

The requirement that a pseudo-random number generator create a sequence of values where each value is equally likely is commonly tested with the frequency test discussed in Chapter V. All of the generators considered in this study, including the proposed method, show sufficient evidence of meeting this requirement at the stated levels of the given parameters. This was shown in the results of empirical testing found in Chapter VIII.

Another requirement is that all pairs of numbers should have an equally likely chance of occurring, as well as triples, quadruples, etc. This attribute of random number

generators is tested by the group of serial tests. Again, both the minimal standard and the proposed method show sufficient evidence that the sequences produced meet these requirements.

A major difference between the minimal standard and the proposed method is in what occurs following the generation of a given value in a sequence. Due to the nature of the linear congruential generators, the sequence is completely determined with the knowledge of any one number (seed). Hence, when one particular value in the sequence appears for a second time, the generator begins to cycle. This fact would be revealed by performing a serial test with a large value for k (>10000). In this scenario, the minimal standard would fail the test. The proposed generator does not have this problem because it is based on three seeds uniquely defining a sequence.

The alternative evaluation provides findings that the proposed method has a great advantage over the minimal standard generator, and assuming the relationship between unpredictability and randomness is valid, the proposed method of pseudo-random number generation produces sequences that are more random than those produced by any other method described in the current literature. The proposed method certainly fills a portion of the gap between pseudo-random and truly random number generators.

CHAPTER X

A COMPLETELY RANDOM NUMBER GENERATOR

The third objective of this study was to provide a modification of the proposed method that would render a truly random number generator, in that the sequences produced would be infinite and would not be repeatable. As with the basis of the proposed method, numerous ways exist in which modifications can be made. The foundation of the completely random number generator is the use of the clock inside the computer. In the case of a PC, a clock generates a tick roughly 18.2 times per second. Although nearly any modification would provide a similar end result, it was decided to replace the direction indicator d_1 with the tens place of the timer at the initiation of each newly generated number. This provided the ability to generate an infinite sequence of non-repeatable values with the same flexibility in terms of number of significant digits required. It is also a very simple modification.

CHAPTER XI

RECOMMENDATIONS AND CONCLUSIONS

The proposed random number generator presented in this study demonstrates a fundamental change in the way pseudo-random numbers are produced. It is simpler in implementation across many languages and machines than the minimal standard due, in part, to the way overflow is handled. The new methodology has shown to be equal, if not superior, in terms of the commonly used empirical tests, and is superior in speed to the minimal standard. The proposed method has also demonstrated potential for much longer periodic cycle lengths which also allows numbers to be duplicated exactly without causing the generator to cycle. The number 0 is also achievable without causing the generator to degenerate.

The results shown in Chapter IX on alternative evaluation of a random number generator display another fundamental change, i.e., in the way randomness of a sequence is evaluated based on predictability. The association of predictability with randomness may provide an approach in which a quantifiable measure could be applied to random number generators in the future.

The difficulties associated with the proposed method are concerned with predicting, if not determining, the cycle length of a given x-decimal place generator and what types of seeds will provide the maximum cycle length. For the string that was evaluated with the proposed three seed algorithm, maximum cycle lengths for different size (decimal places) generators were given in Table I. Not all possible seed combinations, however, result in the maximum cycle length. For a single place generator three different cycles are possible, for two places, fifteen different cycles are possible ranging in length from 4 to 2339. Eleven cycles are possible in a 3 place generator with a range of 90 to 277,118. The random permutation of the table or string used also has an impact on the make up and length of the cycles created. For these reasons, users are warned that any particular seed combination and table should be tested for cycle length before being used.

Even on a super computer, the time required to determine all possible cycles and what seeds creates them empirically would take hundreds of years. From a more practical standpoint, several seed combinations that produce a cycle greater than 2 billion numbers (5 or more decimal places) can be found empirically in a much more realistic amount of time. The fact that there are 10! possible orders for the table or string implies that there are also that many different streams available. Testing each of these

empirically is also not practical. These evaluations are left to the true number theorists.

A SELECTED BIBLIOGRAPHY

Bratley, P., Fox, B.L., and Schrage, E.L. A Guide to Simulation. 2nd Ed. Springer-Verlag, New York, 1987, pp. 192-227.

Carta, David G. "Two Fast Implementations of the 'Minimal Standard' Random Number Generator." Communications of the ACM. Vol. 33, Number 1 (January 1990), pp. 87-88.

Conant, Paul. "Routine Gives Nonrepeat Random Numbers." EDN. Vol. 29 (May 3, 1984), pp. 323.

Fishman, G.S. Principles of Discrete Event Simulation. Wiley-Interscience, New York, 1978, pp. 345-391.

Gleick, James. Chaos: Making a New Science. Viking Peguin Inc., New York, 1987.

Kac, Mark. "More on Randomness." American Scientist. Vol. 72 (May/June 1984), pp. 282-283.

Knuth, D.E. The Art of Computer Programming. Vol. 2, 2nd Ed. Addison-Wesley, Reading, Mass., 1981.

Kolata, G. "What Does It Mean to be Random?" Science. Vol. 231 (March 7, 1986), pp. 1068-70.

Law, A.M., and Kelton, W.D. Simulation Modeling and Analysis. McGraw-Hill, New York, 1982, pp. 219-239.

L'Ecuyer, P. "Efficient and Portable Combined Random Number Generators." Communications of the ACM. Vol. 31, Number 6 (June 1988), pp. 742-749, 774.

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

Marsaglia, G. "Random Numbers Fall Mainly in the Planes." Proceedings of National Academy of Science. Vol. 61, Number 1 (September 1968), pp. 25-28.

Mihram, G. Arther. Simulation: Statistical Foundations and Methodology. Academic Press, New York, 1972, pp. 147-208.

Ott, Lyman. An Introduction to Statistical Methods and Data Analysis, 2d ed. Duxbury Press, Boston, 1984, pp. 179-205.

Park, Stephen K.; Miller, Keith W. "Random Number Generators: Good Ones are Hard to Find." Communications of the ACM. Vol. 31, Number 10 (October 1988), pp. 1192-201.

Pool, Robert. "Chaos Theory: How Big an Advance?" Science. Vol. 245 (July 7, 1989), pp. 26-8.

Pritsker, A. Alan B. Introduction to Simulation and SLAM II, 3d ed. Halsted Press, New York, 1986, pp. 694-719.

Steel, Robert G. D., and Torrie, James H. Principles and Procedures of Statistics, 2d ed. McGraw-Hill, New York, 1980, pp. 58-59, 469-478.

Voelcker, John. "Picturing Randomness." IEEE Spectrum. Vol. 25 (August 1988), pp. 13.

Wallich, Paul. "Random Numbers." IEEE Spectrum. Vol. 24 (February 1987), pp. 22.

Wichmann, B.A., and Hill, I.D. "An Efficient and Portable Pseudo-random Number Generator." Applied Statistics. Vol. 31 (1982), pp. 188-190.

APPENDIXES

APPENDIX A

QUICKBASIC CODE FOR GENERATORS

4 PLACE GENERATOR

```
CLS
DEFINT A-P
N(0) = 4: N(1) = 7: N(2) = 0: N(3) = 5: N(4) = 6
N(5) = 3: N(6) = 1: N(7) = 8: N(8) = 2: N(9) = 9
D(0) = 1: D(1) = -1: D(2) = 1: D(3) = -1: D(4) = 1
D(5) = -1: D(6) = 1: D(7) = -1: D(8) = 1: D(9) = 1
T = TIMER
S1 = 4752
S2 = 3186
S3 = 5927
D1 = 4: D2 = 7: D3 = 5: D4 = 2
A1 = 3: A2 = 1: A3 = 8: A4 = 6
B1 = 5: B2 = 9: B3 = 2: B4 = 7
POINTER = 1
FOR J = 1 TO 1000
POINTER = (POINTER + 20 + D(D1) * (A1 + B1)) MOD 10
D1 = A1: A1 = B1: B1 = N(POINTER)
POINTER = (POINTER + 20 + D(D2) * (A2 + B2)) MOD 10
D2 = A2: A2 = B2: B2 = N(POINTER)
POINTER = (POINTER + 20 + D(D3) * (A3 + B3)) MOD 10
D3 = A3: A3 = B3: B3 = N(POINTER)
POINTER = (POINTER + 20 + D(D4) * (A4 + B4)) MOD 10
D4 = A4: A4 = B4: B4 = N(POINTER)
CNUM = B1 * 1000 + B2 * 100 + B3 * 10 + B4
QNUM = CNUM * .0001
LOCATE 5, 5: PRINT J
NEXT
PRINT TIMER - T
```

7 PLACE GENERATOR

```
CLS
DEFINT A-P
DEFLNG B-C
N(0) = 4: N(1) = 7: N(2) = 0: N(3) = 5: N(4) = 6
N(5) = 3: N(6) = 1: N(7) = 8: N(8) = 2: N(9) = 9
FOR I = 0 TO 8 STEP 2: D(I) = 1: NEXT
FOR I = 1 TO 9 STEP 2: D(I) = -1: NEXT
T = TIMER
S1 = 4752712
S2 = 3186337
S3 = 5927976
D1 = 4: D2 = 7: D3 = 5: D4 = 2: D5 = 7: D6 = 1: D7 = 2
A1 = 3: A2 = 1: A3 = 8: A4 = 6: A5 = 3: A6 = 3: A7 = 7
B1 = 5: B2 = 9: B3 = 2: B4 = 7: B5 = 9: B6 = 7: B7 = 6
POINTER = 4
FOR J = 1 TO 1000
POINTER = (POINTER + 20 + D(D1) * (A1 + B1)) MOD 10
D1 = A1: A1 = B1: B1 = N(POINTER)
POINTER = (POINTER + 20 + D(D2) * (A2 + B2)) MOD 10
D2 = A2: A2 = B2: B2 = N(POINTER)
POINTER = (POINTER + 20 + D(D3) * (A3 + B3)) MOD 10
D3 = A3: A3 = B3: B3 = N(POINTER)
POINTER = (POINTER + 20 + D(D4) * (A4 + B4)) MOD 10
D4 = A4: A4 = B4: B4 = N(POINTER)
POINTER = (POINTER + 20 + D(D5) * (A5 + B5)) MOD 10
D5 = A5: A5 = B5: B5 = N(POINTER)
POINTER = (POINTER + 20 + D(D6) * (A6 + B6)) MOD 10
D6 = A6: A6 = B6: B6 = N(POINTER)
POINTER = (POINTER + 20 + D(D7) * (A7 + B7)) MOD 10
D7 = A7: A7 = B7: B7 = N(POINTER)
CNUM = B1*1000000+B2*100000+B3*10000+B4*1000+B5*100+B6*10+B7
QNUM = CNUM * .0000001
LOCATE 5, 5: PRINT J
NEXT
PRINT TIMER - T
```

MINIMAL STANDARD GENERATOR

```
CLS
DEFINT J
A = 1 / 2147483647
T = TIMER
SEED = 18235
M = 2 ^ 31 - 1
FOR J = 1 TO 1000
TEMP = SEED * 16807
SEED = TEMP - M * INT(TEMP / M)
NUMBER = SEED * A
LOCATE 5, 5: PRINT J
NEXT
PRINT TIMER - T
```

APPENDIX B

QUICKBASIC CODE FOR

EMPIRICAL TESTS

PROPOSED GENERATOR TESTS

FREQUENCY TEST

```
CLS
DEFINT A-P
DEFLNG B-C
DIM A(100)
N(0) = 4: N(1) = 7: N(2) = 0: N(3) = 5: N(4) = 6
N(5) = 3: N(6) = 1: N(7) = 8: N(8) = 2: N(9) = 9
D(0) = 1: D(1) = -1: D(2) = 1: D(3) = -1: D(4) = 1
D(5) = -1: D(6) = 1: D(7) = -1: D(8) = 1: D(9) = 1
S1 = 4752712: S2 = 3186335: S3 = 5927978
D1 = 4: D2 = 7: D3 = 5: D4 = 2: D5 = 7: D6 = 1: D7 = 2
A1 = 3: A2 = 1: A3 = 8: A4 = 6: A5 = 3: A6 = 3: A7 = 5
B1 = 5: B2 = 9: B3 = 2: B4 = 7: B5 = 9: B6 = 7: B7 = 8
POINTER = 7
FOR M = 1 TO 20
FOR J = 1 TO 10000
POINTER = (POINTER + 20 + D(D1) * (A1 + B1)) MOD 10
D1 = A1: A1 = B1: B1 = N(POINTER)
POINTER = (POINTER + 20 + D(D2) * (A2 + B2)) MOD 10
D2 = A2: A2 = B2: B2 = N(POINTER)
POINTER = (POINTER + 20 + D(D3) * (A3 + B3)) MOD 10
D3 = A3: A3 = B3: B3 = N(POINTER)
POINTER = (POINTER + 20 + D(D4) * (A4 + B4)) MOD 10
D4 = A4: A4 = B4: B4 = N(POINTER)
POINTER = (POINTER + 20 + D(D5) * (A5 + B5)) MOD 10
D5 = A5: A5 = B5: B5 = N(POINTER)
POINTER = (POINTER + 20 + D(D6) * (A6 + B6)) MOD 10
D6 = A6: A6 = B6: B6 = N(POINTER)
POINTER = (POINTER + 20 + D(D7) * (A7 + B7)) MOD 10
D7 = A7: A7 = B7: B7 = N(POINTER)
CNUM = B1*1000000+B2*100000+B3*10000+B4*1000+B5*100+B6*10+B7
NUM = INT(CNUM / 100000!)
A(NUM) = A(NUM) + 1
LOCATE 2, 5: PRINT J
NEXT
FOR I = 0 TO 99
SUM = SUM + ((A(I) - 100) ^ 2) / 100
A(I) = 0
NEXT
LOCATE 3 + M, 5
PRINT SUM
SUM = 0: NEXT
```

SERIAL TEST - PAIRS

```
CLS
DEFINT A-P
DEFLNG B-C
DIM A(50, 50)
N(0) = 4: N(1) = 7: N(2) = 0: N(3) = 5: N(4) = 6
N(5) = 3: N(6) = 1: N(7) = 8: N(8) = 2: N(9) = 9
FOR I = 0 TO 8 STEP 2: D(I) = 1: NEXT
FOR I = 1 TO 9 STEP 2: D(I) = -1: NEXT
S1 = 4752712: S2 = 3186337: S3 = 5927976
D1 = 4: D2 = 7: D3 = 5: D4 = 2: D5 = 7: D6 = 1: D7 = 2
A1 = 3: A2 = 1: A3 = 8: A4 = 6: A5 = 3: A6 = 3: A7 = 7
B1 = 5: B2 = 9: B3 = 2: B4 = 7: B5 = 9: B6 = 7: B7 = 6
POINTER = 4
FOR M = 1 TO 20
FOR J = 1 TO 25000
FOR E = 1 TO 2
POINTER = (POINTER + 20 + D(D1) * (A1 + B1)) MOD 10
D1 = A1: A1 = B1: B1 = N(POINTER)
POINTER = (POINTER + 20 + D(D2) * (A2 + B2)) MOD 10
D2 = A2: A2 = B2: B2 = N(POINTER)
POINTER = (POINTER + 20 + D(D3) * (A3 + B3)) MOD 10
D3 = A3: A3 = B3: B3 = N(POINTER)
POINTER = (POINTER + 20 + D(D4) * (A4 + B4)) MOD 10
D4 = A4: A4 = B4: B4 = N(POINTER)
POINTER = (POINTER + 20 + D(D5) * (A5 + B5)) MOD 10
D5 = A5: A5 = B5: B5 = N(POINTER)
POINTER = (POINTER + 20 + D(D6) * (A6 + B6)) MOD 10
D6 = A6: A6 = B6: B6 = N(POINTER)
POINTER = (POINTER + 20 + D(D7) * (A7 + B7)) MOD 10
D7 = A7: A7 = B7: B7 = N(POINTER)
CNUM = B1*1000000+B2*100000+B3*10000+B4*1000+B5*100+B6*10+B7
QNUM = CNUM / 10000000
G(E) = INT(QNUM * 50)
NEXT
A(G(1), G(2)) = A(G(1), G(2)) + 1
LOCATE 2, 5: PRINT J
NEXT
FOR I = 0 TO 49
FOR L = 0 TO 49
SUM = SUM + ((A(I, L) - 10) ^ 2) / 10
A(I, L) = 0
NEXT: NEXT
LOCATE 3 + M, 5: PRINT SUM
SUM = 0:NEXT
```

SERIAL TEST - TRIPLES

```
CLS
DEFINT A-P
DEFLNG B-C, J
DIM A(14, 14, 14)
DEFDBL Q
N(0) = 4: N(1) = 7: N(2) = 0: N(3) = 5: N(4) = 6
N(5) = 3: N(6) = 1: N(7) = 8: N(8) = 2: N(9) = 9
FOR I = 0 TO 8 STEP 2: D(I) = 1: NEXT
FOR I = 1 TO 9 STEP 2: D(I) = -1: NEXT
S1 = 4752712: S2 = 3186327: S3 = 5927975
D1 = 4: D2 = 7: D3 = 5: D4 = 2: D5 = 7: D6 = 1: D7 = 2
A1 = 3: A2 = 1: A3 = 8: A4 = 6: A5 = 3: A6 = 2: A7 = 7
B1 = 5: B2 = 9: B3 = 2: B4 = 7: B5 = 9: B6 = 7: B7 = 5
POINTER = 3
FOR M = 1 TO 20
FOR J = 1 TO 27440
FOR E = 1 TO 3
POINTER = (POINTER + 20 + D(D1) * (A1 + B1)) MOD 10
D1 = A1: A1 = B1: B1 = N(POINTER)
POINTER = (POINTER + 20 + D(D2) * (A2 + B2)) MOD 10
D2 = A2: A2 = B2: B2 = N(POINTER)
POINTER = (POINTER + 20 + D(D3) * (A3 + B3)) MOD 10
D3 = A3: A3 = B3: B3 = N(POINTER)
POINTER = (POINTER + 20 + D(D4) * (A4 + B4)) MOD 10
D4 = A4: A4 = B4: B4 = N(POINTER)
POINTER = (POINTER + 20 + D(D5) * (A5 + B5)) MOD 10
D5 = A5: A5 = B5: B5 = N(POINTER)
POINTER = (POINTER + 20 + D(D6) * (A6 + B6)) MOD 10
D6 = A6: A6 = B6: B6 = N(POINTER)
POINTER = (POINTER + 20 + D(D7) * (A7 + B7)) MOD 10
D7 = A7: A7 = B7: B7 = N(POINTER)
CNUM = B1*1000000+B2*100000+B3*10000+B4*1000+B5*100+B6*10+B7
QNUM = CNUM / 1E+07
G(E) = INT(QNUM * 14)
NEXT
A(G(1), G(2), G(3)) = A(G(1), G(2), G(3)) + 1
NEXT
FOR I = 0 TO 13
FOR K = 0 TO 13
FOR L = 0 TO 13
SUM = SUM + ((A(I, K, L) - 10) ^ 2) / 10
A(I, K, L) = 0
NEXT: NEXT: NEXT
LOCATE 3 + M, 5: PRINT SUM: SUM = 0: NEXT
```

SERIAL TEST - QUADS

```
CLS
DEFINT A-P
DEFLNG B-C, J
DIM A(10, 10, 10, 10)
DEFDBL Q
N(0) = 4: N(1) = 7: N(2) = 0: N(3) = 5: N(4) = 6
N(5) = 3: N(6) = 1: N(7) = 8: N(8) = 2: N(9) = 9
FOR I = 0 TO 8 STEP 2: D(I) = 1: NEXT
FOR I = 1 TO 9 STEP 2: D(I) = -1: NEXT
S1 = 4752712: S2 = 3186337: S3 = 5927976
D1 = 4: D2 = 7: D3 = 5: D4 = 2: D5 = 7: D6 = 1: D7 = 2
A1 = 3: A2 = 1: A3 = 8: A4 = 6: A5 = 3: A6 = 3: A7 = 7
B1 = 5: B2 = 9: B3 = 2: B4 = 7: B5 = 9: B6 = 7: B7 = 6
POINTER = 4
FOR M = 1 TO 20
FOR J = 1 TO 100000
FOR E = 1 TO 4
POINTER = (POINTER + 20 + D(D1) * (A1 + B1)) MOD 10
D1 = A1: A1 = B1: B1 = N(POINTER)
POINTER = (POINTER + 20 + D(D2) * (A2 + B2)) MOD 10
D2 = A2: A2 = B2: B2 = N(POINTER)
POINTER = (POINTER + 20 + D(D3) * (A3 + B3)) MOD 10
D3 = A3: A3 = B3: B3 = N(POINTER)
POINTER = (POINTER + 20 + D(D4) * (A4 + B4)) MOD 10
D4 = A4: A4 = B4: B4 = N(POINTER)
POINTER = (POINTER + 20 + D(D5) * (A5 + B5)) MOD 10
D5 = A5: A5 = B5: B5 = N(POINTER)
POINTER = (POINTER + 20 + D(D6) * (A6 + B6)) MOD 10
D6 = A6: A6 = B6: B6 = N(POINTER)
POINTER = (POINTER + 20 + D(D7) * (A7 + B7)) MOD 10
D7 = A7: A7 = B7: B7 = N(POINTER)
CNUM = B1*1000000+B2*100000+B3*10000+B4*1000+B5*100+B6*10+B7
QNUM = CNUM / 1E+07
G(E) = INT(QNUM * 10)
NEXT
A(G(1), G(2), G(3), G(4)) = A(G(1), G(2), G(3), G(4)) + 1
LOCATE 2, 5: PRINT J
NEXT
FOR H = 0 TO 9:FOR I = 0 TO 9
FOR K = 0 TO 9:FOR L = 0 TO 9
SUM = SUM + ((A(H, I, K, L) - 10) ^ 2) / 10
A(H, I, K, L) = 0
NEXT: NEXT: NEXT: NEXT
LOCATE 3 + M, 5: PRINT SUM: SUM = 0: NEXT
```

RUNS TEST

```
CLS
DEFINT C-R
DEFLNG C
DEFDBL B, S-T
DIM A(6, 6)
N(0) = 4: N(1) = 7: N(2) = 0: N(3) = 5: N(4) = 6
N(5) = 3: N(6) = 1: N(7) = 8: N(8) = 2: N(9) = 9
FOR I = 0 TO 8 STEP 2: D(I) = 1: NEXT
FOR I = 1 TO 9 STEP 2: D(I) = -1: NEXT
FOR I = 1 TO 6
FOR K = 1 TO 6
READ A(I, K)
NEXT: NEXT
DATA 4529.4, 9044.9, 13568,  18091,  22615, 27892
DATA 9044.9, 18097,  27139,  36187,  45234, 55789
DATA 13568, 27139,   40721,  54281,  67852, 83685
DATA 18091, 36187,   54281,  72414,  90470, 111580
DATA 22615, 45234,   67852,  90470, 113262, 139476
DATA 27892, 55789,   83685, 111580, 139476, 172860
B(1) = 1 / 6: B(2) = 5 / 24: B(3) = 11 / 120
B(4) = 19 / 720: B(5) = 29 / 5040: B(6) = 1 / 840
S1 = 4752712
S2 = 3186337
S3 = 5927976
D1 = 4: D2 = 7: D3 = 5: D4 = 2: D5 = 7: D6 = 1: D7 = 2
A1 = 3: A2 = 1: A3 = 8: A4 = 6: A5 = 3: A6 = 3: A7 = 7
B1 = 5: B2 = 9: B3 = 2: B4 = 7: B5 = 9: B6 = 7: B7 = 6
POINTER = 4
FOR M = 1 TO 20
SUM = 0
L = 1
FOR J = 1 TO 10000
POINTER = (POINTER + 20 + D(D1) * (A1 + B1)) MOD 10
D1 = A1: A1 = B1: B1 = N(POINTER)
POINTER = (POINTER + 20 + D(D2) * (A2 + B2)) MOD 10
D2 = A2: A2 = B2: B2 = N(POINTER)
POINTER = (POINTER + 20 + D(D3) * (A3 + B3)) MOD 10
D3 = A3: A3 = B3: B3 = N(POINTER)
POINTER = (POINTER + 20 + D(D4) * (A4 + B4)) MOD 10
D4 = A4: A4 = B4: B4 = N(POINTER)
POINTER = (POINTER + 20 + D(D5) * (A5 + B5)) MOD 10
D5 = A5: A5 = B5: B5 = N(POINTER)
POINTER = (POINTER + 20 + D(D6) * (A6 + B6)) MOD 10
D6 = A6: A6 = B6: B6 = N(POINTER)
```

```
POINTER = (POINTER + 20 + D(D7) * (A7 + B7)) MOD 10
D7 = A7: A7 = B7: B7 = N(POINTER)
CNUM=B1*1000000!+B2*100000!+B3*10000!+B4*1000+B5*100+B6*10+B
7 IF CNUM < S3 THEN
 L = L + 1
ELSE
 IF L > 6 THEN L = 6
 R(L) = R(L) + 1
 L = 1
END IF
S3 = CNUM
LOCATE 2, 5: PRINT J
NEXT
FOR I = 1 TO 6
FOR K = 1 TO 6
TEMP = A(I, K)*(R(I) - 10000 * B(I)) * (R(K) - 10000 * B(K))
SUM = SUM + TEMP
NEXT
NEXT
FOR I = 1 TO 6: R(I) = 0: NEXT
SUM = SUM / 10000
LOCATE 3 + M, 5: PRINT SUM
NEXT
```

GAP TEST

```
CLS
DEFINT A-P
DEFLNG B-C
DIM A(15)
N(0) = 4: N(1) = 7: N(2) = 0: N(3) = 5: N(4) = 6
N(5) = 3: N(6) = 1: N(7) = 8: N(8) = 2: N(9) = 9
FOR I = 0 TO 8 STEP 2: D(I) = 1: NEXT
FOR I = 1 TO 9 STEP 2: D(I) = -1: NEXT
S1 = 4752712: S2 = 3186337: S3 = 5927976
D1 = 4: D2 = 7: D3 = 5: D4 = 2: D5 = 7: D6 = 1: D7 = 2
A1 = 3: A2 = 1: A3 = 8: A4 = 6: A5 = 3: A6 = 3: A7 = 7
B1 = 5: B2 = 9: B3 = 2: B4 = 7: B5 = 9: B6 = 7: B7 = 6
POINTER = 4
FOR M = 1 TO 20
FOR J = 1 TO 10000
R = 0
GENERATE:
POINTER = (POINTER + 20 + D(D1) * (A1 + B1)) MOD 10
D1 = A1: A1 = B1: B1 = N(POINTER)
POINTER = (POINTER + 20 + D(D2) * (A2 + B2)) MOD 10
D2 = A2: A2 = B2: B2 = N(POINTER)
POINTER = (POINTER + 20 + D(D3) * (A3 + B3)) MOD 10
D3 = A3: A3 = B3: B3 = N(POINTER)
POINTER = (POINTER + 20 + D(D4) * (A4 + B4)) MOD 10
D4 = A4: A4 = B4: B4 = N(POINTER)
POINTER = (POINTER + 20 + D(D5) * (A5 + B5)) MOD 10
D5 = A5: A5 = B5: B5 = N(POINTER)
POINTER = (POINTER + 20 + D(D6) * (A6 + B6)) MOD 10
D6 = A6: A6 = B6: B6 = N(POINTER)
POINTER = (POINTER + 20 + D(D7) * (A7 + B7)) MOD 10
D7 = A7: A7 = B7: B7 = N(POINTER)
CNUM = B1*1000000+B2*100000+B3*10000+B4*1000+B5*100+B6*10+B7
IF CNUM < 8000000 THEN R = R + 1: GOTO GENERATE
IF R > 15 THEN R = 15
A(R) = A(R) + 1: LOCATE 2, 5: PRINT J
NEXT
FOR R = 0 TO 14
 PROB = 10000 * (.2 * .8 ^ R)
 SUM = SUM + ((A(R) - PROB) ^ 2) / PROB
 A(R) = 0
NEXT
PROB = 10000 * (.8 ^ 15)
SUM = SUM + ((A(15) - PROB) ^ 2) / PROB: A(15) = 0
LOCATE 3 + M, 5: PRINT SUM: SUM = 0:NEXT
```

MINIMAL STANDARD GENERATOR TESTS

FREQUENCY TEST

```
CLS
DEFINT A-J
DIM A(100)
SEED = 982357
M = 2 ^ 31 - 1
FOR K = 1 TO 20
FOR J = 1 TO 10000
TEMP = SEED * 16807
SEED = TEMP - M * INT(TEMP / M)
NUMBER = SEED / 2.147484E+09
JNUM = INT(NUMBER * 100)
A(JNUM) = A(JNUM) + 1
LOCATE 2, 5: PRINT J
NEXT
FOR I = 0 TO 99
SUM = SUM + ((A(I) - 100) ^ 2) / 100
A(I) = 0
NEXT
LOCATE 3 + K, 5
PRINT SUM
SUM = 0
NEXT
```

SERIAL TEST - PAIRS

```
CLS
DEFINT A-J
DIM A(50, 50)
SEED = 982357
M = 2 ^ 31 - 1
FOR K = 1 TO 20
FOR J = 1 TO 25000
FOR E = 1 TO 2
TEMP = SEED * 16807
SEED = TEMP - M * INT(TEMP / M)
NUMBER = SEED / 2.147484E+09
JNUM(E) = INT(NUMBER * 50)
NEXT
A(JNUM(1), JNUM(2)) = A(JNUM(1), JNUM(2)) + 1
LOCATE 2, 5: PRINT J
NEXT
FOR I = 0 TO 49
FOR L = 0 TO 49
SUM = SUM + ((A(I, L) - 10) ^ 2) / 10
A(I, L) = 0
NEXT: NEXT
LOCATE 3 + K, 5
PRINT SUM
SUM = 0
NEXT
```

SERIAL TEST - TRIPLES

```
CLS
DEFINT A-J
DEFLNG J
DIM A(25, 25, 25)
SEED = 982357
M = 2 ^ 31 - 1
FOR K = 1 TO 20
FOR J = 1 TO 156250
FOR E = 1 TO 3
TEMP = SEED * 16807
SEED = TEMP - M * INT(TEMP / M)
NUMBER = SEED / 2.147484E+09
JNUM(E) = INT(NUMBER * 25)
NEXT
A(JNUM(1), JNUM(2), JNUM(3)) = A(JNUM(1), JNUM(2),
JNUM(3))+1
LOCATE 2, 5: PRINT J
NEXT
FOR I = 0 TO 24
FOR H = 0 TO 24
FOR L = 0 TO 24
SUM = SUM + ((A(I, H, L) - 10) ^ 2) / 10
A(I, H, L) = 0
NEXT: NEXT: NEXT
LOCATE 3 + K, 5
PRINT SUM
SUM = 0
NEXT
```

SERIAL TEST - QUADS

```
CLS
DEFINT A-L
DEFLNG J
DIM A(10, 10, 10, 10)
SEED = 982357
M = 2 ^ 31 - 1
FOR K = 1 TO 20
FOR J = 1 TO 100000
FOR E = 1 TO 4
TEMP = SEED * 16807
SEED = TEMP - M * INT(TEMP / M)
NUMBER = SEED / 2.147484E+09
JN(E) = INT(NUMBER * 10)
NEXT
A(JN(1), JN(2), JN(3), JN(4)) = A(JN(1), JN(2), JN(3),
JN(4)) + 1
LOCATE 2, 5: PRINT J
NEXT
FOR G = 0 TO 9
FOR H = 0 TO 9
FOR I = 0 TO 9
FOR L = 0 TO 9
SUM = SUM + ((A(G, H, I, L) - 10) ^ 2) / 10
A(G, H, I, L) = 0
NEXT
NEXT
NEXT
NEXT
LOCATE 3 + K, 5
PRINT SUM
SUM = 0
NEXT
```

RUNS TEST

```
CLS
DEFINT C-R
DEFLNG C
DIM A(6, 6)
FOR I = 1 TO 6: FOR K = 1 TO 6: READ A(I, K): NEXT: NEXT
DATA 4529.4, 9044.9, 13568,  18091,  22615, 27892
DATA 9044.9, 18097,  27139,  36187,  45234, 55789
DATA 13568, 27139,   40721,  54281,  67852, 83685
DATA 18091, 36187,   54281,  72414,  90470, 111580
DATA 22615, 45234,   67852,  90470, 113262, 139476
DATA 27892, 55789,   83685, 111580, 139476, 172860
B(1) = 1 / 6: B(2) = 5 / 24: B(3) = 11 / 120
B(4) = 19 / 720: B(5) = 29 / 5040: B(6) = 1 / 840
N = 2 ^ 31 - 1
SEED = 18235
S1 = SEED
FOR M = 1 TO 20
SUM = 0
L = 1
FOR J = 1 TO 10000
TEMP = SEED * 16807
SEED = TEMP - N * INT(TEMP / N)
NUMBER = SEED / N
IF NUMBER < S1 THEN
 L = L + 1
ELSE
 IF L > 6 THEN L = 6
 R(L) = R(L) + 1
 L = 1
END IF
S1 = NUMBER
LOCATE 2, 5: PRINT J
NEXT
FOR I = 1 TO 6
FOR K = 1 TO 6
TEMP2 = A(I, K) * (R(I) - 10000 * B(I)) * (R(K) - 10000 *
B(K))
SUM = SUM + TEMP2
NEXT:NEXT
FOR I = 1 TO 6: R(I) = 0: NEXT
SUM = SUM / 10000
LOCATE 3 + M, 5: PRINT SUM
NEXT
```

GAP TEST

```
CLS
DEFINT A-J
DIM A(15)
SEED = 982357
M = 2 ^ 31 - 1
FOR K = 1 TO 20
FOR J = 1 TO 10000
R = 0
GENERATE:
TEMP = SEED * 16807
SEED = TEMP - M * INT(TEMP / M)
NUMBER = SEED / 2.147484E+09
IF NUMBER < .8 THEN R = R + 1: GOTO GENERATE
IF R > 15 THEN R = 15
A(R) = A(R) + 1
LOCATE 2, 5: PRINT J
NEXT
FOR R = 0 TO 14
PROB = 10000 * (.2 * .8 ^ R)
SUM = SUM + ((A(R) - PROB) ^ 2) / PROB
A(R) = 0
NEXT
PROB = 10000 * (.8 ^ 15)
SUM = SUM + ((A(15) - PROB) ^ 2) / PROB
A(15) = 0
LOCATE 3 + K, 5: PRINT SUM
SUM = 0
NEXT
```

VITA

David Charles Musil

Candidate for the Degree of

Master of Science

Thesis: A FUNDAMENTAL CHANGE IN RANDOM NUMBER GENERATION

Major Field: Industrial Engineering and Management

Biographical:

Personal Data:
Born in Sheboygan Falls, Wisconsin, May 25, 1966, the son of Leland C. and Shirley M. Musil. Married to Maureen K. Albers on June 10, 1989.

Education:
Graduated from Sheboygan Falls High School in June, 1984; received Bachelor of Science Degree in Industrial Engineering from Marquette University, Milwaukee, Wisconsin, in May, 1988; received Master of Science in Industrial Engineering and Management from Oklahoma State University, Stillwater, Oklahoma, in December, 1990.

Professional Experience:
Scientist, Kimberly Clark Corporation, May, 1988 to August, 1989; Co-op Engineer, Kohler Company, January, 1987 to August, 1987; Engineering Intern, General Motors Corporation, May, 1986 to August, 1986.

Publications:
Musil, David C., and Akbay, Kunter S., "Computer Simulation Modeling of a Flexible Manufacturing Cell." Industrial Engineering Magazine. November, 1989. Volume 21, Number 11.